# CRiSP Macro Primitives

This document provides reference material on the primitive functions built into CRiSP which are available to macro programmers.

There are a lot of functions and facilities available. Many functions are similar to ISO C, but there are a lot of functions for manipulating the basic data structures available inside the editor. These cover such areas as buffer management, files, window manipulation, callbacks and GUI programming.

→ Alphabetical function summary(pg. 28).

→ Macro Programmers Guide

Available function categories:

## Arithmetic and Logical

These functions allow you to perform arithmetic and logical operations. Most of these functions are operators, rather than functions.

Some of the operators are overloaded, for example, the plus operator can be used to add strings, integers and lists. Automatic type casting is provided where relevant.

| | |
|---|---|
| !(pg. **Error! Bookmark not defined.**). | Logical not operator(pg. 38). |
| !=(pg. **Error! Bookmark not defined.**). | Inequality operator(pg. 38). |
| %(pg. **Error! Bookmark not defined.**). | Integer modulus(pg. 38). |
| %=(pg. **Error! Bookmark not defined.**). | Integer modulus assignment(pg. 38). |
| &(pg. **Error! Bookmark not defined.**). | Logical and(pg. **Error! Bookmark not defined.**). |
| &=(pg. **Error! Bookmark not defined.**). | Logical and assignment operator(pg. **Error! Bookmark not defined.**). |
| &&(pg. **Error! Bookmark not defined.**). | Conditional and operator(pg. **Error! Bookmark not defined.**). |
| *(pg. **Error! Bookmark not defined.**). | Multiply/string replicate(pg. 39). |
| *=(pg. **Error! Bookmark not defined.**). | Multiply and assign value(pg. 39). |
| +(pg. **Error! Bookmark not defined.**). | Arithmetic, string or list addition(pg. **Error! Bookmark not defined.**). |
| ++(pg. **Error! Bookmark not defined.**). | Pre-increment variable(pg. **Error! Bookmark not defined.**). |
| +=(pg. **Error! Bookmark not defined.**). | Add assignment operator(pg. **Error! Bookmark not defined.**). |
| -(pg. **Error! Bookmark not defined.**). | Subtract operator(pg. **Error! Bookmark not defined.**). |
| --(pg. **Error! Bookmark not defined.**). | Pre-decrement(pg. **Error! Bookmark not defined.**). |
| -=(pg. **Error! Bookmark not defined.**). | Subtract assignment operator(pg. **Error! Bookmark not defined.**). |
| /(pg. **Error! Bookmark not defined.**). | Divide operator(pg. **Error! Bookmark not defined.**). |
| /=(pg. **Error! Bookmark not defined.**). | Divide assignment operator(pg. **Error! Bookmark not defined.**). |
| <(pg. **Error! Bookmark not defined.**). | Less than(pg. **Error! Bookmark not defined.**). |
| <<(pg. **Error! Bookmark not defined.**). | Bit left-shift operator(pg. **Error! Bookmark not defined.**). |
| <=(pg. **Error! Bookmark not defined.**). | Less than or equal(pg. **Error! Bookmark not defined.**). |
| =(pg. **Error! Bookmark not defined.**). | Assignment operator(pg. 42). |
| <<=(pg. **Error! Bookmark not defined.**). | Bit left-shift assignment operator(pg. **Error! Bookmark not defined.**). |
| ==(pg. **Error! Bookmark not defined.**). | Equality(pg. 42). |
| >(pg. **Error! Bookmark not defined.**). | Greater than(pg. 42). |
| >=(pg. **Error! Bookmark not defined.**). | Greater or equal(pg. 42). |
| >>(pg. **Error! Bookmark not defined.**). | Bit right-shift operator(pg. 43). |
| >>=(pg. **Error! Bookmark not defined.**). | Bit right-shift assignment operator(pg. 43). |
| \|(pg. **Error! Bookmark not defined.**). | Logical or operator(pg. **Error! Bookmark not defined.**). |
| \|=(pg. **Error! Bookmark not defined.**). | Logical or assignment operator(pg. **Error! Bookmark not defined.**). |
| \|\|(pg. **Error! Bookmark not defined.**). | Conditional or(pg. **Error! Bookmark not defined.**). |
| ^(pg. **Error! Bookmark not defined.**). | Exclusive or(pg. **Error! Bookmark not defined.**). |
| ^=(pg. **Error! Bookmark not defined.**). | Exclusive or assignment operator(pg. **Error! Bookmark not defined.**). |
| ~(pg. **Error! Bookmark not defined.**). | Bit-wise not operator(pg. **Error! Bookmark not defined.**). |
| above(pg. **Error! Bookmark not defined.**). | Greater than comparison(pg. 44). |
| above_eq(pg. **Error! Bookmark not defined.**). | Greater than or equal(pg. 44). |
| abs(pg. **Error! Bookmark not defined.**). | Absolute value of expression(pg. 45). |
| below(pg. **Error! Bookmark not defined.**). | Less than operator(pg. 51). |
| below_eq(pg. **Error! Bookmark not defined.**). | Less than or equal operator(pg. 51). |
| max(pg. **Error! Bookmark not defined.**). | Get maximum value(pg. 143). |
| min(pg. **Error! Bookmark not defined.**). | Get minimum value(pg. 144). |
| rand(pg. **Error! Bookmark not defined.**). | Get a random number(pg. 158). |
| srand(pg. **Error! Bookmark not defined.**). | Set random number seed(pg. 200). |

## Bookmarks

These functions allow you to access the bookmarks in CRiSP.

| | |
|---|---|
| bookmark_list(pg. **Error! Bookmark not defined.**). | Get list of defined bookmarks(pg. 52). |
| delete_bookmark(pg. **Error! Bookmark not defined.**). | Delete a bookmark(pg. 69). |
| drop_bookmark(pg. **Error! Bookmark not defined.**). | Create a bookmark(pg. 79). |

goto_bookmark(pg. **Error! Bookmark not defined.**). Jump to a bookmark(pg. 100).

## Buffers

These functions allow you to operate on buffers.

acknowledge_changed_buffer(pg. **Error! Bookmark not defined.**). Turn off changed file status trigger(pg. 45).
attach_buffer(pg. **Error! Bookmark not defined.**). Attach a buffer to window(pg. 49).
beautify_buffer(pg. **Error! Bookmark not defined.**). Reformat a buffer(pg. 51).
clear_buffer(pg. **Error! Bookmark not defined.**). Delete all text from current buffer(pg. 56).
compare_files(pg. **Error! Bookmark not defined.**). Determine if two files are identical.(pg. 59).
connect(pg. **Error! Bookmark not defined.**). Attach buffer to an external process(pg. 59).
create_buffer(pg. **Error! Bookmark not defined.**). Create a new buffer(pg. 62).
create_blob(pg. **Error! Bookmark not defined.**). Create an arrow marker.(pg. 61).
create_edge(pg. **Error! Bookmark not defined.**). Split character mode window(pg. 63).
create_nested_buffer(pg. **Error! Bookmark not defined.**). Create a new buffer or increment reference count(pg. 64).
delete_blob(pg. **Error! Bookmark not defined.**). Delete arrow marker.(pg. 69).
delete_buffer(pg. **Error! Bookmark not defined.**). Delete a buffer(pg. 69).
diff_buffers(pg. **Error! Bookmark not defined.**). Compare two buffers and mark different lines(pg. 74).
distance_to_tab(pg. **Error! Bookmark not defined.**). Calculate distance to next tab(pg. 77).
disconnect(pg. **Error! Bookmark not defined.**). Disconnect a buffer from an external process(pg. 75).
end_of_buffer(pg. **Error! Bookmark not defined.**). Go to end of buffer(pg. 83).
end_of_window(pg. **Error! Bookmark not defined.**). Move cursor to bottom of window(pg. 83).
filename_extension(pg. **Error! Bookmark not defined.**). Extract file extension from a string(pg. 86).
find_line_flags(pg. **Error! Bookmark not defined.**). Find lines marked with specific flag attributes(pg. 90).
hide_ctl(pg. **Error! Bookmark not defined.**). Functions which affect outline mode.(pg. 101).
inherit_buffer_attributes(pg. **Error! Bookmark not defined.**). Inherit buffer attributes from a model buffer(pg. 103).
inq_buffer(pg. **Error! Bookmark not defined.**). Get current buffer id(pg. 105).
inq_buffer_flags(pg. **Error! Bookmark not defined.**). Retrieve buffer attributes(pg. 105).
inq_buffer_title(pg. **Error! Bookmark not defined.**). Get title at top of a window(pg. 107).
inq_byte_pos(pg. **Error! Bookmark not defined.**). Return cursor position as a byte offset(pg. 107).
inq_changed_buffers(pg. **Error! Bookmark not defined.**). Retrieve reason for a file state change(pg. 108).
inq_filesize(pg. **Error! Bookmark not defined.**). Estimate size of buffer when written to disk.(pg. 109).
inq_index(pg. **Error! Bookmark not defined.**). Return current character position on line. (pg. 111).
inq_line_flags(pg. **Error! Bookmark not defined.**). Retrieve line flags.(pg. 113).
inq_line_length(pg. **Error! Bookmark not defined.**). Get length of longest line(pg. 113).
inq_lines(pg. **Error! Bookmark not defined.**). Get count/percentage of lines in buffer(pg. 114).
inq_modified(pg. **Error! Bookmark not defined.**). Determine if current buffer has been modified(pg. 117).
inq_names(pg. **Error! Bookmark not defined.**). Get buffer and file names(pg. 118).
inq_offset(pg. **Error! Bookmark not defined.**). Get current byte offset of cursor(pg. 119).
inq_position(pg. **Error! Bookmark not defined.**). Get current buffer position(pg. 119).
inq_process_position(pg. **Error! Bookmark not defined.**). Get position of process buffer(pg. 120).
inq_record_size(pg. **Error! Bookmark not defined.**). Retrieve buffer record size attribute(pg. 121).
inq_ruler(pg. **Error! Bookmark not defined.**). Get current ruler marker settings(pg. 121).
inq_system(pg. **Error! Bookmark not defined.**). Determine if buffer is a system buffer or not(pg. 122).
inq_time(pg. **Error! Bookmark not defined.**). Determine when buffer was last modified(pg. 122).
inq_top_left(pg. **Error! Bookmark not defined.**). Get co-ordinates of top left hand side of window(pg. 122).
inq_views(pg. **Error! Bookmark not defined.**). Count how many windows display a buffer(pg. 123).
insert(pg. **Error! Bookmark not defined.**). Insert string into current buffer(pg. 125).
insert_buffer(pg. **Error! Bookmark not defined.**). Insert string into specified buffer(pg. 126).
insert_process(pg. **Error! Bookmark not defined.**). Send string to a process attached to a buffer(pg. 127).
line_add(pg. **Error! Bookmark not defined.**). Arithmetic on line numbers when outlining enabled(pg. 136).

| | |
|---|---|
| list_of_buffers(pg. **Error! Bookmark not defined.**). | List of defined buffers(pg. 137). |
| make_local_variable(pg. **Error! Bookmark not defined.**). | Make a variable a buffer local variable(pg. 142). |
| next_buffer(pg. **Error! Bookmark not defined.**). | Switch to next buffer(pg. 147). |
| read(pg. **Error! Bookmark not defined.**). | Read characters from the buffer(pg. 160). |
| reload_buffer(pg. **Error! Bookmark not defined.**). | Reload buffer from file(pg. 167). |
| restore_excursion(pg. **Error! Bookmark not defined.**). | Restore position(pg. 169). |
| restore_position(pg. **Error! Bookmark not defined.**). | Restore buffer position(pg. 170). |
| set_buffer(pg. **Error! Bookmark not defined.**). | Set the current buffer(pg. 176). |
| set_buffer_flags(pg. **Error! Bookmark not defined.**). | Set buffer attributes(pg. 176). |
| set_buffer_title(pg. **Error! Bookmark not defined.**). | Set title at top of a window(pg. 178). |
| set_line_flags(pg. **Error! Bookmark not defined.**). | Set flags for a block of lines.(pg. 183). |
| set_model_buffer(pg. **Error! Bookmark not defined.**). | Mark a buffer so that its attribuets can be inherited.(pg. 183). |
| set_process_position(pg. **Error! Bookmark not defined.**). | Set insert position for a process buffer(pg. 184). |
| set_record_size(pg. **Error! Bookmark not defined.**). | Set input file fixed length line size(pg. 184). |
| set_ruler(pg. **Error! Bookmark not defined.**). | Set current ruler markers(pg. 185). |
| sort_buffer(pg. **Error! Bookmark not defined.**). | Sort buffer or selected region(pg. 196). |
| tabs(pg. **Error! Bookmark not defined.**). | Set tab stops for current buffer(pg. 205). |
| top_of_buffer(pg. **Error! Bookmark not defined.**). | Move cursor to start of current buffer(pg. 208). |
| transfer(pg. **Error! Bookmark not defined.**). | Move text from one buffer to another(pg. 208). |
| write_block(pg. **Error! Bookmark not defined.**). | Write selected area to a file(pg. 216). |
| write_buffer(pg. **Error! Bookmark not defined.**). | Save buffer contents to disk(pg. 216). |
| write(pg. **Error! Bookmark not defined.**). | Write text to a file.(pg. 215). |

## Character Maps

Character maps are a way of translating displayed characters in a window. They are used, for example, to provide the facility for displaying all characters as printable characters, or hexadecimal characters.

| | |
|---|---|
| create_char_map(pg. **Error! Bookmark not defined.**). | Create a mapping for displayed characters(pg. 62). |
| inq_char_map(pg. **Error! Bookmark not defined.**). | Get id of current character display map(pg. 108). |
| set_buffer_cmap(pg. **Error! Bookmark not defined.**). | Set character mode mapping for displayed characters(pg. 176). |

## Character mode terminal functions

Most of these functions are relevant to the character mode version of CRiSP and provide the facility to define attributes of the terminal you are using. These are provided because of the limitations of the termcap facillity on most Unix systems.

The redraw(pg. 162). and refresh(pg. 162). functions are relevant in the GUI environment as well.

| | |
|---|---|
| get_term_characters(pg. **Error! Bookmark not defined.**). | Get value of character mode terminal character mappings(pg. 97). |
| get_term_features(pg. **Error! Bookmark not defined.**). | Get value of character mode terminal features(pg. 97). |
| get_term_keyboard(pg. **Error! Bookmark not defined.**). | Get character mode function key escape sequences(pg. 97). |
| redraw(pg. **Error! Bookmark not defined.**). | Redraw screen(pg. 162). |
| refresh(pg. **Error! Bookmark not defined.**). | Update the display(pg. 162). |
| screen_dump(pg. **Error! Bookmark not defined.**). | Dump character mode screen to a file(pg. 172). |
| set_char_timeout(pg. **Error! Bookmark not defined.**). | Set inter-character timeout delay.(pg. 179). |
| set_term_characters(pg. **Error! Bookmark not defined.**). | Set character mode window drawing characters(pg. 186). |
| set_term_features(pg. **Error! Bookmark not defined.**). | Define terminal emulation facilities(pg. 187). |
| set_term_keyboard(pg. **Error! Bookmark not defined.**). | Define character mode keyboard definitions(pg. 188). |

## Color & colorizer macros

These macros provide the mechanism for configuring CRiSPs colors and accessing the colorizer mechanism.

| | |
|---|---|
| color(pg. **Error! Bookmark not defined.**). | Set screen color scheme(pg. 57). |
| colorizer(pg. **Error! Bookmark not defined.**). | set/get language specific coloring attributes(pg. 58). |
| colorizer_get_info(pg. **Error! Bookmark not defined.**). | Get parse state of a position in a buffer.(pg. 58). |
| cvt_hsv_to_rgb(pg. **Error! Bookmark not defined.**). | Convert HSV color value to RGB(pg. 66). |
| cvt_rgb_to_hsv(pg. **Error! Bookmark not defined.**). | Convert RGB color value to HSV format(pg. 67). |
| get_color(pg. **Error! Bookmark not defined.**). | Get current color mappings(pg. 93). |
| load_keywords(pg. **Error! Bookmark not defined.**). | Load a colorizer description from a .KWD file(pg. 139). |
| set_color(pg. **Error! Bookmark not defined.**). | Set colors(pg. 179). |
| set_keyword_index(pg. **Error! Bookmark not defined.**). | Set name of colorizer for a buffer(pg. 182). |

## Cursor Movement

These functions provide basic cursor movement within a buffer.

| | |
|---|---|
| beginning_of_line(pg. **Error! Bookmark not defined.**). | Move cursor to start of line(pg. 51). |
| down(pg. **Error! Bookmark not defined.**). | Move cursor down(pg. 78). |
| end_of_line(pg. **Error! Bookmark not defined.**). | Go to end of line(pg. 83). |
| goto_line(pg. **Error! Bookmark not defined.**). | Goto a particular line(pg. 100). |
| goto_old_line(pg. **Error! Bookmark not defined.**). | Goto a specific line before file was modified(pg. 100). |
| left(pg. **Error! Bookmark not defined.**). | Move cursor left(pg. 136). |
| move_abs(pg. **Error! Bookmark not defined.**). | Move to an absolute line and column number(pg. 147). |
| move_rel(pg. **Error! Bookmark not defined.**). | Move relative to current position(pg. 147). |
| next_char(pg. **Error! Bookmark not defined.**). | Move to next character(pg. 148). |
| page_down(pg. **Error! Bookmark not defined.**). | Move cursor down a page(pg. 150). |
| page_up(pg. **Error! Bookmark not defined.**). | Move cursor up a page(pg. 150). |
| prev_char(pg. **Error! Bookmark not defined.**). | Move back one character(pg. 152). |
| right(pg. **Error! Bookmark not defined.**). | Move cursor to the right(pg. 170). |
| save_excursion(pg. **Error! Bookmark not defined.**). | Save current cursor position(pg. 171). |
| save_position(pg. **Error! Bookmark not defined.**). | Save current cursor position(pg. 172). |
| up(pg. **Error! Bookmark not defined.**). | Move cursor up one line(pg. 212). |

## Deleting Text

These functions provide the ability to delete text in a buffer.

| | |
|---|---|
| delete_block(pg. **Error! Bookmark not defined.**). | Deleted selected region(pg. 69). |
| delete_char(pg. **Error! Bookmark not defined.**). | Delete character(pg. 70). |
| delete_line(pg. **Error! Bookmark not defined.**). | Delete current line(pg. 71). |
| delete_to_eol(pg. **Error! Bookmark not defined.**). | Delete text to end of line(pg. 72). |

## Dictionary Functions

The dictionary facility is a way of creating user defined symbol tables which are most commonly used to associate state information with dialog boxes.

| | |
|---|---|
| create_dictionary(pg. **Error! Bookmark not defined.**). | Create user defined symbol table(pg. 63). |
| delete_dictionary(pg. **Error! Bookmark not defined.**). | Free up a private symbol table(pg. 70). |
| dict_delete(pg. **Error! Bookmark not defined.**). | Delete an entry in a dictionary(pg. 73). |
| dict_exists(pg. **Error! Bookmark not defined.**). | Check for entry in a dictionary(pg. 73). |
| dict_list(pg. **Error! Bookmark not defined.**). | Return list of all entries in a dictionary(pg. 74). |
| get_property(pg. **Error! Bookmark not defined.**). | Retrieve a private value from a GUI dialog |

box object(pg. 95).

| | |
|---|---|
| list_of_dictionaries(pg. **Error! Bookmark not defined.**). | Returns list of all defined dictionaries.(pg. 137). |
| remove_property(pg. **Error! Bookmark not defined.**). | Delete an entry from a dictionary(pg. 168). |
| set_property(pg. **Error! Bookmark not defined.**). | Change value of a user-defined property associated with a GUI object(pg. 185). |

## Displaying messages

These functions are used to display information whilst a macro is running. The printf(pg. 153). function is mostly useful as a debugging aid.

| | |
|---|---|
| beep(pg. **Error! Bookmark not defined.**). | Issue a sound(pg. 51). |
| error(pg. **Error! Bookmark not defined.**). | Issue an error message on the command line(pg. 83). |
| message(pg. **Error! Bookmark not defined.**). | Display a message on the command line(pg. 143). |
| printf(pg. **Error! Bookmark not defined.**). | Print formatted string to stdout(pg. 153). |

## Editing functions

These functions provide the basic editing facilities available in CRiSP.

| | |
|---|---|
| backspace(pg. **Error! Bookmark not defined.**). | Delete character to the left of the cursor(pg. 50). |
| detab_text(pg. **Error! Bookmark not defined.**). | Convert embedded TABs to spaces(pg. 72). |
| find_marker(pg. **Error! Bookmark not defined.**). | Find find marked lines(pg. 91). |
| inq_mode(pg. **Error! Bookmark not defined.**). | Get value of INS or OVR mode(pg. 117). |
| insert_mode(pg. **Error! Bookmark not defined.**). | Change INS/OVR mode(pg. 126). |
| mark_line(pg. **Error! Bookmark not defined.**). | Mark a line(pg. 142). |
| self_insert(pg. **Error! Bookmark not defined.**). | Insert a single character as if it were typed(pg. 174). |
| use_tab_char(pg. **Error! Bookmark not defined.**). | Set use of hard/soft tabs(pg. 213). |
| undo(pg. **Error! Bookmark not defined.**). | Undo last edit operation(pg. 210). |

## Environment functions

These functions allow you to access environment variables.

| | |
|---|---|
| getenv(pg. **Error! Bookmark not defined.**). | Return value of an environment variable(pg. 97). |
| inq_brief_level(pg. **Error! Bookmark not defined.**). | Get value of nested CRiSP invocations(pg. 104). |
| inq_environment(pg. **Error! Bookmark not defined.**). | Get environment variable(pg. 109). |
| putenv(pg. **Error! Bookmark not defined.**). | Set environment variable(pg. 156). |

## File & Directory functions

These functions provide a variety of mechanisms for manipulating files and directories.

| | |
|---|---|
| access(pg. **Error! Bookmark not defined.**). | Test file accessability(pg. 45). |
| cd(pg. **Error! Bookmark not defined.**). | Change working directory(pg. 53). |
| chmod(pg. **Error! Bookmark not defined.**). | Change file protection(pg. 55). |
| chown(pg. **Error! Bookmark not defined.**). | Change ownership/group permissions of a file(pg. 55). |
| close(pg. **Error! Bookmark not defined.**). | Close a file descriptor(pg. 57). |
| del(pg. **Error! Bookmark not defined.**). | Delete file(pg. 69). |
| edit_file(pg. **Error! Bookmark not defined.**). | Edit a file(pg. 81). |
| exist(pg. **Error! Bookmark not defined.**). | Check if a file exists(pg. 85). |
| getwd(pg. **Error! Bookmark not defined.**). | Get current working directory(pg. 99). |
| file_glob(pg. **Error! Bookmark not defined.**). | Expand wildcard string into names of files(pg. 86). |
| file_pattern(pg. **Error! Bookmark not defined.**). | Set up file regular expression for reading directories(pg. 87). |
| filename_append(pg. **Error! Bookmark not defined.**). | Create a new filename by appending a filename to a directory name(pg. 87). |
| filename_delimiter(pg. **Error! Bookmark not defined.**). | Return a string which can be used to delimit directory and filenames(pg. 88). |

| | |
|---|---|
| filename_first_component(pg. **Error! Bookmark not defined.**). | Return first part of a filename(pg. 88). |
| filename_has_dirspec(pg. **Error! Bookmark not defined.**). | Indicate whether a filename contains a directory specification.(pg. 88). |
| filename_is_absolute(pg. **Error! Bookmark not defined.**). | Indicate whether a filename is an absolute or relative filename.(pg. 88). |
| filename_last_component(pg. **Error! Bookmark not defined.**). | Return last part of a filename(pg. 89). |
| filename_match(pg. **Error! Bookmark not defined.**). | Compare a filename against a file expression(pg. 89). |
| find_distribution_file(pg. **Error! Bookmark not defined.**). | Find files which are a part of the CRiSP package(pg. 89). |
| find_file(pg. **Error! Bookmark not defined.**). | Read next directory entry.(pg. 90). |
| find_path(pg. **Error! Bookmark not defined.**). | Find a file in a list of directories(pg. 91). |
| inq_extension(pg. **Error! Bookmark not defined.**). | Get real or imaginary file extension.(pg. 109). |
| inq_filesystem(pg. **Error! Bookmark not defined.**). | Get operating system and file-system attributes(pg. 110). |
| is_directory(pg. **Error! Bookmark not defined.**). | Test to see if filename is a directory.(pg. 132). |
| is_pipe_filename(pg. **Error! Bookmark not defined.**). | Determine if file name refers to a pipeline(pg. 133). |
| is_url_filename(pg. **Error! Bookmark not defined.**). | Determine if a filename is a URL reference (e.g. HTTP or FTP based).(pg. 133). |
| mkdir(pg. **Error! Bookmark not defined.**). | Create a directory(pg. 145). |
| mkfifo(pg. **Error! Bookmark not defined.**). | Create a named pipe(pg. 145). |
| mktemp(pg. **Error! Bookmark not defined.**). | Generate a temporary filename(pg. 145). |
| mode_string(pg. **Error! Bookmark not defined.**). | Convert file permission bits to a string.(pg. 145). |
| open(pg. **Error! Bookmark not defined.**). | Open an external file(pg. 149). |
| output_file(pg. **Error! Bookmark not defined.**). | Set name of output file for current buffer(pg. 150). |
| read_file(pg. **Error! Bookmark not defined.**). | Insert file into buffer(pg. 162). |
| read_from_file(pg. **Error! Bookmark not defined.**). | Read from an open file(pg. 160). |
| readlink(pg. **Error! Bookmark not defined.**). | Read value of a symbolic link(pg. 161). |
| remove(pg. **Error! Bookmark not defined.**). | Delete a file(pg. 168). |
| rename(pg. **Error! Bookmark not defined.**). | Rename a file(pg. 168). |
| rmdir(pg. **Error! Bookmark not defined.**). | Remove a directory.(pg. 171). |
| set_backup(pg. **Error! Bookmark not defined.**). | Set backup creation mode(pg. 175). |
| simplify_filename(pg. **Error! Bookmark not defined.**). | Remove redundant directory names from a filename(pg. 196). |
| split_url(pg. **Error! Bookmark not defined.**). | Split a string containing a URL into its components(pg. 199). |
| stat(pg. **Error! Bookmark not defined.**). | Get information on a file(pg. 201). |
| tempnam(pg. **Error! Bookmark not defined.**). | Generate a temporary filename(pg. 206). |
| tmpnam(pg. **Error! Bookmark not defined.**). | Generate a temporary filename(pg. 207). |
| umask(pg. **Error! Bookmark not defined.**). | Set file creation mask(pg. 210). |
| utime(pg. **Error! Bookmark not defined.**). | Set file modification times(pg. 213). |
| write(pg. **Error! Bookmark not defined.**). | Write to an open file(pg. 215). |

## General control functions

These functions are used to control various aspects of the CRiSP editor.

| | |
|---|---|
| display_mode(pg. **Error! Bookmark not defined.**). | Set miscellaneous character mode attributes(pg. 76). |
| edit_control(pg. **Error! Bookmark not defined.**). | Set/get various editing mode flags(pg. 79). |
| get_mem_info(pg. **Error! Bookmark not defined.**). | Get current VM parameters(pg. 94). |
| get_system_resources(pg. **Error! Bookmark not defined.**). | Machine dependent primitive to retrieve system values(pg. 97). |
| mem_info(pg. **Error! Bookmark not defined.**). | Get/set CRiSPs virtual memory size(pg. 143). |
| profile(pg. **Error! Bookmark not defined.**). | Reserved(pg. 155). |
| set_application_name(pg. **Error! Bookmark not defined.**). | Set name of application.(pg. 175). |
| set_application_icon(pg. **Error! Bookmark not defined.**). | Set application icon.(pg. 174). |
| set_crisp_flags(pg. **Error! Bookmark not defined.**). | Set various internal CRiSP flags(pg. 180). |
| uname(pg. **Error! Bookmark not defined.**). | Get operating system version information.(pg. 210). |

version(pg. **Error! Bookmark not defined.**). Retrieve CRiSP version information(pg. 213).

## GUI Programming

These functions are used to program the GUI dialog boxes and other aspects of the GUI environment.

change_object(pg. **Error! Bookmark not defined.**). Set dialog box object attributes(pg. 54).
clipboard_available(pg. **Error! Bookmark not defined.**). Determine if data is in the clipboard for retrieval,(pg. 56).
clipboard_owner(pg. **Error! Bookmark not defined.**). Determine if CRiSP owns the clipboard(pg. 56).
create_notice(pg. **Error! Bookmark not defined.**). Modal information dialog box(pg. 64).
create_object(pg. **Error! Bookmark not defined.**). Create GUI dialog box(pg. 64).
delete_object(pg. **Error! Bookmark not defined.**). Destroy a dialog box(pg. 71).
dialog_box(pg. **Error! Bookmark not defined.**). Create a user-defined dialog box(pg. 72).
font_ctl(pg. **Error! Bookmark not defined.**). Get information about available fonts.(pg. 92).
get_mouse_pos(pg. **Error! Bookmark not defined.**). Extract (X,Y) position of last mouse action(pg. 94).
get_mouse_screen(pg. **Error! Bookmark not defined.**). Return screen id of last mouse action(pg. 94).
get_property(pg. **Error! Bookmark not defined.**). Retrieve a private value from a GUI dialog box object(pg. 95).
get_selection(pg. **Error! Bookmark not defined.**). Retrieve contents of GUI cut/paste selection(pg. 96).
inq_font(pg. **Error! Bookmark not defined.**). Get current editing or dialog default font(pg. 110).
inq_objects(pg. **Error! Bookmark not defined.**). Determine if a dialog box exists(pg. 118).
inq_screen(pg. **Error! Bookmark not defined.**). Get id of the current screen(pg. 121).
inq_screen_flags(pg. **Error! Bookmark not defined.**). Get flags associated with the current screen(pg. 121).
inq_screen_size(pg. **Error! Bookmark not defined.**). Get size of current screen(pg. 122).
insert_object(pg. **Error! Bookmark not defined.**). Insert a new widget hierarchy into a dialog box(pg. 127).
list_of_bitmaps(pg. **Error! Bookmark not defined.**). List pixmaps in .xpl library(pg. 137).
list_of_objects(pg. **Error! Bookmark not defined.**). List of sub-children objects(pg. 138).
object_value(pg. **Error! Bookmark not defined.**). Get GUI object specific attributes(pg. 149).
parent_of_object(pg. **Error! Bookmark not defined.**). Get object id of owning DBOX_SCREEN object(pg. 150).
popup_object(pg. **Error! Bookmark not defined.**). Popup a dialog box(pg. 152).
process_mouse(pg. **Error! Bookmark not defined.**). Get mouse information(pg. 154).
put_selection(pg. **Error! Bookmark not defined.**). Make text available for the clipboard(pg. 156).
remove_object(pg. **Error! Bookmark not defined.**). Delete an object from a dialog box(pg. 168).
set_busy_state(pg. **Error! Bookmark not defined.**). Change cursor style for a dialog box(pg. 178).
set_cursor_type(pg. **Error! Bookmark not defined.**). Set stype of cursor(pg. 180).
set_editable_screen(pg. **Error! Bookmark not defined.**). Change to the last edit screen.(pg. 181).
set_font(pg. **Error! Bookmark not defined.**). Get/set editing window or dialog box fonts(pg. 181).
set_property(pg. **Error! Bookmark not defined.**). Change value of a user-defined property associated with a GUI object(pg. 185).
set_screen(pg. **Error! Bookmark not defined.**). Change character mode screen(pg. 185).
set_screen_size(pg. **Error! Bookmark not defined.**). Set GUI window size(pg. 186).
set_wm_name(pg. **Error! Bookmark not defined.**). Set GUI mode window and icon title(pg. 192).
translate_pos(pg. **Error! Bookmark not defined.**). Convert X,Y position to window co-ordinates(pg. 209).
wait_for_object(pg. **Error! Bookmark not defined.**). Wait for a dialog box to terminate(pg. 214).

## Interprocess communication & Sub-processes

These functions provide inter-process communications facilities and access to sub-processes.

crisp_instance(pg. **Error! Bookmark not defined.**). Communicate with another copy of

| | |
|---|---|
| | CRiSP(pg. 65). |
| dos(pg. **Error! Bookmark not defined.**). | Create a sub-shell(pg. 77). |
| ipc_accept(pg. **Error! Bookmark not defined.**). | Accept a client TCP connection(pg. 128). |
| ipc_close(pg. **Error! Bookmark not defined.**). | Close an IPC channel(pg. 128). |
| ipc_create(pg. **Error! Bookmark not defined.**). | Create an external IPC connection(pg. 128). |
| ipc_read(pg. **Error! Bookmark not defined.**). | Read data from an IPC connection(pg. 130). |
| ipc_register(pg. **Error! Bookmark not defined.**). | Register a callback for an IPC connection(pg. 130). |
| ipc_status(pg. **Error! Bookmark not defined.**). | Return status of IPC channel(pg. 131). |
| ipc_unregister(pg. **Error! Bookmark not defined.**). | Remove a callback for an IPC connection(pg. 131). |
| ipc_write(pg. **Error! Bookmark not defined.**). | Write data to an IPC connection(pg. 132). |
| kill(pg. **Error! Bookmark not defined.**). | Send signal to a process(pg. 136). |
| list_of_screens(pg. **Error! Bookmark not defined.**). | return list of displayed screens (frames)(pg. 138). |
| send_signal(pg. **Error! Bookmark not defined.**). | Send signal to a process buffer(pg. 174). |
| shell(pg. **Error! Bookmark not defined.**). | Spawn a sub-shell process to execute a command(pg. 194). |
| shell_execute(pg. **Error! Bookmark not defined.**). | Execute command or file based on file type.(pg. 195). |
| suspend(pg. **Error! Bookmark not defined.**). | Same as ^Z(pg. 203). |
| wait(pg. **Error! Bookmark not defined.**). | Wait for a attached process to terminate(pg. 213). |
| wait_for(pg. **Error! Bookmark not defined.**). | Wait for output from an attached process(pg. 214). |

## Keyboard functions

These functions provide ways to manipulate the keyboard bindings and perform various key code conversions.

| | |
|---|---|
| assign_to_key(pg. **Error! Bookmark not defined.**). | Assign a key binding(pg. 47). |
| assign_to_raw_key(pg. **Error! Bookmark not defined.**). | Assign a macro to a raw key binding(pg. 48). |
| copy_keyboard(pg. **Error! Bookmark not defined.**). | Copy a set of keyboard bindings(pg. 61). |
| delete_keystroke_macro(pg. **Error! Bookmark not defined.**). | Delete keystroke macro(pg. 71). |
| enable_interrupt(pg. **Error! Bookmark not defined.**). | Enable keyboard interrupts during certain macro operations(pg. 82). |
| execute_keystroke(pg. **Error! Bookmark not defined.**). | Execute macro associated with a keystroke.(pg. 84). |
| input_mode(pg. **Error! Bookmark not defined.**). | Get/set effect of certain system keys(pg. 103). |
| inq_assignment(pg. **Error! Bookmark not defined.**). | Get keyboard key assignment(pg. 104). |
| inq_kbd_char(pg. **Error! Bookmark not defined.**). | Get a character from the keyboard or return immediately(pg. 111). |
| inq_kbd_flags(pg. **Error! Bookmark not defined.**). | Get current state of keyboard modifiers.(pg. 111). |
| inq_kbd_name(pg. **Error! Bookmark not defined.**). | Get name of current keyboard bindings(pg. 112). |
| inq_keyboard(pg. **Error! Bookmark not defined.**). | Get current keyboard binding id(pg. 112). |
| inq_keys_pressed(pg. **Error! Bookmark not defined.**). | Get count of keys pressed so far(pg. 112). |
| inq_keystroke_macro(pg. **Error! Bookmark not defined.**). | Get id of current keystroke macro(pg. 113). |
| inq_local_keyboard(pg. **Error! Bookmark not defined.**). | Get keyboard id for the private buffer mappings(pg. 114). |
| inq_playback(pg. **Error! Bookmark not defined.**). | Determine if a macro is being recorded.(pg. 119). |
| inq_raw_kbd_char(pg. **Error! Bookmark not defined.**). | Get a raw scancode from the keyboard or return immediately(pg. 120). |
| int_to_key(pg. **Error! Bookmark not defined.**). | Convert a key code to a key name(pg. 128). |
| key_list(pg. **Error! Bookmark not defined.**). | Get keyboard bindings(pg. 133). |
| key_to_int(pg. **Error! Bookmark not defined.**). | Convert key name to a key code(pg. 134). |
| keyboard_flush(pg. **Error! Bookmark not defined.**). | Flush typeahead buffer(pg. 134). |
| keyboard_pop(pg. **Error! Bookmark not defined.**). | Pop a keyboard mapping(pg. 134). |
| keyboard_push(pg. **Error! Bookmark not defined.**). | Save a keyboard binding(pg. 135). |
| keyboard_reset(pg. **Error! Bookmark not defined.**). | Clear out all key assignments(pg. 135). |
| keyboard_typeables(pg. **Error! Bookmark not defined.**). | Assign key-bindings to all normal typeable |

keyboard characters(pg. 135).

| | |
|---|---|
| list_of_keyboards(pg. **Error! Bookmark not defined.**). | Get list of currently defined keyboard bindings(pg. 138). |
| list_of_keystroke_macros(pg. **Error! Bookmark not defined.**). | List of keystroke macros(pg. 138). |
| load_keyboard_file(pg. **Error! Bookmark not defined.**). | Load a keyboard bindings file(pg. 139). |
| load_keystroke_macro(pg. **Error! Bookmark not defined.**). | Load a recorded macro from a file.(pg. 139). |
| pause(pg. **Error! Bookmark not defined.**). | Pause keystroke definition(pg. 151). |
| playback(pg. **Error! Bookmark not defined.**). | Playback a keystroke macro(pg. 151). |
| push_back(pg. **Error! Bookmark not defined.**). | Put key code back in the input stream(pg. 155). |
| read_char(pg. **Error! Bookmark not defined.**). | Wait for a character from the keyboard(pg. 161). |
| remember(pg. **Error! Bookmark not defined.**). | Start recording keystrokes(pg. 167). |
| set_kbd_name(pg. **Error! Bookmark not defined.**). | Set name of current keyboard bindings(pg. 182). |
| use_local_keyboard(pg. **Error! Bookmark not defined.**). | Attach a local keyboard to a buffer(pg. 212). |

## List manipulation

These functions manipulate the **list** data type.

| | |
|---|---|
| car(pg. **Error! Bookmark not defined.**). | Get first element from a list(pg. 53). |
| cdr(pg. **Error! Bookmark not defined.**). | Remove first element from a list(pg. 54). |
| delete_nth(pg. **Error! Bookmark not defined.**). | Remove one or more elements from a list(pg. 71). |
| length_of_list(pg. **Error! Bookmark not defined.**). | Count number of items in list(pg. 136). |
| list_extract(pg. **Error! Bookmark not defined.**). | Extract alternate entries from a list.(pg. 137). |
| list_to_string(pg. **Error! Bookmark not defined.**). | Format items in a list(pg. 139). |
| make_list(pg. **Error! Bookmark not defined.**). | Create a list(pg. 141). |
| nth(pg. **Error! Bookmark not defined.**). | Extract n'th item from a list(pg. 149). |
| put_nth(pg. **Error! Bookmark not defined.**). | Modify element in a list(pg. 156). |
| quote_list(pg. **Error! Bookmark not defined.**). | Create an unevaluated list(pg. 157). |
| sort_list(pg. **Error! Bookmark not defined.**). | Sort a list(pg. 198). |
| split(pg. **Error! Bookmark not defined.**). | Split a string into tokens(pg. 199). |
| uniq_list(pg. **Error! Bookmark not defined.**). | Remove duplicate strings from a list.(pg. 211). |

## Macro Programming

These functions are used in writing macros.

| | |
|---|---|
| abort(pg. **Error! Bookmark not defined.**). | Abort CRiSP execution immediately(pg. 44). |
| arg_list(pg. **Error! Bookmark not defined.**). | Retrieve list of arguments to current macro(pg. 46). |
| autoload(pg. **Error! Bookmark not defined.**). | specify where macros are located and search path for macros(pg. 49). |
| break(pg. **Error! Bookmark not defined.**). | Break out of a loop or switch statement(pg. 52). |
| breaksw(pg. **Error! Bookmark not defined.**). | Internal function implementing the break statement(pg. 52). |
| command_list(pg. **Error! Bookmark not defined.**). | Get list of all macros and builtin functions(pg. 58). |
| continue(pg. **Error! Bookmark not defined.**). | Continue execution of a loop(pg. 60). |
| debug(pg. **Error! Bookmark not defined.**). | Turn on/off CRiSP debugging and tracing(pg. 67). |
| declare(pg. **Error! Bookmark not defined.**). | Define polymorphic variables(pg. 68). |
| delete_macro(pg. **Error! Bookmark not defined.**). | Remove a macro from memory(pg. 71). |
| dlerror(pg. **Error! Bookmark not defined.**). | Return text of last load_library() failure(pg. 77). |
| do(pg. **Error! Bookmark not defined.**). | Do loop construct(pg. 77). |
| evaluate(pg. **Error! Bookmark not defined.**). | evaluate an expression contained in a string(pg. 84). |
| execute_macro(pg. **Error! Bookmark not defined.**). | Execute a macro(pg. 84). |
| exit(pg. **Error! Bookmark not defined.**). | Exit from nested macro level(pg. 85). |
| extern(pg. **Error! Bookmark not defined.**). | Declare a variable as being defined in another macro file(pg. 86). |
| find_macro(pg. **Error! Bookmark not defined.**). | Find location of a macro file(pg. 91). |
| first_time(pg. **Error! Bookmark not defined.**). | Determine if this is the first time a macro was invoked(pg. 92). |
| float(pg. **Error! Bookmark not defined.**). | Declare a floating point variable(pg. 92). |

| | |
|---|---|
| for(pg. **Error! Bookmark not defined.**). | For loop construct(pg. 93). |
| get_exception_info(pg. **Error! Bookmark not defined.**). | Get run-time information about a run-time error(pg. 94). |
| get_parm(pg. **Error! Bookmark not defined.**). | Retrieve value of a passed argument.(pg. 95). |
| global(pg. **Error! Bookmark not defined.**). | Define a variable as a global variable(pg. 99). |
| if(pg. **Error! Bookmark not defined.**). | If statement(pg. 102). |
| inq_called(pg. **Error! Bookmark not defined.**). | Get name of calling macro(pg. 107). |
| inq_debug(pg. **Error! Bookmark not defined.**). | Inquire about current debug status(pg. 109). |
| inq_macro(pg. **Error! Bookmark not defined.**). | Determine whether a macro or primitive exists(pg. 115). |
| inq_macro_files(pg. **Error! Bookmark not defined.**). | Get list of loaded macro files(pg. 115). |
| inq_module(pg. **Error! Bookmark not defined.**). | Return current module name.(pg. 117). |
| inq_msg_level(pg. **Error! Bookmark not defined.**). | Get current message level(pg. 118). |
| inq_process_level(pg. **Error! Bookmark not defined.**). | Get current interactive macro nesting level(pg. 120). |
| int(pg. **Error! Bookmark not defined.**). | Define an integer variable(pg. 127). |
| is_integer(pg. **Error! Bookmark not defined.**). | Check polymorphic variable to see if it contains an integer.(pg. 132). |
| is_list(pg. **Error! Bookmark not defined.**). | Determine if a variable contains a list(pg. 132). |
| is_null(pg. **Error! Bookmark not defined.**). | Determine if a variable is NULL(pg. 132). |
| is_string(pg. **Error! Bookmark not defined.**). | Determine if variable is a string(pg. 133). |
| list(pg. **Error! Bookmark not defined.**). | Declare list variables(pg. 137). |
| load_macro(pg. **Error! Bookmark not defined.**). | Load a macro file(pg. 140). |
| macro(pg. **Error! Bookmark not defined.**). | Define a macro(pg. 141). |
| macro_list(pg. **Error! Bookmark not defined.**). | Get list of macros(pg. 141). |
| module(pg. **Error! Bookmark not defined.**). | Define a module name for a set of macros(pg. 146). |
| nothing(pg. **Error! Bookmark not defined.**). | No-op(pg. 149). |
| pause_on_error(pg. **Error! Bookmark not defined.**). | Set pause mode when errors displayed on command line(pg. 151). |
| process(pg. **Error! Bookmark not defined.**). | Nest interactive macro execution(pg. 154). |
| put_parm(pg. **Error! Bookmark not defined.**). | Pass back values from a macro(pg. 156). |
| replacement(pg. **Error! Bookmark not defined.**). | Overload an existing macro definition.(pg. 168). |
| require(pg. **Error! Bookmark not defined.**). | Preload a macro file if not already loaded(pg. 169). |
| set_calling_name(pg. **Error! Bookmark not defined.**). | Set name of invoking macro(pg. 179). |
| set_msg_level(pg. **Error! Bookmark not defined.**). | Set level of informational messages(pg. 183). |
| static(pg. **Error! Bookmark not defined.**). | Define variables local to a macro file(pg. 201). |
| string(pg. **Error! Bookmark not defined.**). | Define string variables(pg. 201). |
| switch(pg. **Error! Bookmark not defined.**). | Switch statement(pg. 204). |
| return(pg. **Error! Bookmark not defined.**). | Return from a macro(pg. 170). |
| returns(pg. **Error! Bookmark not defined.**). | Return an expression from a macro(pg. 170). |
| typeof(pg. **Error! Bookmark not defined.**). | Get data type of an expression(pg. 210). |
| while(pg. **Error! Bookmark not defined.**). | Looping construct(pg. 215). |

## Mathematical functions

These functions provide floating point support for various standard mathematical functions.

| | |
|---|---|
| acos(pg. **Error! Bookmark not defined.**). | Inverse cos()(pg. 46). |
| asin(pg. **Error! Bookmark not defined.**). | Inverse sin() function(pg. 46). |
| atan(pg. **Error! Bookmark not defined.**). | Inverse tan() function(pg. 46). |
| atan2(pg. **Error! Bookmark not defined.**). | Inverse tan(y/x)(pg. 46). |
| ceil(pg. **Error! Bookmark not defined.**). | Get smallest integer not larger than expression(pg. 54). |
| cos(pg. **Error! Bookmark not defined.**). | Cosine function(pg. 61). |
| cosh(pg. **Error! Bookmark not defined.**). | Hyperbolic cosine function(pg. 61). |
| exp(pg. **Error! Bookmark not defined.**). | Return e^x(pg. 85). |
| fabs(pg. **Error! Bookmark not defined.**). | Get absolute value of a floating point value(pg. 86). |
| floor(pg. **Error! Bookmark not defined.**). | Get largest integer not greater than x(pg. 92). |
| fmod(pg. **Error! Bookmark not defined.**). | Floating point remainder operation(pg. 92). |
| frexp(pg. **Error! Bookmark not defined.**). | Convert number to a normalized floating point |

value(pg. 93).

|  |  |
|---|---|
| ldexp(pg. **Error! Bookmark not defined.**). | Evaluate x*2^exp(pg. 136). |
| log(pg. **Error! Bookmark not defined.**). | Return natural logarithm of argument(pg. 140). |
| log10(pg. **Error! Bookmark not defined.**). | Return base-10 logarithm of argument(pg. 140). |
| modf(pg. **Error! Bookmark not defined.**). | Get integer and fractional parts of a float(pg. 146). |
| pow(pg. **Error! Bookmark not defined.**). | x^y(pg. 152). |
| sin(pg. **Error! Bookmark not defined.**). | Sine function(pg. 196). |
| sinh(pg. **Error! Bookmark not defined.**). | Hyperbolic sine function(pg. 196). |
| sqrt(pg. **Error! Bookmark not defined.**). | Square-root function(pg. 200). |
| tan(pg. **Error! Bookmark not defined.**). | Tangent function(pg. 206). |
| tanh(pg. **Error! Bookmark not defined.**). | Hyperbolic tangent function(pg. 206). |

## Printing

These functions provide access to CRiSP's printing facility.

|  |  |
|---|---|
| print(pg. **Error! Bookmark not defined.**). | Unimplemented(pg. 153). |
| print_block(pg. **Error! Bookmark not defined.**). | Print a buffer or region of a buffer to a file in PostScript(pg. 153). |
| printer_dialog(pg. **Error! Bookmark not defined.**). | Native printing support.(pg. 153). |

## Registered macros and callbacks

These functions provide a variety of mechanisms for registering callback macros which are called when interesting events occur. Some of these are macros which CRiSP calls to provide assistance in the operation of certain events.

|  |  |
|---|---|
| _bad_key(pg. **Error! Bookmark not defined.**). | CRiSP callback for handling command line prompts(pg. 44). |
| _extension(pg. **Error! Bookmark not defined.**). | CRiSP callback to handle unknown filetypes(pg. 85). |
| _fatal_error(pg. **Error! Bookmark not defined.**). | CRiSP callback when internal error detected(pg. 86). |
| _prompt_begin(pg. **Error! Bookmark not defined.**). | CRiSP callback to implement command line(pg. 155). |
| _prompt_end(pg. **Error! Bookmark not defined.**). | CRiSP callback to implement command line(pg. 155). |
| _startup_complete(pg. **Error! Bookmark not defined.**). | Macro called on startup(pg. 200). |
| call_registered_macro(pg. **Error! Bookmark not defined.**). | Invoke registered macro callbacks(pg. 52). |
| register_file_class(pg. **Error! Bookmark not defined.**). | Specify wild cards for matching file types(pg. 162). |
| register_macro(pg. **Error! Bookmark not defined.**). | Register a callback procedure(pg. 163). |
| reregister_macro(pg. **Error! Bookmark not defined.**). | Reregister a callback procedure(pg. 163). |
| register_timer(pg. **Error! Bookmark not defined.**). | Create a timer(pg. 166). |
| unregister_macro(pg. **Error! Bookmark not defined.**). | De-install a callback macro(pg. 211). |
| unregister_timer(pg. **Error! Bookmark not defined.**). | Cancel a timer(pg. 211). |

## Searching

These functions perform search and replace operations.

|  |  |
|---|---|
| quote_regexp(pg. **Error! Bookmark not defined.**). | Make a string safe for regular expression searching(pg. 157). |
| re_search(pg. **Error! Bookmark not defined.**). | Search for a string(pg. 158). |
| re_syntax(pg. **Error! Bookmark not defined.**). | Set regular expression mode(pg. 159). |
| re_translate(pg. **Error! Bookmark not defined.**). | Search and replace(pg. 160). |
| search_back(pg. **Error! Bookmark not defined.**). | Backwards search(pg. 172). |
| search_case(pg. **Error! Bookmark not defined.**). | Set case matching mode in searches(pg. 172). |
| search_fwd(pg. **Error! Bookmark not defined.**). | Search forwards for a string(pg. 173). |
| search_list(pg. **Error! Bookmark not defined.**). | Search list for a value(pg. 173). |
| search_string(pg. **Error! Bookmark not defined.**). | Search string for a value(pg. 173). |
| translate(pg. **Error! Bookmark not defined.**). | Search and replace(pg. 208). |

## Selections and Regions

These functions provide access to the cut and paste mechanism and for highlighting parts of a buffer.

| | |
|---|---|
| cut(pg. **Error! Bookmark not defined.**). | Copy selection to scrap and delete selection(pg. 66). |
| copy(pg. **Error! Bookmark not defined.**). | Copy selected region to the scrap(pg. 61). |
| drop_anchor(pg. **Error! Bookmark not defined.**). | Start marking a selection(pg. 78). |
| end_anchor(pg. **Error! Bookmark not defined.**). | Set end of a selected region(pg. 82). |
| get_region(pg. **Error! Bookmark not defined.**). | Get hilight area as a string(pg. 96). |
| hilite_create(pg. **Error! Bookmark not defined.**). | Create a visible highlight in a buffer which is unaffected by editing.(pg. 101). |
| hilite_destroy(pg. **Error! Bookmark not defined.**). | Destroy one or more visible highlights.(pg. 102). |
| inq_mark_size(pg. **Error! Bookmark not defined.**). | Get size of selected area(pg. 116). |
| inq_marked(pg. **Error! Bookmark not defined.**). | Get extent of selected area(pg. 116). |
| inq_scrap(pg. **Error! Bookmark not defined.**). | Get information about the current scrap(pg. 121). |
| inside_region(pg. **Error! Bookmark not defined.**). | Check if cursor is inside a selected region.(pg. 127). |
| mark(pg. **Error! Bookmark not defined.**). | Set selected region scrap type(pg. 142). |
| paste(pg. **Error! Bookmark not defined.**). | Paste scrap into buffer(pg. 150). |
| print_block(pg. **Error! Bookmark not defined.**). | Print a buffer or region of a buffer to a file in PostScript(pg. 153). |
| put_selection(pg. **Error! Bookmark not defined.**). | Make text available for the clipboard(pg. 156). |
| raise_anchor(pg. **Error! Bookmark not defined.**). | Unhighlight selected area(pg. 157). |
| set_scrap_info(pg. **Error! Bookmark not defined.**). | Change information about the scrap(pg. 184). |
| swap_anchor(pg. **Error! Bookmark not defined.**). | Swap cursor position with other end of selected region(pg. 203). |

## String manipulation

These functions are used to manipulate the **string** data type.

| | |
|---|---|
| NLS(pg. **Error! Bookmark not defined.**). | Get NLS (nationalized string) version of string(pg. 148). |
| atof(pg. **Error! Bookmark not defined.**). | Convert string to floating-point(pg. 48). |
| atoi(pg. **Error! Bookmark not defined.**). | Convert string to decimal or hex number(pg. 48). |
| basename(pg. **Error! Bookmark not defined.**). | Return pathname of a file-spec string(pg. 50). |
| compress(pg. **Error! Bookmark not defined.**). | Remove multiple spaces in a string(pg. 59). |
| convert_white_space(pg. **Error! Bookmark not defined.**). | Convert tabs to spaces or spaces to tabs(pg. 60). |
| cvt_to_object(pg. **Error! Bookmark not defined.**). | Convert a string to an internal data type(pg. 67). |
| dirname(pg. **Error! Bookmark not defined.**). | Return directory part of a file-spec string(pg. 75). |
| filename_extension(pg. **Error! Bookmark not defined.**). | Extract file extension from a string(pg. 86). |
| filename_match(pg. **Error! Bookmark not defined.**). | Compare a filename against a file expression(pg. 89). |
| format(pg. **Error! Bookmark not defined.**). | Evaluate formatted string(pg. 93). |
| gsub(pg. **Error! Bookmark not defined.**). | Global search and replace in a string(pg. 101). |
| index(pg. **Error! Bookmark not defined.**). | Search string for a leftmost sub-string(pg. 102). |
| ltrim(pg. **Error! Bookmark not defined.**). | Strip leading white space from a string(pg. 141). |
| lower(pg. **Error! Bookmark not defined.**). | Convert string to lower case(pg. 140). |
| rindex(pg. **Error! Bookmark not defined.**). | Look for last occurence of a pattern in a string(pg. 171). |
| simplify_filename(pg. **Error! Bookmark not defined.**). | Remove redundant directory names from a flename(pg. 196). |
| split(pg. **Error! Bookmark not defined.**). | Split a string into tokens(pg. 199). |
| split_url(pg. **Error! Bookmark not defined.**). | Split a string containing a URL into its components(pg. 199). |
| sprintf(pg. **Error! Bookmark not defined.**). | Formatted printing to a string(pg. 200). |
| sscanf(pg. **Error! Bookmark not defined.**). | Unimplemented primitive(pg. 200). |
| strerror(pg. **Error! Bookmark not defined.**). | Return string corresponding to an operating system |

| | |
|---|---|
| | error(pg. 201). |
| string_count(pg. **Error! Bookmark not defined.**). | Count occurences of characters in a string(pg. 202). |
| string_match(pg. **Error! Bookmark not defined.**). | Return length of initial matching substring.(pg. 202). |
| strlen(pg. **Error! Bookmark not defined.**). | Length of string(pg. 202). |
| strtol(pg. **Error! Bookmark not defined.**). | Scan string and convert number(pg. 202). |
| sub(pg. **Error! Bookmark not defined.**). | Search and replace inside a string(pg. 203). |
| substr(pg. **Error! Bookmark not defined.**). | Extract substring(pg. 203). |
| tokenize(pg. **Error! Bookmark not defined.**). | Split a string into tokens(pg. 207). |
| trim(pg. **Error! Bookmark not defined.**). | Strip characters from end of string(pg. 210). |
| upper(pg. **Error! Bookmark not defined.**). | Convert string to upper case or captilize first letter only(pg. 212). |

## Status & Command line

These functions are used to manipulate the status line and command prompt area.

| | |
|---|---|
| echo_line(pg. **Error! Bookmark not defined.**). | Set flags affecting Line:/Col: area(pg. 79). |
| inq_cmd_line(pg. **Error! Bookmark not defined.**). | Get command line message(pg. 108). |
| inq_command(pg. **Error! Bookmark not defined.**). | Get name of macro invoking command line prompt(pg. 108). |
| inq_message(pg. **Error! Bookmark not defined.**). | Get message from the command line(pg. 116). |
| inq_line_col(pg. **Error! Bookmark not defined.**). | Get current value of Line:/Col: status area(pg. 113). |
| inq_prompt(pg. **Error! Bookmark not defined.**). | Determine if we are at a command prompt or not(pg. 120). |

## System Calls

These functions provide access to system dependent system calls. Some of them may require special privileges, although CRiSP is not installed by default with these special privileges.

| | |
|---|---|
| chroot(pg. **Error! Bookmark not defined.**). | Change root directory(pg. 56). |
| getgid(pg. **Error! Bookmark not defined.**). | get group ID(pg. 98). |
| gethostid(pg. **Error! Bookmark not defined.**). | Return 32-bit host ID value(pg. 98). |
| gethostname(pg. **Error! Bookmark not defined.**). | Get name of current host(pg. 94). |
| getpid(pg. **Error! Bookmark not defined.**). | get current process ID(pg. 98). |
| getrlimit(pg. **Error! Bookmark not defined.**). | Get user resource limits(pg. 99). |
| getuid(pg. **Error! Bookmark not defined.**). | get user ID(pg. 99). |
| setegid(pg. **Error! Bookmark not defined.**). | Set effective group-id(pg. 193). |
| seteuid(pg. **Error! Bookmark not defined.**). | Set effective user-id(pg. 193). |
| setgid(pg. **Error! Bookmark not defined.**). | Set group-id(pg. 193). |
| setrgid(pg. **Error! Bookmark not defined.**). | Set real group-id(pg. 193). |
| setrlimit(pg. **Error! Bookmark not defined.**). | Set user resource limits(pg. 194). |
| setruid(pg. **Error! Bookmark not defined.**). | Set real user-id(pg. 194). |
| setuid(pg. **Error! Bookmark not defined.**). | Set user-id(pg. 194). |

## Tagging functions

These functions are used to parse a buffer or file and create a tag database or return a list of objects defined in the file.

| | |
|---|---|
| tag_close_db(pg. **Error! Bookmark not defined.**). | Close tag database(pg. 205). |
| tag_list_tags(pg. **Error! Bookmark not defined.**). | Enumerate tags in a buffer.(pg. 205). |

## Time & Date functions

These functions provide ways of accessing date and time information.

| | |
|---|---|
| cftime(pg. **Error! Bookmark not defined.**). | Formatted date conversion(pg. 54). |
| ctime(pg. **Error! Bookmark not defined.**). | Convert time to printable string(pg. 66). |
| date(pg. **Error! Bookmark not defined.**). | Get current date information(pg. 67). |
| inq_clock(pg. **Error! Bookmark not defined.**). | Get current CPU time(pg. 108). |

| | |
|---|---|
| inq_idle_default(pg. **Error! Bookmark not defined.**). | Get current idle timer/keystroke count(pg. 110). |
| inq_idle_time(pg. **Error! Bookmark not defined.**). | Get current status of idle timeout/keycount(pg. 111). |
| inq_time(pg. **Error! Bookmark not defined.**). | Determine when buffer was last modified(pg. 122). |
| set_idle_default(pg. **Error! Bookmark not defined.**). | Set idle timer and keystroke values(pg. 182). |
| sleep(pg. **Error! Bookmark not defined.**). | Pause execution for a number of seconds(pg. 196). |
| time(pg. **Error! Bookmark not defined.**). | Get current time of day(pg. 206). |
| time_passed(pg. **Error! Bookmark not defined.**). | Check if real-time has elapsed(pg. 207). |

## Window Manipulation

These functions manipulate windows.

| | |
|---|---|
| borders(pg. **Error! Bookmark not defined.**). | Toggle character mode window borders(pg. 52). |
| change_window(pg. **Error! Bookmark not defined.**). | Select new character mode window(pg. 55). |
| change_window_pos(pg. **Error! Bookmark not defined.**). | Change character mode window boundary(pg. 55). |
| create_tiled_window(pg. **Error! Bookmark not defined.**). | Create a character mode tiled window(pg. 65). |
| create_window(pg. **Error! Bookmark not defined.**). | Create a character mode popup window(pg. 65). |
| delete_edge(pg. **Error! Bookmark not defined.**). | Delete a character mode window(pg. 70). |
| delete_window(pg. **Error! Bookmark not defined.**). | Delete a character mode window(pg. 72). |
| display_enable(pg. **Error! Bookmark not defined.**). | Enable/display CRiSP windows(pg. 75). |
| display_windows(pg. **Error! Bookmark not defined.**). | Prepare to use create_tiled_window(pg. 77). |
| end_of_window(pg. **Error! Bookmark not defined.**). | Move cursor to bottom of window(pg. 83). |
| inq_borders(pg. **Error! Bookmark not defined.**). | Get value of character mode border flag(pg. 104). |
| inq_menu_bar(pg. **Error! Bookmark not defined.**). | Get window id of character mode menu bar(pg. 116). |
| inq_top_left(pg. **Error! Bookmark not defined.**). | Get co-ordinates of top left hand side of window(pg. 122). |
| inq_views(pg. **Error! Bookmark not defined.**). | Count how many windows display a buffer(pg. 123). |
| inq_window(pg. **Error! Bookmark not defined.**). | Get current window id(pg. 123). |
| inq_window_buf(pg. **Error! Bookmark not defined.**). | Get buffer id associated with a window(pg. 123). |
| inq_window_color(pg. **Error! Bookmark not defined.**). | Get current background and foreground color of a window(pg. 123). |
| inq_window_flags(pg. **Error! Bookmark not defined.**). | Retrieve current window display attributes(pg. 124). |
| inq_window_info(pg. **Error! Bookmark not defined.**). | Get information about a character mode window(pg. 125). |
| inq_window_size(pg. **Error! Bookmark not defined.**). | Get size of a character mode window(pg. 125). |
| inq_window_system(pg. **Error! Bookmark not defined.**). | Get information about the current GUI environment(pg. 125). |
| list_of_windows(pg. **Error! Bookmark not defined.**). | return list of windows within the current screen(pg. 139). |
| move_edge(pg. **Error! Bookmark not defined.**). | Move boundary between character mode windows(pg. 147). |
| next_window(pg. **Error! Bookmark not defined.**). | Set window to the next window(pg. 148). |
| set_top_left(pg. **Error! Bookmark not defined.**). | Set top-left position of a window(pg. 189). |
| set_window(pg. **Error! Bookmark not defined.**). | Change window(pg. 190). |
| set_window_cmap(pg. **Error! Bookmark not defined.**). | Set character mode character map for a window(pg. 190). |
| set_window_flags(pg. **Error! Bookmark not defined.**). | Set display attributes for a window(pg. 191). |
| set_window_layer(pg. **Error! Bookmark not defined.**). | Set Z-axis for a character mode window(pg. 192). |
| set_window_title(pg. **Error! Bookmark not defined.**). | Set title at top of a window(pg. 192). |
| top_of_window(pg. **Error! Bookmark not defined.**). | Move cursor to top of current window(pg. 208). |
| window_color(pg. **Error! Bookmark not defined.**). | Set character mode fg & bg colors(pg. 215). |

## Alphabetical function summary

{button !,JK(idx_misc)} {button A,JK(idx_a)} {button B,JK(idx_b)} {button C,JK(idx_c)} {button D,JK(idx_d)} {button E,JK(idx_e)} {button F,JK(idx_f)} {button G,JK(idx_g)} {button H,JK(idx_h)} {button I,JK(idx_i)} {button K,JK(idx_k)} {button L,JK(idx_l)} {button M,JK(idx_m)} {button N,JK(idx_n)} {button O,JK(idx_o)} {button P,JK(idx_p)} {button Q,JK(idx_q)} {button R,JK(idx_r)} {button S,JK(idx_s)} {button T,JK(idx_t)} {button U,JK(idx_u)} {button V,JK(idx_v)} {button W,JK(idx_w)}

| | |
|---|---|
| !(pg. **Error! Bookmark not defined.**). | Logical not operator(pg. 38). |
| !=(pg. **Error! Bookmark not defined.**). | Inequality operator(pg. 38). |
| %(pg. **Error! Bookmark not defined.**). | Integer modulus(pg. 38). |
| %=(pg. **Error! Bookmark not defined.**). | Integer modulus assignment(pg. 38). |
| &(pg. **Error! Bookmark not defined.**). | Logical and(pg. **Error! Bookmark not defined.**). |
| &&(pg. **Error! Bookmark not defined.**). | Conditional and operator(pg. **Error! Bookmark not defined.**). |
| &=(pg. **Error! Bookmark not defined.**). | Logical and assignment operator(pg. **Error! Bookmark not defined.**). |
| *(pg. **Error! Bookmark not defined.**). | Multiply/string replicate(pg. 39). |
| *=(pg. **Error! Bookmark not defined.**). | Multiply and assign value(pg. 39). |
| +(pg. **Error! Bookmark not defined.**). | Arithmetic, string or list addition(pg. **Error! Bookmark not defined.**). |
| ++(pg. **Error! Bookmark not defined.**). | Pre-increment variable(pg. **Error! Bookmark not defined.**). |
| +=(pg. **Error! Bookmark not defined.**). | Add assignment operator(pg. **Error! Bookmark not defined.**). |
| -(pg. **Error! Bookmark not defined.**). | Subtract operator(pg. **Error! Bookmark not defined.**). |
| --(pg. **Error! Bookmark not defined.**). | Pre-decrement(pg. **Error! Bookmark not defined.**). |
| -=(pg. **Error! Bookmark not defined.**). | Subtract assignment operator(pg. **Error! Bookmark not defined.**). |
| /(pg. **Error! Bookmark not defined.**). | Divide operator(pg. **Error! Bookmark not defined.**). |
| /=(pg. **Error! Bookmark not defined.**). | Divide assignment operator(pg. **Error! Bookmark not defined.**). |
| <(pg. **Error! Bookmark not defined.**). | Less than(pg. **Error! Bookmark not defined.**). |
| <<(pg. **Error! Bookmark not defined.**). | Bit left-shift operator(pg. **Error! Bookmark not defined.**). |
| <<=(pg. **Error! Bookmark not defined.**). | Bit left-shift assignment operator(pg. **Error! Bookmark not defined.**). |
| <=(pg. **Error! Bookmark not defined.**). | Less than or equal(pg. **Error! Bookmark not defined.**). |
| =(pg. **Error! Bookmark not defined.**). | Assignment operator(pg. 42). |
| ==(pg. **Error! Bookmark not defined.**). | Equality(pg. 42). |
| >(pg. **Error! Bookmark not defined.**). | Greater than(pg. 42). |
| >=(pg. **Error! Bookmark not defined.**). | Greater or equal(pg. 42). |
| >>(pg. **Error! Bookmark not defined.**). | Bit right-shift operator(pg. 43). |
| >>=(pg. **Error! Bookmark not defined.**). | Bit right-shift assignment operator(pg. 43). |
| ^(pg. **Error! Bookmark not defined.**). | Exclusive or(pg. **Error! Bookmark not defined.**). |
| ^=(pg. **Error! Bookmark not defined.**). | Exclusive or assignment operator(pg. **Error! Bookmark not defined.**). |
| \|(pg. **Error! Bookmark not defined.**). | Logical or operator(pg. **Error! Bookmark not defined.**). |
| \|=(pg. **Error! Bookmark not defined.**). | Logical or assignment operator(pg. **Error! Bookmark not defined.**). |
| \|\|(pg. **Error! Bookmark not defined.**). | Conditional or(pg. **Error! Bookmark not defined.**). |
| ~(pg. **Error! Bookmark not defined.**). | Bit-wise not operator(pg. **Error! Bookmark not defined.**). |

| | |
|---|---|
| __stack_frame(pg. **Error! Bookmark not defined.**). | Unsupported operator(pg. 43). |
| _bad_key(pg. **Error! Bookmark not defined.**). | CRiSP callback for handling command line prompts(pg. 44). |
| _extension(pg. **Error! Bookmark not defined.**). | CRiSP callback to handle unknown filetypes(pg. 85). |
| _fatal_error(pg. **Error! Bookmark not defined.**). | CRiSP callback when internal error detected(pg. 86). |
| _prompt_begin(pg. **Error! Bookmark not defined.**). | CRiSP callback to implement command line(pg. 155). |
| _prompt_end(pg. **Error! Bookmark not defined.**). | CRiSP callback to implement command line(pg. 155). |
| _startup_complete(pg. **Error! Bookmark not defined.**). | Macro called on startup(pg. 200). |

| | |
|---|---|
| abort(pg. **Error! Bookmark not defined.**). | Abort CRiSP execution immediately(pg. 44). |
| above(pg. **Error! Bookmark not defined.**). | Greater than comparison(pg. 44). |
| above_eq(pg. **Error! Bookmark not defined.**). | Greater than or equal(pg. 44). |

## ! iexpr

### Description

Returns the logical not of the expression expr. expr must evaluate to an integer expression.

### Example

```
int     i = 23;

i = !i;
/* i now has the value of 0. */

if (!i)
    message("i is zero");
else
```

```
                message("i is not zero");
```

### Return value

Returns 1 if **expr** evaluates to zero; returns 0 if **expr** is non-zero.

## expr1 != expr2

### Return value

Returns 1 if expr1 is not equal to expr2; 0 if expr1 is equal to expr2.

## iexpr1 % iexpr2

### Description

This is the modulus function. The example below shows the results of using negative numbers.

### Example

The following example show the various pathological cases of the (%) function:

```
mod_test()
{
        message("%d %d %d %d %d",
                4 % -3,
                4 % -3,
                -4 % 3,
                -4 % -4,
                -4 % 0);
}
```

The output is:

```
1 1 -1 -1 0
```

### Return value

Returns the integral remainder of the expression (iexpr1)/(iexpr2).

## ivar %= iexpr

### Description

This macro is the same as:

```
ivar = ivar % iexpr;
```

ivar is the name of an integer variable; iexpr is an expression which evaluate to an integer.

See (%) for details on the pathological use of negative numbers.

### Return value

Returns the value assigned to ivar.

## iexpr1 & iexpr2

### Return value

Returns the bit-wise AND of iexpr1 and iexpr2. A 32-bit integer result is returned.

## ivar &= iexpr1;

### Return value

The value of ivar logically ANDed with iexpr.

### Description

This is equivalent to:

```
ivar = ivar & iexpr;
```

## expr1 && expr2

### Description

This is the conditional AND macro. If *expr1* evaluates to non-zero, then *expr2* is evaluated. If *expr1* evaluates to zero, then *expr2* is not evaluated. Thus, it is safe for *expr1* and *expr2* to have side-effects.

*expr1* and/or *expr2* may be integer or string expressions. If either expression is a string expression then the implicit test is for a null string. For example, **`(s1 && s2)`** is treated as if the expression were: **`(s1 != "" && s2 != "")`**. This gives a similar feeling to the C language.

### Example

The following example tests var1 and if it is non-zero, increments var2 and tests to see if it is greater than 10.

```
if (var1 && ++var2 > 10)
     message("var2 > 10 and var1 is non-zero");
```

### Return value

Returns 1 if *expr1* evaluates to non-zero and *expr2* evaluates to non-zero; 0 otherwise.

## expr1 * expr2

### Description

The '*' operator is used to perform multiplication. It can be used on numbers (integers or float's) and also it can be used to replicate strings. Automatic co-oercion between float and integer is performed. If either argument is a float, then the result will be a float.

If either *expr1* or *expr2* is a string, then string replication will be performed. For example, **`4*"x"`** will return the string **`"xxxx"`**. The number and string may be in either order.

All other combinations of types are invalid.

### Return value

The value of *iexpr1* multiplied by *iexpr2* or the specified string replicated the number of times.

## ivar *= iexpr1;

### Description

This is equivalent to:

```
ivar = ivar * iexpr;
```

### Return value

The value of *ivar* multiplied by *iexpr*.

## expr1 + expr2

### Description

The plus operator is used to add, or concatenate two expressions. The operation is dependent on the type of the two specified expressions. The processing is performed in the following order.

If *expr1* or *expr2* is a list, then a new list is returned which is the concatenation of *expr1* and *expr2*.

If *expr1* or *expr2* is a string, then the other operand is converted to a string, and a string is returned.

If either operand is a floating point number then the other the result is the sum of the two expressions.

Otherwise the integer value of the sum of the two expressions is returned.

### Return value

A list value if either operand is a list; a string if either operand is a string; a float if either operand is a float; otherwise an integer value.

### Example

The following example illustrates how to convert a number to a string:

```
string s = "" + 2.3;
```

## ++ivar

### Description

This macro forms the pre-increment instruction. (There is no post-increment equivalent, due to limitations in the syntax of the language).

The variable *ivar* is incremented and the value may then be used in an expression.

### Example

The following example can be used for controlling loops:

```
int     i = 0;
while (++i < 10) {
      ..
      ..
}
```

### Return value

Returns the value (ivar+1).

## var += expr;

### Description

This is equivalent to:

```
var = var + expr;
```

### Return value

Adds the expression expr to the variable 'var' and returns the value. Addition is performed the same as the '+' operator, i.e. float's and int's will be cast appropriately.

## iexpr1 - iexpr2

### Description

iexpr1 and iexpr2 must both evaluate to integer expressions.

### Return value

Returns the integer value iexpr1 minus iexpr2.

## --ivar

### Description

This macro forms the pre-decrement instruction. (There is no post-increment equivalent, due to limitations in the syntax of the language).

The variable ivar is decremented and the value may then be used in an expression.

### Return value

Returns the value (ivar-1).

## ivar -= iexpr

### Description

This is equivalent to:

```
ivar = ivar - iexpr;
```

### Return value

Returns the value (ivar-iexpr).

## expr1 / iexpr2

### Description

If *iexpr2* is zero, then *iexpr1* is returned.

### Return value

Returns the value of *iexpr1* divided by *iexpr2*.

## ivar /= iexpr

### Return value

Returns the value ivar divided by iexpr.

### Description

This is equivalent to:

```
ivar = ivar / iexpr;
```

## expr1 < expr2

### Description

expr1 & expr2 must either be both integer expressions or string expressions.

### Return value

Returns 1 if expr1 and expr2 are integer values and expr1 is less than expr2; returns 1 if expr1 & expr2 are string expressions and expr1 lexicographically precedes expr2. Returns 0 otherwise.

## expr1 << expr2

### Return value

Returns expr1 shifted left by expr2 bit positions. expr1 and expr2 must be integer expressions.

## expr1 <= expr2

### Description

expr1 & expr2 must either be both integer expressions or string expressions.

### Return value

Returns 1 if expr1 and expr2 are integer values and expr1 is less than or equal to expr2; returns 1 if expr1 & expr2 are string expressions and expr1 lexicographically precedes expr2 or is equal to expr2. Returns 0 otherwise.

## var = expr

### Description

This is the assignment operator. var is the name of a symbol, and expr must be an expression which evaluates to the same type as 'var', unless 'var' has been declared as a polymorphic variable. (See (declare)).

**var** and **expr** must agree in type. They can have integer, string or list type. For list variables, it is acceptable for expr to be NULL or omitted, in which case the storage allocated to the variable var is freed.

### Return value

Returns the value of **expr**.

### Example

```
int     i;
string  s;
```

```
        list    l;

        i = 1 + 2;
        s = "fred" + " bloggs";
        l = quote_list(1, 2, 3);
        l = NULL;
```

## var <<= expr

### Return value

Returns value of **var** shifted left by **expr** bit positions. var must be an integer variable and expr must be an integer expression.

## expr1 == expr2

### Description

Lists cannot be compared.

### Return value

Returns 1 if expr1 is equal to expr2. Returns 0 otherwise.

## expr1 > expr2

### Description

expr1 & expr2 must either be both numeric expressions or string expressions.

### Return value

Returns 1 if expr1 and expr2 are integer values and expr1 is greater than expr2; returns 1 if expr1 & expr2 are string expressions and expr1 lexicographically follows expr2. Returns 0 otherwise.

## expr1 >= expr2

### Description

expr1 & expr2 must either be both numeric expressions or string expressions.

### Return value

Returns 1 if expr1 and expr2 are integer values and expr1 is greater than or equal to expr2; returns 1 if expr1 & expr2 are string expressions and expr1 lexicographically precedes expr2 or is equal to expr2. Returns 0 otherwise.

## expr1 >> expr2

### Return value

Returns *expr1* shifted right by expr2 bit positions. *expr1* and *expr2* must be integer expressions.

## var >>= expr

### Return value

Returns value of var shifted right by expr bit positions. var must be an integer variable and expr must be an integer expression.

## iexpr1 | iexpr2

### Return value

Returns the logical OR of integer expression iexpr1 and iexpr2.

## ivar |= iexpr1

### Description

This is equivalent to:

```
ivar = ivar | iexpr;
```

### Return value

The value of ivar logically ORed with iexpr.

## expr1 || expr2

### Return value

Returns 1 if expr1 OR expr2 evaluate to non-zero.

### Description

This is the conditional OR function. If expr1 evaluates to non-zero then expr2 is not evaluated. If expr1 evaluates to zero, then the result of expr2 is returned.

## iexpr1 ^ iexpr2

### Return value

Returns the logical XOR of integer expression iexpr1 and iexpr2.

## ivar ^= iexpr1

### Return value

The value of ivar logically XORed with iexpr.

### Description

This is equivalent to:

```
ivar = ivar ^ iexpr;
```

## ~iexpr

### Return value

Returns the 1's complement of the integer expression iexpr.

## __stack_frame(type, size)

### Description

This primitive is generated and used by the high level crunch compiler. It should not normally be used by macro programmers.

This primitive is used to allocate space for a stack frame or space for static/global variables.

*type* specifies which heap area is affect, and *size* is the number of bytes allocated.

This primitive is automatically generated by the crunch compiler to declare space for variables.

NOTE: This primitive is not currently implemented or supported.

### Return value

None

## _bad_key()

### Description

*_bad_key()* is not actually a builtin primitive, but is used by CRiSP to implement the command history and abbreviation features.

The macro _bad_key is called from within CRiSP when a bad key is pressed during a prompt. This macro can use the *inq_message()* and *inq_cmd_line()* macros to see what the state of the status line is. It can also call *read_char()* to retrieve the key which caused the problem -- this allows the _bad_key macro to distinguish between context sensitive help, or abbreviations etc.

The macro should return a string value, in which case this is taken as the current input for the response field (it will be highlighted as the default value). The *push_back()* macro can be used to force acceptance of a default response.

See the history.cr macro file for an example of how to use the *_bad_key()* macro.

### Return value

A string representing the default for the field (which does not terminate the input), or an integer. An integer value of 0 will cancel the command input. A value of 1 will cause the invalid key to be accepted as part of the input. A value of 2 will abort the command input but not display the **Command cancelled** message.

## abort()

### Description

This macro is used to abort CRiSP - it causes an unconditional exit from CRiSP after resetting the keyboard state.

It should be used with extreme care since no checks are made to see if buffers need to be saved, etc.

Its primary use will be in debugging macros.

### Return value

Nothing.

### See also

exit()(pg. 85).

## above(expr1, expr2)

### Description

This function provides a functional form of the '>' operator. expr1 & expr2 must either be both integer expressions or string expressions.

### Return value

Returns 1 if expr1 and expr2 are integer values and expr1 is greater than expr2; returns 1 if expr1 & expr2 are string expressions and expr1 lexicographically follows expr2. Returns 0 otherwise.

## above_eq(expr1, expr2)

### Description

This function is a functional form of the '>=' operator. expr1 & expr2 must either be both integer expressions or string expressions.

### Return value

Returns 1 if expr1 and expr2 are integer values and expr1 is greater than or equal to expr2; returns 1 if expr1 & expr2 are string expressions and expr1 lexicographically follows expr2 or is equal to expr2. Returns 0 otherwise.

### See also

>=(pg. 42).

## abs(num)

### Description

This function returns the absolute value of the specified expression. The *num* argument may be either an integer or floating point value.

### See also

min()(pg. 144)., max()(pg. 143).

## access(string file, int mode)

### Description

This function is used to check the accessibility of a file. The *file* parameter is the name of a directory or file which is to be tested and *mode* are a set of access bits which are used to check the effective access.

The meaning of mode is operating system dependent (refer to <unistd.h> or *access(2)*). Most operating systems support the following bit definitions:

| | |
|---|---|
| 0 | Check if file exists |
| 1 | Check to see if file is executable |
| 2 | Check if file is writable |
| 3 | Check if file is readable |

### Return value

On successful return this function returns >= 0; -1 is returned on an error, and the global variable *errno* is set to the reason for the failure.

### See also

stat()(pg. 201).

## acknowledge_changed_buffer(buf_id, reason)

### Description

This function is used after a files status has changed. It is used by the popup dialog boxes which tell you a file has been modified or deleted to avoid repeated diagnostics about the particular file.

The first parameter is the buffer which is to be acknowledged. The second parameter acknowledges a particular reason, e.g. the file being modified or deleted. This means that if CRiSP detects a file has been modified then whilst you are acknowledging the state change, you do not lose information about the file subsequently being deleted.

The reason code should be the return value from the *inq_changed_buffers()* primitive.

### Return value

-1 if the buffer does not exist. 0 on success.

### See also

inq_changed_buffers()(pg. 108).

## acos(float_expr)

### Description

Returns the inverse cosine value of the expression, in the range [-pi/2, pi/2]. float_expr must be in the range [-1, 1].

### Return value

Arc-cosine of the passed argument (in radians).

### See also

sin()(pg. 196)., cos()(pg. 61)., tan()(pg. 206)., asin()(pg. 46)., atanh()(pg. 47).

## arg_list()

### Description

This primitive returns a list representing the arguments which have been passed to the calling macro. This is useful when writing macros which take arbitrary arguments and need to pass them to other macros or primitives.

### Return value

List of arguments passed to calling macro.

### See also

get_parm()(pg. 95)., put_parm()(pg. 156).

## asin(float_expr)

### Description

Returns the inverse sine value of the expression, in the range [-pi/2, pi/2]. float_expr must be in the range [-1, 1].

### Return value

Arc-sine of the passed argument (in radians).

### See also

sin()(pg. 196)., cos()(pg. 61)., tan()(pg. 206)., acos()(pg. 46)., atanh()(pg. 47).

## atan(float_expr)

### Description

Returns the inverse tangent value of the expression, in the range [-pi/2, pi/2].

### Return value

Arc-tangent of the passed argument (in radians).

### See also

atan2()(pg. 46)., sin()(pg. 196)., cos()(pg. 61)., tan()(pg. 206)., asin()(pg. 46)., acos()(pg. 46).

## atan2(x, y)

### Description

Returns the inverse tangent of y/x in the range [-pi, pi].

### Return value

Arc-tangent of (y/x) (in radians).

### See also

atan()(pg. 46)., sin()(pg. 196)., cos()(pg. 61)., tan()(pg. 206)., asin()(pg. 46).

## atanh(float_expr)

### Description

Returns the inverse hyperbolic tangent value of the expression.

### Return value

Hyperbolic arc-tangent of the passed argument (in radians).

### See also

atan2()(pg. 46)., sin()(pg. 196)., cos()(pg. 61)., tan()(pg. 206)., tanh()(pg. 206)., asin()(pg. 46)., acos()(pg. 46).

## assign_to_key([key], [macro], ...)

### Description

This macro is used to bind a macro to a function key. The key is given by the string parameter, key, and if omitted is prompted for. key may be specified as either an internal key code, or may be specified by a portable abbreviation. The portable abbreviations are given below.

**macro** is a string containing the name of a macro to execute when that key is pressed.

The sequence of key/macro name pairs may be repeated indefinitely. Repeating key/macro names is slightly faster and generates smaller code than using individual assign_to_key() macros for each keystroke.

If **key** evaluates to more than one keystroke, then this is defined as a multi-key sequence, i.e. more than one key must be pressed to execute the macro. In this case, an internal key code is assigned for this key, as if it were a builtin key. This internal key code is returned as the value of this macro. Muilti-key sequences do not time out between key presses unlike the other internal keys when there is an ambiguity.

(The backslash character can be used to turn off the effect of the special characters). key is a string which may take one of the following forms:

| **Key** | **Description** |
|---|---|
| **x** | The actual key 'x'. |
| **^x** | The key control-x. |
| **#123** | The key whose internal key code is 123. This method of defining keys is not recommended. It is provided as an *escape* mechanism where predefined string names are not available under CRiSP. |
| **xyz** | The sequence of keys x, followed by y, followed by z. This allows multiple keys to be bound to a macro. Note that this is not the same as using set_term_keyboard() to define a key. |
| **<Fn>** | when n is in the range 1..255. This is a synonym for Function Key n. |
| **<Ctrl-Fn>** **<Shift-Fn>** **<Alt-Fn>** **<Meta-Fn>** | Ctrl, Shift and Alt versions of function keys. (Alt-Fn and Meta-Fn are synonymous). |
| **<Alt-x> <Meta-x>** | where n is an upper or lower case letter. The case of the letter is ignored. On the Sun keyboard, LEFT/RIGHT act as the ALT key for the main keyboard only. |
| **<Alt-n> <Meta-n>** | where n is in the range 0..9. On a PC keyboard this means the top numeric keys with the ALT key pressed. On a Sun-3 keyboard, this is the same as LEFT/RIGHT pressed whilst hitting a digit key. |
| **<Ins>** | The keypad Insert key. |
| **<Del>** | The DELETE key. |
| **<Backspace>** | The backspace key. |
| **<Keypad-minus>** | The '-' key on the keypad. |
| **<Keypad-plus>** | The '+' key on the keypad. |
| **<Enter>** | The ENTER or RETURN key. |
| **<Esc>, <Tab>, <PgDn>, <PgUp>, <Left>, <Right>, <Up>, <Down>, <Home>, <End>** | Same as keys labelled as such on keyboard. On keyboards which don't have markings for these keys, various approximations are possible. Refer to the tty/*.cr files for further details. |

## Return value

Key value assigned to sequence or -1 if key sequence invalid or operation aborted.

## See also

assign_to_raw_key()(pg. 48)., load_keyboard_file()(pg. 139).

## assign_to_raw_key([key], [macro], ...)

### Description

This function is similar to the **assign_to_key** macro but is used to bind a macro to a raw scan code. This is needed if you want to assign a different macro, for example, to the cursor keys on a PC keyboard and have different bindings for the keypad keys. Normally CRiSP will consider the gray cursor keys, along with the **Insert, Delete, Home, End, PgUp** and **PgDn** keys as identical to their numeric keypad counterparts.

Assigning macro bindings using scan codes is very system and keyboard specific. For instance, a macro written for Windows will not run properly if run on an X Windows machine, and a user running an application on a DEC Alpha keyboard will likely get different scan codes compared, say, to a user running on a Solaris 2 keyboard.

This function and the related primitives were added so that the EDT emulation macro could be written.

In order to use scan codes in the **key** string, you will need to use the **inq_raw_kbd_char()** primitive and use the **"#nnnn"** notation to specify a numeric scan code. Using the normal **<..>** format is not likely to result in anything sensible happening.

### See also

assign_to_key()(pg. 47).,

## atof(string)

### Description

This macro converts string, which is a string representation of a floating point number, to its binary value.

### Return value

Returns floating point value of string.

### See also

atoi()(pg. 48).

## atoi(string, [char], [hex])

### Description

This macro converts string, which is a string representation of a decimal number to binary. If char is specified and is zero, then the ascii value of the first character in string is returned.

If *hex* is specified then the specified string will be parsed as an octal, decimal or hexadecimal number depending on the prefix. If the number starts with 0x or 0X then the number will be parsed as a hex number; otherwise if the number starts with a zero then the number will be decoded in octal. Lastly it will be decoded as a decimal number.

### Return value

Returns integer value of argument string treated as an ascii number or the ascii value of the first character in string if char is zero.

### See also

atof()(pg. 48)., strtol()(pg. 202).

## attach_buffer(bufnum)

### Description

This command attaches the specified buffer to the currently selected window. This means that the next time the window is refreshed the specified buffer will be displayed in the current window.

When the specified buffer is attached to the current window, the top title of the window is changed to reflect the buffer or filename associated with the buffer.

This call is necessary after changing a window with *set_window()*. Take care to ensure that when a window is refreshed on the screen that it always has a buffer attached to it, otherwise CRiSP may core dump.

This command is only intended for use by macro programmers, and macros using *attach_buffer()* should be frequently saved during development to avoid unexpected crashes.

### Return value

No value is returned.

### Example

The following example shows the outline of a macro which attaches a new buffer to the current window.

```
int     newbuf = user_func();
attach_buffer(newbuf);

/* Refresh screen with new buffer contents. */
refresh();
```

### See also

create_buffer()(pg. 62)., create_window()(pg. 65)., inq_buffer()(pg. 105)., inq_window()(pg. 123).

## autoload(file, [macro1, [macro2], ..])

### Description

This macro is used to define the whereabouts of macros. Whenever CRiSP encounters a reference to a macro which is not currently defined, it searches an internal list to see if an *autoload()* macro has mentioned it. If so, the file parameter says in which file it can find the macro.

*file* should be specified as the basename of the macro file (i.e. without a .m or .cm suffix). This lets CRiSP search the CRPATH variable to see if it can find a .m or .cm file.

If it finds the file, it then performs a *load_macro()* operation. If the macro definition is still not resolved after auto-loading the macro file, then CRiSP prints an error message.

An arbitrary number of macro names may appear after the file parameter. CRiSP allows wildcarded macro names to be used, using the shell regular expression characters: *, ? and []. Care should be taken when using this facility as you can easily confuse CRiSP into believing macros are to be found in the associated file and overriding existing definitions.

autoloading is a useful feature since it speeds CRiSP up in that it doesn't need to load macros which aren't referenced.

If no macro arguments are supplied then the *file* parameter is appended to the search path list. When a macro is not found by the usual lookup mechanism, then the *file* parameter will be prefixed to the macro name and a search will be performed.

### Example

```
autoload("fred", "tom", "dick", "harry");
```

This example says that if the macros *tom()*, *dick()*, or *harry()* are mentioned then to look in the macro file *fred*. (CRiSP will search the default path looking for files: fred.cm, fred.m in each directory in the search path.

(The search current search path can be seen by accessing the environment variable **$CRPATH** from within CRiSP or a subshell of CRiSP. If **CRPATH** is not set before execution, then a suitable default is applied).

autoload("games");

This example specifies that any macro which cannot normally be found should be looked up in the 'games' sub-directory. The sub-directory will be searched in all directories mentioned in the **CRPATH** list.

### Return value

Nothing.

### See also

inq_macro()(pg. 115)., load_macro()(pg. 140)., require()(pg. 169).

## backspace([num])

### Description

This deletes the character to the left of the cursor in the current buffer. If the cursor is at the beginning of the line then the current line is appended to the end of the previous line.

If the cursor is on a virtual space, then the cursor moves back to the next tab stop or the next character position.

If the buffer is in overtype mode then the cursor simply moves the cursor backwards without deleting any characters.

If num is specified then the backspace primitive is repeated the specified number of times.

### Return value

Nothing.

### See also

delete_char()(pg. 70).

## basename(path, [noexpand[])

### Description

This function is similar to the Unix *basename(1)* utility. Given a path string, it returns the filename part of the string.

If **noexpand** is specified and is non-zero, then the filename will not have wildcards expanded before returning the basename.

### Example

**`basename("/tmp/fred")`** returns **`"fred"`**

### Return value

String containing the filename part (last component) of a file specification.

### See also

dirname()(pg. 75).

## beautify_buffer([buf_id], [file_type], [arg-list])

### Description

This primitive is used to reformat a buffer to prettify the code. This is useful for example, when working on code written by someone else but you want to make the code conform to either your own private layout style or the style of project you are working on. It is also useful after making drastic changes to a file, and the indentation structure has become inconsistent.

### Return value

Returns -1 on an error..

## beep([frequency], [duration])

### Description

This macro sends a bell or beep to the screen causing an audible sound. This macro can be used by error macros.

**Frequency** and **duration** are optional integers. If specified they specify the frequency (in Hz and milliseconds) for the sound. This is only supported for Windows; the parameters are ignored under

Unix and the character versions.

### Return value

Nothing.

## beginning_of_line()

### Description

Moves cursor to column 1 of the current buffer.

### Return value

Nothing.

### See also

end_of_line()(pg. 83)., move_abs()(pg. 147)., move_rel()(pg. 147).

## below(expr1, expr2)

### Description

This primitive is a functional version of the '<' operator. expr1 & expr2 must either be both integer expressions or string expressions.

### Return value

Returns 1 if expr1 and expr2 are integer values and expr1 is less than expr2; returns 1 if expr1 & expr2 are string expressions and expr1 lexicographically precedes expr2. Returns 0 otherwise.

### See also

above()(pg. 44).

## below_eq(expr1, expr2)

### Description

This primitive is a functional version of the '<=' operator. expr1 & expr2 must either be both integer expressions or string expressions.

### Return value

Returns 1 if expr1 and expr2 are integer values and expr1 is less than or equal expr2; returns 1 if expr1 & expr2 are string expressions and expr1 lexicographically follows expr2 or is equal to expr2. Returns 0 otherwise.

### See also

above_eq()(pg. 44).

## bookmark_list()

### Description

This macro is used to return a list containing all currently defined bookmarks, or **NULL** if no bookmarks are defined. The list is a list of 4-tuples in the following format:

```
{id, buf_id, line_no, col}
```

### Return value

List containing all currently defined bookmarks.

### See also

delete_bookmark()(pg. 69)., drop_bookmark()(pg. 79)., goto_bookmark()(pg. 100).

## borders([expr])

### Description

This macro can be used to turn off borders for windows. By default they are on.

If expr is omitted, then the current value is toggled. Otherwise the internal toggle is set to the value of expr. Non-zero means borders are on.

If expr is a string, then the current setting of the borders is not changed -- only the value is returned.

On some terminals, having borderless windows is less expensive on screen updating than on others, because of the overhead involved in drawing the borders around the windows.

### Return value

Returns zero if borders are turned off; non-zero if borders are enabled for windows.

## break;

### Description

The break statement is used to terminate switch, while, do and for loops.

### Return value

Nothing.

### See also

for(pg. 93)., case(pg. 53)., continue(pg. 60)., switch(pg. 204).

## (breaksw)

### Description

The breaksw primitive is internal to the lisp-like macro language only. It is generated by the crunch compiler when a 'break' keyword is encountered inside a switch statement.

### Return value

Nothing.

### See also

break(pg. 52).

## call_registered_macro(n)

### Description

This macro is used to trigger a registered macro. n is an integer expression specifying the registered macro to trigger.

See *register_macro()* for further details on the different registered macro types.

### Return value

Nothing.

### See also

register_macro()(pg. 163)., unregister_macro()(pg. 211).

## car(list_expr)

### Description

This macro is used to make a copy of the first atom in the expression list_expr.

Since lists may contain any data type, it is probably wise to assign the result of a car operation to a polymorphic variable, so that its type can be ascertained.

### Example

The following example prints a message saying what type the variable at the head of a list is:

```
                list    llist;
                message("Type is %s.", typeof(car(llist)));
```

### Return value

Returns the value of the first element in the list_expr.

### See also

cdr()(pg. 54)., delete_nth()(pg. 71)., nth()(pg. 149)., put_nth()(pg. 156).

## case <expr>:

### Description

The case statement is used inside a **switch** statement to select from number of values. The **expr** can be any expression, e.g. string or floating point.

### Return value

Nothing.

### See also

for(pg. 93)., break(pg. 52)., continue(pg. 60)., switch(pg. 204).

## cd([path]) or chdir([path])

### Description

This macro changes the current working directory. If path is omitted, then the current path is displayed on the status line.

If the command is successful then the REG_CHANGE_DIRECTORY trigger is executed allowing macros which want to know about changes to the current working directory to be informed.

**cd** and **chdir** are identical. The main difference is that **cd** tends to be overloaded by the CRiSP macros whereas **chdir** is not.

### Example

Display current path on screen:

```
        cd();
```

Change to users home directory:

```
        cd(inq_environment("HOME"));
```

### Return value

0 if successful; non-zero if path does not exist.

### See also

getwd()(pg. 99)., is_directory()(pg. 132).

## cdr(list_expr)

### Description

The *cdr()* macro is used to create a new list out of an existing list by removing the first atom in the list.

Using *car()* and *cdr()*, macros can be written which manipulate all elements on a list. However, it is more efficient to use *nth()* to extract individual elements, since internally it avoids having to copy sub-lists.

### Return value

Returns the list formed by deleting the first element of the list, list_expr.

### See also

car()(pg. 53)., delete_nth()(pg. 71)., nth()(pg. 149).

## ceil(x)

### Return value

Returns smallest integer not larger than x. Return value is a floating point number.

### See also

floor()(pg. 92).

## cftime(string fmt, int timeval)

### Description

This function can be used to generate a formatted time string. **fmt** is the format specification and **timeval** is a time in seconds since the epoch:

This function is implemented on top of the systems C function **cftime()** so refer to your systems manual page for documentation on the formatting strings available.

### Return value

String representing time in the format provided.

### See also

ctime()(pg. 66)., date()(pg. 66)., stat()(pg. 201)., time()(pg. 206).

## change_object(obj_id, attr_list...)

### Description

The *change_object* primitive is used to change the attributes of a GUI object (a user defined dialog box). The first parameter refers to the object which is being modified. The remaining arguments are a set of attribute/values which are to be modified. The argument list is similar to the XView programming toolkits *xv_set()* function.

All attributes are specified by a keyword indicating the attribute to be modified followed by the sub-object ID, followed by one or more expressions forming the value to be changed. A sub-object corresponds to an item in the dialog box; sub-objects are numbered from zero and are assigned increasing numbers within the dialog box for each item created. E.g. a dialog box with two inputs fields would have the first one with a sub-object ID of zero and the second one of 1.

### Return value

-1 if primitive is not supported.

-2 if *obj_id* does not refer to a valid object.

-3 if any of the attribute values are the wrong type or the specified sub-object does not exist.

1 on a successful attribute change.

### See also

dialog_box()(pg. 72)., delete_object()(pg. 71)., object_value()(pg. 149).

## change_window([direction], [msg])

### Description

This macro is used to move to another window. direction specifies in which direction to select the window. If it is omitted, then the user is prompted to select the window, by using the cursor keys.

If direction is specified, it should have one of the following values:

| | |
|---|---|
| 0 | Up |
| 1 | Right |
| 2 | Down |
| 3 | Left. |

| | |
|---|---|
| 4 | Next window |
| 5 | Previous window |

The *msg* parameter allows the calling macro to customise the message appearing on the status line. If not omitted, the string "**Point to destination**" is used.

### Return value

Returns 0 if unsuccessful; 1 if successful.

## change_window_pos([x], [y], [w], [h])

### Description

This primitive can be used to change the current window co-ordinates or size. Each argument is an integer value which is optional. If not specified then the existing value will be preserved.

### Return value

Nothing

### See also

create_window()(pg. 65)., create_tiled_window()(pg. 65).

## chmod(mode, file)

### Description

This primitive is used to change the protection and access information associated with a file. *mode* is the set of protection bits as defined by your operating system, and *file* is the name of the file or directory which should be changed.

### Return value

Returns 0 on success, -1 on failure. On a failure the global value *errno* is set to the reason for the failure.

### See also

chown()(pg. 55).

## chown(string path, int owner, int group)

### Description

The *chown()* function is used to change the ownership of a file. *path* is the name of the file and *owner* and *group* are the permissions bits.

This function is operating system dependent and the meanings of the permissions bits is machine dependent.

### Return value

0 on success; -1 on error and *errno* set to the error value.  -2 if system does not support the system call.

### See also

chmod()(pg. 55).

## chroot(string dirname)

### Description

Change the definition of "/" for the current process and all child processes -- used to restrict file system accessibility.

Whether this function succeeds or not is machine dependent and almost certainly requires system privileges to work.

### Return value

0 on success; -1 on error and *errno* set to the error value.  -2 if system does not support the system call.

### See also

cd()(pg. 53)., getwd()(pg. 99).

## clear_buffer([int buf_id])

### Description

Deletes the text from the specified buffer. If **buf_id** is not specified, then defaults to the current buffer.

### Return value

-1 on error. 0 on success.

### See also

delete_block()(pg. 69).

## clipboard_available()

### Description

This function can be used to determine if there is something in the clipboard for pasting. It is used by the paste user macros (icon, menu, keyboard short cut). It returns an indication indicating if data is available. If it is, then the paste scrap function will use this data, else it will use the data in the internal scrap created, for example by a **copy** macro call.

### Return value

1 if data is available in the clipboard. 0 if no data is available.

Returns -1 if it has not been determined if any data is available. This can happen in X Windows. In X Windows, determining if any data is available is an asynchronous operation. CRiSP normally checks whenever it receives the input focus but it can take as long as a few seconds to determine this if the network is busy, for example. In this circumstance, -1 indicates a *don't know* result.

### See also

clipboard_owner()(pg. 56)., get_selection()(pg. 96)., put_selection()(pg. 156).

## clipboard_owner()

### Description

This function can be used to determine if CRiSP currently owns the clipboard.

### Return value

1 if CRiSP owns the current selection; 0 otherwise.

### See also

clipboard_available()(pg. 56)., get_selection()(pg. 96)., put_selection()(pg. 156).

## close(fd)

### Description

This primitive can be used to close a file previously opened with open().

### Return value

Zero is returned on success, -1 on failure.

### See also

open()(pg. 149)., write()(pg. 215).

## color([background], [normal], [region], [messages], [errors], [hilite],

## [hi_foreground])

### Description

NOTE:: This primitive is provided for compatibility reasons only. It is no longer a builtin primitive but is implemented using the more sophisticated set_color() and get_color() primitives.

This macro is used to control the colors that CRiSP uses on a color display. (The values are ignored for monochrome displays).

If no arguments are specified then the argument values are prompted for.

*background* is the color for the background of normal text. *normal* is the color associated with text in the editing windows. *region* is the color of text when it is marked in a region. *messages* is the color for normal messages on the status line. *errors* is the colors for error messages on the status line. *hilite* is used to indicate the relationship between highlighted text and its background.

Colors may be specified as integers or strings. Strings may be specified in upper or lower case. The *hilite* parameter is to specify the foreground and background color of selected text. It may be a number or a string. If it is a number then the top nibble specifies the background color, and the low nibble the foreground color. If it is a string then it only specifies the background color. If *hi_foreground* is specified it specifies the foreground color of the highlighted region.

| **Color no** | **Color name** |
|---|---|
| 0 | Black |
| 1 | Blue |
| 2 | Green |
| 3 | Cyan |
| 4 | Red |
| 5 | Magenta |
| 6 | Brown |
| 7 | White |
| 8 | Dark-Grey |
| 9 | Light-Blue |
| 10 | Light-Green |
| 11 | Light-Cyan |
| 12 | Light-Red |
| 13 | Light-Magenta |
| 14 | Light-Yellow |
| 15 | Light-White |

### Return value

Nothing.

### See also

get_color()(pg. 93)., set_color()(pg. 179)., colorizer()(pg. 58)., set_keyword_index()(pg. 182).

## colorizer(NULL, ext, flags, keywd-list, [sym-regexp])

### Description

This primitive is used to configure and enable the CRiSP colorisation mechanism.  This function has been superseded by the **load_keywords()** primitive.

### Return value

The previous value of the colorisation flag.

### See also

color()(pg. 57)., colorizer_get_info()(pg. 58)., get_color()(pg. 93)., load_keywords()(pg. 139)., set_color()(pg. 179)., set_buffer_flags()(pg. 176)., set_keyword_index()(pg. 182).

## colorizer_get_info(int cmd, [int buf_id], [int line], [int col])

### Description

This primitive is used to get the parse state of a buffer which is being colorized. For example, this could be used to determine if the cursor is inside a comment or not.

| | |
|---|---|
| **bmd** | Operation to perform. 0 means retrieve parse state. |
| **buf_id** | Buffer to which the operation is applied, or the current buffer if this is omitted. |
| **line, col** | Line and column where the parse info is to be returned. If omitted, defaults to current line and column. |

### Return value

Parse state according to the supplied arguments. -1 on an error.

### See also

color()(pg. 57)., get_color()(pg. 93)., load_keywords()(pg. 139)., set_color()(pg. 179)., set_buffer_flags()(pg. 176)., set_keyword_index()(pg. 182).

## command_list([commands-only], [regexp])

### Description

This macro is used to get a list of all the keywords in CRiSP. Its primary purpose is to support the help macro package, and avoids hardcoding the macro primitives into a macro or text file.

If *commands-only* is omitted then a list of all builtin primitives and macros are returned. If *commands-only* is specified and is non-zero then only a list of builtin commands is returned.

If *regexp* is specified then only macros or primitives matching this regular expression will be returned. The syntax of the regular expression is that of the shell, viz. '*', '[..]' and '?' for wild-card characters.

### See also

macro_list()(pg. 141).

### Return value

Returns a list consisting of a sequence of strings which are the names of all the CRiSP macro primitives and all currently defined macro names.

## compare_files([int flags], string file1, string file2)

### Description

Use this primitive to perform a file comparison. It can be used to quickly determine if two files are identical (byte for byte). In contrast, the diff_buffers()(pg. 74). primitive is used to compare files and mark up the differences.

### Return value

Returns 1 if the files are identical, 0 if they differ.

-1 if the first file cannot be read, -2 if the second file cannot be read.

### See also

diff_buffers()(pg. 74).

## compress(string, [trim])

### Description

This macro takes a string and removes all multiple white space characters (spaces, tabs and newlines). If *trim* is specified then *compress()* acts the same as: trim(ltrim(compress(str))).

### Return value

Returns a copy of string with all spaces, tabs and newlines mapped to single spaces.

### See also

trim()(pg. 210)., ltrim()(pg. 141).

## connect([ipc_id], [mode], [shell], [arg_list ...])

### Description

This primitive is used to set up interprocess communication mechanisms. The actual communication mechanism used depends on various factors. Primarily this primitive is used to create a process buffer whereby output from a process is viewable in a buffer and can be interacted with by keyboard input.

The *ipc_id* parameter should be specified when you wish to change flags associated with a specific IPC channel which has already been created.

mode specifies some flags which are defined below. By default, the process created is a shell process, and the shell is got from the SHELL environment variable. If shell is specified, then it is taken as the pathname of a shell to execute.

*arg_list* is a list of zero or more string or list expressions containing command line arguments to be passed to the *shell* command. (Note that the *shell* parameter need not refer to a shell program -- it could be any executable binary program). Specify a list (e.g. in a variables) allows a macro to be written which takes a variable number of arguments for passing to the connect() macro.

Output from a sub-process is automatically inserted into the buffer, at a position known as the process position (see *inq_process_position()*, *set_process_position()*). This position is analogous to the usual input cursor. (This automatic insertion can be overridden by using a process input trigger. Refer to the *register_macro* primitive for further details).

Any text which would normally be inserted into the buffer via *self_insert()*, *insert()* or *paste()* is also forwarded to the sub-process.

CRiSP implements sub-processes out of pipes or pty's depending on the availability of the underlying Unix implementation.

mode is a set of flags which have the following meaning:

| **Value** | **Description** |
|---|---|
| 0x01 | If this bit is set, then the buffer operates in echo mode, i.e. the characters which are typed are inserted into the buffer. If this bit is not set, then the buffer operates in no-echo mode, and the characters the user types are not directly inserted into the buffer, but instead only the output from the sub-process. |
| 0x02 | Reserved. |
| 0x04 | Don't invoke shell with the '-i' switch (for interactive mode). |
| 0x4000 | This bit indicates whether output from a process overwrites text in the buffer or inserts and shifts text over as it does so. This is needed to effectively allow typeahead to not be destroyed but allow escape sequences to cause data in the buffer to be overwritten when running termcap oriented programs. |

| | |
|---|---|
| 0x8000 | This is the wait mode of operation. Normally when a buffer is created, the output from the subprocess is inserted directly into the buffer. Setting this bit causes the output from the process to be held onto, until the calling macro issues a wait() or waitfor() macro call. |

connect() may be called after a buffer has been connected to change the mode flag.

### Return value

Returns 0 if an error occurs. Otherwise the IPC identifier associated with this connection is returned (a non-zero value).

### See also

disconnect()(pg. 75)., insert_process()(pg. 127)., register_macro()(pg. 163).

## continue;

### Description

This macro is used to finish the current iteration of a *while()* loop, and go back to the top of the loop, causing the while condition to be re-evaluated.

This works similar to the C continue statement.

### Return value

Nothing.

### See also

break(pg. 52).,for(pg. 93)., switch(pg. 204)., while(pg. 215).

## convert_white_space(string, [flags])

### Description

This macro is used to perform tab/space conversions.

**string** is the input string.

**flags specifies the manner of conversion. The following bit definitions apply:**

| | |
|---|---|
| **0x01** | **If specified then spaces are converted to tabs.** |
| **0x02** | **If specified then spaces/tabs inside single or double quotes are not converted.** |

**When tab conversion is performed, the size of the tab stops are governed by the current buffers tab stop size.**

### Return value

Converted string.

### See also

break(pg. 52).,for(pg. 93)., switch(pg. 204)., while(pg. 215).

## copy([append], [leave_hilite])

### Description

copy() copies the currently marked region to the scrap buffer.

If **leave_hilite** is not specified or is zero, then the currently selected area will be unhighlighted

If **append** is specified and is non-zero, then the text is appended to the current scrap without deleting it first.

Copying works for all highlighted region types.

### Return value

Nothing.

### See also

cut()(pg. 66)., get_region()(pg. 96)., paste()(pg. 150).

## copy_keyboard(src_id, [cmd1 ...])

### Description

This macro is used to copy key assignments from an alternate keyboard id. cmd1, ..., are strings containing macro names. These macro names are looked up in the source keyboard and if found are assigned to the same keystrokes used in the source keyboard. If no commands are specified, then the entire keyboard is copied.

This macro is very useful for popup displays because it allows the macro writer to ensure that if a user has tailored the keyboard layout that the popup will use the same keys at the main editing level, without having to hard-code constants all over the place.

### Return value

Returns current keyboard id on success. If source keyboard not found then returns zero.

### See also

assign_to_key()(pg. 47)., inq_keyboard()(pg. 112).

## cos(x)

### Return value

Cosine of x (in radians) as a floating point value.

### See also

sin()(pg. 196)., tan()(pg. 206)., acos()(pg. 46)., asin()(pg. 46)., atan()(pg. 46).

## cosh(x)

### Return value

Hyperbolic-cosine of x as a floating point value.

### See also

sinh()(pg. 196)., tanh()(pg. 206).

## create_blob(win_id, line_no, col_no, name, bitmap, macro, fg, bg, flags)

### Return value

This primitive is used to create a blob. A blob is an object which is overlaid over the character mode editing window, e.g. to represent a cursor position using an arrow..

### See also

delete_blob()(pg. 69).

## create_buffer(buffer-name, [file-name], [system], [edit_flags])

### Description

This macro is used to create a new buffer. The buffer is given the name "buffer-name" which may be any sequence of ASCII characters. This buffer name is used to label the top of any windows the buffer is attached to (see attach_buffer). Care should be chosen in selecting buffer names, since the top title on any displayed window may be truncated if the window is not wide enough to handle the title.

"file-name" is the full or relative path name of a file which should be read into the buffer. If this parameter is omitted, then an initially empty buffer will be created. The "file-name" read into the buffer is saved for later use, if the buffer is saved (via write_buffer).

[system] is an optional integer which indicates whether the buffer is to be a system buffer or user buffer. If it is 0 or omitted, then the buffer is a user buffer; a non-zero value indicates a system buffer. System buffers are like normal buffers, except for two side-effects - no *undo()* information is saved for operations on system buffers, and most high-level macros tend to ignore system buffers, e.g. the *buffer_list()* macro does not usually display system buffers.

*edit_flags* is used to control how the specified file will be read in to the buffer. These flags are used to force CRiSP to read the file in ascii, binary or *normal* modes. Refer to the *edit_file()* primitive for information describing this field.

System buffers are for use by user macros or CRiSP supplied macros and provide a way of storing information without getting in the way of the user. *undo()* information is not saved for these buffers, and thus means that operations on system buffers are slightly faster than normal buffers. It is not important that *undo()* information is not stored with these buffers, because these buffers tend to only be manipulated by the various macros - not directly by the user.

### Example

The following example shows how to create a buffer and display it in the current window. *edit_file()* may be more appropriate for this particular task, but the example is illustrative.

```
int     newbuf;

newbuf = create_buffer("My Buffer", "myfile.txt");
attach_buffer(newbuf);

refresh();
```

### Return value

Returns the buffer identifier associated with the newly created buffer.

### See also

create_nested_buffer()(pg. 64)., edit_file()(pg. 81)., inq_buffer()(pg. 105).

## create_char_map([cmap_id], [start_ch], ch_list, [flags_list])

### Description

This primitive creates a character map. A character map is a mapping between the characters stored in the buffer and the way they are printed on the screen. Character maps have two main uses - to configure how control characters and characters with the top bit are printed, or to view characters in a different format, e.g. hex.

If cmap_id is not specified, then a new character map is created. If cmap_id is specified it should be an integer value corresponding to a previously created character map (or zero to change the default one).

Character maps are a mapping for all 256 possible byte values. The ch_list parameter is a list giving up to 256 value for these sequences. If you only want to remap part of the displayable character set, then specify start_ch to indicate where you are starting.

ch_list may have up to 257 values. The 257th entry is used to indicate how the end of a line should be marked. By default, the end of line is marked with a space (i.e. not distinguishable). The literal macro uses this character to specify a '$' to mark end of lines.

CRiSP understands that certain displayable characters need special processing. These characters are specified by the optional flags_list parameter. The flags list is specified as a pairing of values. The first element in the pair is an integer defining the character value, and the second entry in the pair indicates the action to be performed. The following actions are defined:

| Value | Meaning |
|---|---|
| 0 | No special processing. |
| 1 | Character is a tab. If this is used, the first character of the string is used in the TAB expansion. Set this to a space for normal invisible tabs. Setting it to a |

| | |
|---|---|
| | dot would make the first character of a tab visible as a dot. |
| 2 | Character is a backspace. |
| 3 | Character is an ESCAPE. |

Backspace and ESCAPE flags are needed if you want the ANSI processing mode to work.

For examples of this primitive, consult the view.cr macro.

### Return value

Returns the character map ID, or -1 if the specified map does not exist.

### See also

inq_char_map()(pg. 108)., set_buffer_cmap()(pg. 176)., set_buffer_flags()(pg. 176)., set_window_cmap()(pg. 190).

## create_dictionary()

### Description

This primitive creates a user-defined dictionary. A dictionary is a private symbol table in which you can create arbitrary symbols. A dictionary is a bit like an associative table. You can use the *get_property()* or *set_property()* primitives to get and set values in the dictionary.

create_dictionary() returns a dictionary id. When you have finished with a dictionary you should delete it using *delete_dictionary*().

### Return value

Returns an object id corresponding to the new dictionary id.

### See also

delete_dictionary()(pg. 70)., dict_exists()(pg. 73)., dict_list()(pg. 74)., get_property()(pg. 95)., remove_property()(pg. 168)., set_property()(pg. 185).

## create_edge([direction])

### Description

This macro is used to split the current window. The window is split in the direction specified. If direction is not specified then the user is prompted, and can use the arrow keys to indicate the direction of the split.

If direction is specified, it should have one of the following values:

| | |
|---|---|
| 0 | Up |
| 1 | Right |
| 2 | Down |
| 3 | Left. |

### Return value

Returns 0 if unsuccessful; 1 if successful.

### See also

create_window()(pg. 65)., delete_window()(pg. 72)., move_edge()(pg. 147).

## create_nested_buffer(buffer-name, [file-name], [system], [edit_flags])

### Description

This function is similar to the **create_buffer()** primitive but is provided for convenience. If you are writing macro code which needs to visit private files you have to be careful that the user is not editing the file at the time you attempt to create a buffer. Also when you delete your private buffer you need to be careful not to delete it if the user is currently editing the buffer.

This function works identically to create_buffer() but if the buffer is already loaded, then no new buffer is created. Instead a reference counter associated with the buffer is incremented. You can safely call delete_buffer() to undo the effect of create_nested_buffer() but the buffer will not actually be deleted until the last reference to the buffer is destroyed.

This makes it much more convenient when you need temporary access to a file within a macro.

### Return value

Returns the buffer identifier associated with the newly created buffer or the existing buffer id if the file is already loaded..

### See also

create_buffer()(pg. 62)., edit_file()(pg. 81)., inq_buffer()(pg. 105).

## create_notice(msg, button1, [button2], ...)

### Description

This function is available only in the GUI versions of CRiSP.

This primitive is used to display a notice dialog box. The dialog box contains the message *msg* (a string containing multiple lines of text separated by newlines). One, two or three buttons can be displayed in the dialog box, labelled with the text specified by the *buttonN* parameters.

This primitives halts execution of CRiSP until the user dismisses the dialog box (e.g. by clicking on one of the buttons).

### Return value

This primitive returns the number of the button pressed (0, 1, or 2). If the primitive is not supported (e.g. in the character mode versions of CRiSP), then a value of -1 is returned.

## create_object(list)

### Description

The create_object() primitive is used for creating dialog box and GUI based display objects. The *list* parameter is a list of indefinite parameters which allows the calling macro to construct the list of objects and attributes for the items in the dialog box. Any macro using this primitive will need to #include the file <gui.h> which defines a whole bunch of macros starting with the prefix "DBOX_". You are strongly advised to scan through this file.

A dialog box is made up of objects (widgets in X11 terminology or Controls in Microsoft Windows terminology). The definitions in **<gui.h>** define the set of widgets CRiSPs supports. (Not all of the definitions are valid and some are reserved for the future). Generally speaking, object types are **#define**'d as a small integer starting from 1 onwards. Object attributes are numbered from 0x1000 onwards. Object attributes apply to the immediately preceding object definition.

The objects and attributes supported are designed to enabled CRiSP's user interface to be written. The library of functionality available is not necessarily designed to be all singing and dancing (e.g. such as Motif or XView). This would make CRiSP very much larger than it currently whilst adding only limited functionality).

### Return value

If called with no values then this primitive returns -1 if the primitive is not supported (e.g. it is the character mode version of CRiSP), or a value >= 0 if the primitive is available for use.

Otherwise an object identifier is returned which can be used to refer to the dialog box object.

### See also

dialog_box()(pg. 72)., change_object()(pg. 54)., inq_objects()(pg. 118).

## create_tiled_window(lx, by, rx, ty, [buf])

### Description

This macro is used to create a tiled window (as opposed to a popup window). Normally tiled windows (background windows) are created by the user splitting the current window. This macro is provided to allow the implementation of the state restore macro so that the window layout can be saved and restored on restarting CRiSP.

The (lx,by,rx,ty) arguments are the co-ordinates of the window, irrespective of whether borders are enabled or not, with (0,0) being the top left hand corner of the screen.

buf is an optional integer containing a buffer ID to attach to the created window.

It is probably best to disable the screen updating whilst using this macro, otherwise the screen may become inconsistent. (Refer to the display_windows() macro).

### Return value

Returns the window id of the created window.

### See also

create_tiled_window()(pg. 65)., delete_window()(pg. 72)., inq_window()(pg. 123).

## create_window(left_x, bottom_y, right_x, top_y, [message])

### Description

This macro is used to create a new window. left_x, bottom_y, right_x and top_y specifies the co-ordinates of the window.

message is an optional string expression, which if specified is a message to appear centered on the bottom line of the window.

Be careful not to create windows which do not fit within the physical screen, otherwise CRiSP may crash.

The new window created becomes the current window - its id can be inquired via *inq_window()*.

### Return value

Returns the window ID of the new window.

### See also

create_tiled_window()(pg. 65)., delete_window()(pg. 72)., inq_window()(pg. 123).

## crisp_instance(cmd, [args])

### Description

This function is used  to implement the CRiSP single-instancing feature under Windows. Instancing is a mechanism which allows you to use one copy of CRiSP and when you attempt to invoke a new copy, to send a message to the currently running copy to load files as specified from the command line. There are some similarities to the CRiSP server and other IPC mechanisms, but this is designed specifically for the Windows platform.

| | |
|---|---|
| **cmd** | == 0 then the function returns 1 if another CRiSP instance is running. Returns zero otherwise.<br>== 1 Send the argument **args** (which is a string command line) to the currently running instance of CRiSP. The calling application should probably then exit.<br>== 2 Declare current instance as the master copy of CRiSP. Should normally be called only if no other instance is running.<br>== 3 Get state of command line **-multi_inst** or **-single_inst** command line flag. If non-zero, then multiple instances are being requested. |
| **args** | String argument to pass to existing CRiSP instance if **cmd** is set to 1. |

### Availability

Win32 CRISP.EXE only. Not available in CR.EXE.

## ctime(timeval)

### Description

This function can be used to take a time in seconds since the epoch (e.g. as returned by the stat() call) and return an ASCII string in the standard ANSI-C notation. The typical format returned is:

```
Fri Jun 11 14:54:07 GMT 1993
```

### Return value

String representing time in the format above.

### See also

cftime()(pg. 54)., date()(pg. 67)., stat()(pg. 201)., time()(pg. 206).

## cut([append], [save_pos])

### Description

The currently selected region is copied to the scrap buffer and the region deleted from the current buffer.

If **append** is specified and is non-zero, then the selected region is appended to the end of the scrap buffer, without first clearing it.

If **save_pos** is specified and is non-zero, then the current cursor position is preserved in the buffer. Use this in conjunction with end_anchor() to create disjoint marked regions (i.e. the cursor can move outside of the marked block).

### Return value

Returns zero on success; -1 if an error occurs or a region is not currently highlighted.

### See also

end_anchor()(pg. 82)., get_region()(pg. 96)., copy()(pg. 61)., paste()(pg. 150)., delete_block()(pg. 69).

## string cvt_hsv_to_rgb(string hsv-value)

### Description

This function converts a string representing a HSV (Hue, Saturation and Brightness) color value to its equivalent RGB value. The string represents a color in the format: "#HHSSVV" where HH, SS and VV are two digit hexadecimal color values. The color format representation is designed to allowing easy passing to and from the DBOX_COLOR_SELECTOR display widget.

### Return value

An RGB color string in the format: "#RRGGBB".

### See also

cvt_rgb_to_hsv()(pg. 67).

## string cvt_rgb_to_hsv(string hsv-value)

### Description

This function converts a string representing an RGB (Red, Green, Blue) color value to its equivalent value expressed in the HSV (Hue, Saturation, Brightness) format. The string represents a color in the format: "#RRGGBB" where RR, GG and BB are two digit hexadecimal color values. The color format representation is designed to allowing easy passing to and from the DBOX_COLOR_SELECTOR display widget.

### Return value

An RGB color string in the format: "#HHSSVV".

### See also

cvt_hsv_to_rgb()(pg. 66).

## cvt_to_object(val, [len])

### Description

cvt_to_object may be used to parse a string and decode the object contained in it, e.g. a number, or a float. It is the converse to sprintf(), and may be used to parse a string which may be input by the user or created by a program for decoding. The 'val' parameter should be a string which contains the object to be decoded.

*len* is an optional integer variable which will receive the number of characters used in the parsing. This may be useful if the string contains multiple items which will need parsing.

### Return value

Value of 'val'. The value is an integer, or float or string depending on the syntax of the item in the expression.

If the item cannot be parsed, then the **NULL** value will be returned.

## date([year], [mon], [day], [month-name], [day-name])

### Description

This macro retrieves the current date. year, mon and day are optional integer variable names which receive the current year, month number, and day of month number. Year is of the form 1988; mon is in the range 1-12; day is in the range 1-31.

month-name is an optional string variable, and contains the name of the month, in the format: January; day-name is an optional string variable containing the day of the week, in the form: Saturday.

### Return value

Nothing.

### See also

ctime()(pg. 66)., time()(pg. 206).

### Example

The following example can be used to print an American date string:

```
int     day, mon, year;
date(year, mon, day);
message("%02d/%02d/%02d", mon, day, year - 1900);
```

## debug([n], [time])

### Description

*debug* turns on or off the macro trace mode. When debug is on, CRiSP traces the execution of every macro statement, and in addition traces assignments to the accumulators (integer, string and list).

This is the primary means of debugging macros.

If n is omitted, then debugging is toggled and a message is printed on the status line saying whether debug is on or off.

By default, debug toggles between 0 and 1. By specifying a value for the debug macro, other events in CRiSP can be traced. The other bits are mostly for debugging CRiSP itself, and include the ability to trace the following:

If the first argument is a macro name then debugging will be turned on automatically when that macro is executed. This provides a convenient mechanism for turning on debugging only when a deeply nested macro is executed.

If **time** is specified and is non-zero, then the crisp.log file will have a time stamp added at the beginning of each line of output. This is mainly used for diagnosing end-user problems where it is important to see the real-time sequence of events which they are performing.

| Flag | Meaning |
|---|---|
| 0x01 | Trace macro execution. |
| 0x02 | Trace regular expression parsing. This is used to check that regular expressions are being compiled and executed correctly. |
| 0x04 | Trace buffer *undo()* information. Use this to check that a primitive is recording the correct information to undo it. |
| 0x08 | Cause log file to be flushed each time it is written to. This is useful when CRiSP core dumps and you want to see the last primitive executed. This significantly slows down operation. |
| 0x10 | Disable SIGBUS and SIGSEGV processing. Useful when debugging and you want core dumps but could cause problems. (Implemented because under XView you can hang the server). |
| 0x4000 | Turns on tracing for the prompt history code. This traces execution of the command  line macros. |

When debug is turned on, all debug output is put in the file **/tmp/crisp.log**. The name of this file can be overridden by specifying the environment variable **CRiSP_LOG**.

If the user is trying to debug a macro which causes CRiSP to core dump, then CRiSP should be run with the -f flag. This causes all debug info to be flushed to the log file as it is produced, rather than using the stdio buffering. By default this is off, since it significantly slows CRiSP down due to the high volume of output.

### Return value

When setting a numeric debug value the old debug value is returned. When setting debugging on a macro, -1 is returned if the macro is not found. 0 otherwise.

### See also

inq_debug()(pg. 109).

## declare var1, var2, ..;

### Description

This is used to define one or more polymorphic variables. Polymorphic variables are variables that may take on any of the available CRiSP data types (integer, string or list), depending on context.

Polymorphic variables are most useful when dealing with lists which contain unknown data types.

The variables declared are made into local variables, unless a *global()* declaration follows.

By default, polymorphic variables are typed as integers, and are given the value zero.

## del(filename)

### Description

THIS FUNCTION IS PROVIDED FOR COMPATIBILITY ONLY via the brief.cr MACRO FILE.

This macro is used to delete a file, as specified by the string expression 'filename'.

Wild-cards, etc, are not valid in 'filename'.

### Return value

Zero or less if unsuccessful; greater than zero means the file was successfully deleted.

### See also

remove()(pg. 168).

## delete_blob(blob, [buf])

### Description

This primitive is used to delete a blob associated with the current buffer, or the specified buffer. (A blob is either a character based overlay onto a buffer, e.g. a debuggers breakpoint arrow, or a GUI clickable icon, e.g. a post-it note).

The id of the blob, as previously returned from a *create_blob()* function is passed as the first argument.

### Return value

Zero on success; -1 if specified buffer does not exist; -2 if specified  blob does not exist.

### See also

create_blob()(pg. 61).

## delete_block()

### Description

The currently highlighted block is deleted.

### Return value

Returns 0 on success, -1 on failure.

## delete_bookmark(bookmark)

### Description

This macro is used to delete a bookmark. **bookmark** is a string, containing the name of the bookmark to delete.

### Return value

Nothing.

### See also

bookmark_list()(pg. 52)., drop_bookmark()(pg. 79)., goto_bookmark()(pg. 100).

## delete_buffer(bufnum)

### Description

This command deletes the buffer specified by bufnum. The entire buffer contents are freed, and any other resources attached to that buffer are destroyed as well. In the case of a process buffer, the subprocess is killed.

Any attempt to *set_buffer()* to a buffer which has been deleted will fail.

Care should be taken with *delete_buffer()* since no tests are made to see if the buffer being deleted is currently displayed. If a currently displayed buffer is deleted, then CRiSP may core-dump. Macros using *delete_buffer()* should be carefully tested to ensure this does not happen, e.g. by calling *inq_views()*.

### Return value

No value is returned.

### Example

The following example creates a temporary buffer and then deletes it after restoring the current buffer at the time of the macro is invoked.

```
int     newbuf;
int     curbuf;

curbuf = inq_buffer();
newbuf = create_buffer("Example", NULL, 1);
```

```
set_buffer(newbuf);
set_buffer(curbuf);
delete_buffer(newbuf);
```

## delete_char([num])

### Description

This macro deletes one or more characters from the current buffer. If num is not specified, then 1 character is deleted; if num is specified then that number of characters are deleted. The deleted characters are on and to the right of the current buffer position.

This macro is the default assignment for the <Delete> key on the keyboard.

## delete_dictionary(dict_id)

### Description

This primitive is used to free up a previously created dictionary. *dict_id* is an object identifier previously created via *create_dictionary()*.

### Return value

Zero on success, or -1 if the specified *dict_id* does not exist.

### See also

create_dictionary()(pg. 63)., get_property()(pg. 95)., remove_property()(pg. 168)., set_property()(pg. 185).

## delete_edge([direction])

### Description

This macro can be used to delete a window on-screen. It is used by specifying an edge, and if there is another window which adjoins the edge, then the edge is deleted and the two windows combined into one.

direction specifies the edge to delete and if not specified, is prompted for, and the user can use the arrow keys to specify the edge.

When using this macro, the edge to be deleted must not be obscured by other edges, i.e. only tiled window support is allowed for this macro.

If direction is specified, it should have one of the following values:

| | |
|---|---|
| 0 | Up |
| 1 | Right |
| 2 | Down |
| 3 | Left |

### Return value

Returns <= 0 if unsuccessful; >0 otherwise.

## delete_keystroke_macro(id)

### Description

This primitive is used to delete a previously defined keystroke macro.

### Return value

Zero on success, or -1 if the specified macro does not exist.

### See also

inq_keystroke_macro()(pg. 113)., load_keystroke_macro()(pg. 139).

## delete_line([num_lines])

### Description

This macro deletes the line the cursor is currently on. The cursor is placed at the same column position in the line below the one deleted.

If *num_lines* is specified then that number of lines is deleted. If *num_lines* is not specified then one line is deleted.

### Example

The following example assigns the *delete_line()* macro to the <ALT-D> key (the usual case).

```
assign_to_key("<Alt-D>", "delete_line");
```

## delete_macro([file])

### Description

This macro is used to delete all macros which were loaded from a particular macro file. file is a string expression and if omitted is prompted for, containing the name of the file.

This is currently a no-op in CRiSP - macros are not deleteable entities. The storage allocated for a macro is lost when a macro by the same name is loaded from another file.

### Return value

Nothing.

## delete_nth(list, pos, [num])

### Description

This primitive is used to delete an element or elements from a list. *list* is the list and *pos* is the index into the list to start deleting.

*num* is how many consecutive elements to delete. If *num* is omitted then a single element will be deleted.

### Return value

Returns a new list with the elements deleted.

### See also

nth()(pg. 149)., put_nth()(pg. 156).

## delete_object(obj_id)

### Description

This macro is used to delete a dialog box.

### See also

create_object()(pg. 64)., insert_object()(pg. 127)., object_value()(pg. 149)., remove_object()(pg. 168).

## delete_to_eol()

### Description

This macro deletes all characters up to the end of the line in the current buffer.

### Return value

Nothing.

## delete_window([win_id])

### Description

This macro can be used to delete a window. If win_id is not specified then the current window is deleted.

### Return value

Nothing.

## detab_text(len)

### Description

*detab_text()* is used to strip out tabs from the current line. The tabs are replaced by an equivalent number of spaces.

The principal use of this function is for converting columns of information into a canonical form without having to special case what happens when tabs are inside and over the borders of a column of text.

The *len* parameter specifies how many columns are affected. The affected text is from the current column up to the number of characters specified. If *len* is < 0, then the remainder of the line from the current column is converted.

### Return value

-1 on an error; 0 on success.

### See also

delete_char()(pg. 70)., read()(pg. 160)., tabs()(pg. 205).

## dialog_box(type, retvar, [title], ...)

### Description

This primitive provides support for certain built in dialog boxes. The dialog boxes support is only available in the GUI versions of CRiSP and as such the dialog boxes supported may vary from GUI version to GUI version. The principal idea of this function is to aid in supporting CRiSP on multiple platforms and this function is key in providing the hooks needed in the macro language to do all the visible things seen in the dialog boxes.

This function is designed to allow macro writers create custom dialog boxes accessing a large useful fraction of the underlying toolkit in a machine independent fashion. Dialog boxes can include common user objects, such as push-buttons, scrolling lists, and even CRiSP windows. The programming style vaguely resembles the old XView toolkit, e.g. the dialog_box() function is called with a list of attributes and values. An entire dialog box can be created with a single call to this function. Complex dialog boxes (e.g. the startup CRiSP window) can be built up by constructing a list containing a description of all the component objects.

The first parameter, *type* specifies the name of the common dialog box. Refer to the table below for the values which may be specified.

*retvar* is the name of a symbol to receive certain status information as described below. If this variable is omitted it is possible to check for the availability of the dialog box function without actually invoking it. This is used for example in the menu bar macros to decide whether to create a cascaded font menu or a dialog box for the font menu.

The *title* parameter is an optional string used for the Title of the dialog box. Each dialog box has an appropriate default title, which may be changed by specifying this parameter.

| type | Description |
|---|---|
| font | This pops up a font selection dialog box. retvar receives the name of the font selected. There is no guaranteed meaning to the font name returned, except that it may be passed verbatim to the set_font() primitive to effect the font change. (For example, under Microsoft Windows, no font naming convention is available. Under X11, the font name will be a |

| | |
|---|---|
| | font name, e.g. as returned by the xlsfonts(1) program). |
| open_file | This pops up a file-selection dialog box. The dialog box is popped up in such a way that the calling macro will be suspended until the dialog box is dismissed. When the user has finished selecting a file (either by accepting the file or cancelling the selection) the primitive will return. If the user selects a file, then the retvar parameter will contain the name of the file selected.<br><br>An optional third argument may be specified for use as the dialog box title. If this is not specified then it will default to "Open File".<br><br>If the user selects a file then a value of 1 will be returned; if the user cancels the selection then 0 will be returned. |

### Return value

Returns -1 if dialog box type not support. Returns 0 if the dialog box was cancelled. Returns 1 if dialog box operation was successful (e.g. user clicked on the OK button).

### See also

create_object()(pg. 64)., set_font()(pg. 181).

## dict_delete(obj_id, "name")

### Description

This primitive can be used to remove a symbol from a dictionary. *obj_id* is an integer representing a previously created dictionary or object, and *name* is the name of the symbol to remove.

### Return value

-1 if *obj_id* is invalid. 0 if the specified *name* does not exist and 1 if the symbol has been deleted.

### See also

create_dictionary()(pg. 63)., dict_exists()(pg. 73).

## dict_exists(obj_id, ["name"])

### Description

This primitive can be used to verify whether a dictionary exists or a symbol exists in a dictionary. *obj_id* is an integer representing a previously created dictionary or object, and *name* is the name to lookup.

If name is omitted then the function will return a value corresponding to the validity of the dictionary.

### Return value

If *name* is omitted, then -1 is returned if the specified object does not exist, or zero if the dictionary is no longer valid. Returns 1 if the dictionary is valid.

If name is specified, then -1 if *obj_id* is invalid. 0 if the specified *name* does not exist and 1 if the symbol does exist.

### See also

create_object()(pg. 64)., create_dictionary()(pg. 63)., dict_list()(pg. 74).

## dict_list(obj_id, [prefix])

### Description

This primitive is used to get the list of all symbols in the specified dictionary. *obj_id* is an integer representing an object previously created via *create_dictionary()* or *create_object()*.

If *prefix* is specified then only those symbols matching the specified prefix are returned.

### Return value

List of symbol names in a dictionary.

### See also

dict_exists()(pg. 73)., create_dictionary()(pg. 63)., create_object()(pg. 64).

## diff_buffers(flags, buf_id1, buf_id2, [string merge_file], [int new_buf1], [int new_buf2])

### Description

The diff_buffers() primitive is used to compare two buffers. CRiSP will mark lines which have been changed in both buffers, as well as new lines. (In order to see these marked lines you will need to enable the display of the change-bar margin by setting the BF_MOD_LINES in the buffer flags.

There are essentially three ways to use this primitive. You can use it to compare two buffers and determine if the buffers are the same; you can create a merged file of differences allowing you to manually manage the merging of differences. Or you can create one or two new buffers which contain difference lines which can convert one file to the other.

CRiSP will automatically clear the modified, new and deleted line flags associated with all lines in both buffers and the result of a successful diff will only have the different lines in the two buffers displayed (i.e. if you are displaying your own modified lines then these markers will be cleared).

The macro file **src/crunch/diff.cr** provides the user interface to this primitive, and shows good examples of how this primitive is designed to be used.

***flags*** is a set of flags which control how the diff is to be performed:

| Flag | Meaning |
|---|---|
| 0x01 | If set, then CRiSP will ignore trailing ^M characters. This is useful when comparing a DOS text file against a Unix text file. |
| 0x02 | If set, the diff will ignore trailing white space at the end of the line when doing the comparison. |
| 0x04 | CRiSP will compress multiple white spaces when doing the comparison. |
| 0x08 | All white space is stripped before comparing lines. |

***buf_id1*** and ***buf_id2*** are the buffer IDs of the two buffers to compare.

**merge_file** is the optional name of an output file which will be created showing the differences between the two files. The style of the output is designed to allow easy recognition of changed blocks or for automating the merging of the data.

**new_buf1** and **new_buf2** are two optional buffer identifiers. These buffers will be created as copies of the input buffer but with line markings to show where lines have been inserted, deleted or been modified. The normal line marking mechanism cannot show deleted lines in the input buffers, so use these arguments if you require this information.

### Return value

-1 if either buffer identifier is invalid; otherwise the number of different lines in buf_id1 is returned (zero if the two buffers are identical).

### See also

compare_files()(pg. 59)., set_buffer_flags()(pg. 176)., re_search()(pg. 158).

## dirname(path, [noexpand])

### Description

This function is similar to the Unix *dirname(1)* utility. Given a path string, it returns the directory part of the string.

If **noexpand** is specified and is non-zero, then the filename will not have wildcards expanded before returning the directory name.

### Example

`basename("/tmp/fred")` returns `"/tmp"`.

### Return value

String containing the directory of a file specification.

### See also

basename()(pg. 50).

## disconnect([ipc_id])

### Description

This macro is used to close an IPC link. The IPC link refers either to a process buffer (ipc_id not specified) or a previous IPC link created with the *connect()* macro. In the latter case the *ipc_id* parameter refers to the IPC connection previously returned by *connect()*.

For process oriented IPC links (e.g. process buffers and pipes) the subprocess is sent a SIGTERM followed by a SIGKILL signal.

### Return value

Returns 0 on success, or -1 if no IPC link connected to current buffer or *ipc_id* is invalid.

### See also

connect()(pg. 59)., send_signal()(pg. 174).

## display_enable([enable])

### Description

This macro is used to enable the display. By default when CRiSP starts up, the code for updating the screen is disabled. The startup macros initialise CRiSPs internal character translation tables and then call this macro to allow CRiSP to proceed.

If enable is not specified, then no action is taken. This is useful for simply inquiring the startup state.

If the display is not enabled, then CRiSP can be used as a normal programming language, in the style of awk and sed, using the command line arguments to load and execute a file.

If this macro is called as:

```
        display_enable();
```

then the current display status is returned without toggling the display state.

If **enable** is 1, then display is enabled.

### Return value

Previous value of the display flag.

## display_mode([and-mask], [or-mask], [shift-width])

### Description

This macro is used to read and optional set the display control flags. The display control flags are a set of attributes used to control the way certain things happen on the screen.

The *and-mask* and *or-mask* allow you to turn on or off bits.

*shift-width* is an integer which represents how much to sideways scroll the window when the cursor moves past the right hand edge of the window. The default of 0 means that the screen will scroll one character at a time. On slower systems this can be annoying and slow. Therefore it may be better to set it to a value between 2 and 8 to scroll more characters at a time thus reducing the amount of display traffic. If a negative value is specified for this field then the current shift-width setting will be returned.

| Bit | Name | Description |
|---|---|---|
| 0x0001 | DC_WINDOW | Window mode. This indicates we are running natively under a windowing system, e.g. X-Windows. This bit is read only. |
| 0x0002 | DC_SHADOW | When set, causes pop up windows to have a shadow (Default is set). This may be turned off for slow terminals. |
| 0x0004 | DC_SHADOW_SHOWTHRU | This indicates that the window shadow should show through the underlying text on the screen. |
| 0x0008 | DC_EOL_HILITE | This flag, when set, will cause highlighting to only extend to the end of the physical line and not necessarily the full width of the window. |
| 0x0010 | DC_CHAR_MODE | Bit set if DC_WINDOW is set but the application is really a character mode version. This applies to the Windows/NT CR.EXE program which supports some GUI aspects (e.g. the mouse) but not the fully fledged menu bars and scrollbars of the GUI version (CRiSP.EXE). |
| 0x0020 | DC_WRITES_DISABLED | RESERVED. |
| 0x0040 | DC_SUFFIX | If set then windows will have a modifier and/or read-only suffix appended to the window title. |
| 0x0080 | DC_MOD_FG | Show modified lines using the MODIFIED foreground color |
| 0x0100 | DC_MOD_BG | Show modified lines using the MODIFIED background color |
| 0x0200 | Reserved | |
| 0x0400 | Reserved | |
| 0x0800 | DC_EOF | Display [EOF] marker at the end of all windows to show the physical end of file. |

### Return value

Returns previous value of display control flags. If *shift_width* is specified and is less than zero, then the current shift_width value will be returned.

## display_windows([enable])

### Description

This macro is used to prepare for creation of a screen layout using the **create_tiled_window** primitive. This macro is provided for compatibility with the BRIEF editor.

If you want to create a new window layout, e.g. to restore a saved state, then you can use this macro before and after a series of **create_tiled_window** macro calls. Call it before creating the windows, with the **enable** argument set to zero, to disable updating of the display whilst the windows are being created. When the windows have been created, call this function again with **enable** set to 1.

Although this macro is provided for compatibility with BRIEF it is not mandatory as it is in BRIEF as CRiSP optimises display updates and window creations, so usage of this macro is optional.

### Return value

None

### See also

create_tiled_window()(pg. 65).

## dlerror()

### Description

This primitive returns a string explaining why the last load_library() primitive failed.

### Return value

String containing reason for last failure.

## distance_to_tab()

### Description

This macro returns the number of characters between the current cursor location and the next tab stop, even if there are no more characters after the cursor. This number will always be > than zero.

If the cursor is on a tab stop, then the number of characters to the next tab stop are returned.

### Return value

Number of characters between current cursor location and next tab stop.

## do stmt while (cond);

### Description

The do keyword is used to implement a C-like do loop. This loop structure is similar to the (while) macro, except the condition for loop termination is tested at the end of the loop. stmt is evaluated at least once.

*break* and *continue* may be used within a do-loop.

### Return value

Returns zero.

## dos([command] [use_shell] [completion])

### Description

THIS FILE IS PROVIDED FOR COMPATIBILITY ONLY via the brief.cr MACRO FILE.

Please see the *shell()* command for further information.

### Return value

Returns the shell exit status (under Unix, 0 means command exited successfully, non-zero means command failed for some reason).

## double var1, var2, ...;

### Return value

Declares a floating point variable. (64-bit size).

### Description

This primitive is used for declaring a floating point variable. Floating point variables are stored using double precision (64-bits).

Future versions of CRiSP may use single precision floating point format for floats. If you want to ensure that your code will use double precision format, use the ***double*** declaration keyword.

### See also

float(pg. 92).

## down([lines])

### Description

Moves the cursor to the same column on the line below. If **lines** is specified then cursor is moved to the **lines**'th line after the current. **lines** may be negative in which case the cursor moves backwards.

### Return value

Returns 1 if the cursor moves or 0 if already at the end of the buffer (up to screen size lines beyond the physical end of the buffer).

### See also

beginning_of_line()(pg. 51)., page_down()(pg. 150)., page_up()(pg. 150)., up()(pg. 212).

## drop_anchor([type])

### Description

This macro is used to define a region within the current buffer. type specifies the type of region, and the values are given below. If omitted a normal marked region is selected.

Regions are areas of a buffer upon which some macros have special effects, e.g. *cut()* and *copy()* are used to extract fragments of buffers and save them for later re-insertion. The *search_fwd()* and *search_back()* macros can be told to limit their searches to the highlighted regions, etc.

Regions are displayed on screen either in a different color, or in reverse video so that they stand out.

The following are the different region types:

| | |
|---|---|
| 1 | Normal |
| 2 | Column |
| 3 | Line |
| 4 | Non-inclusive. |

A normal mark is a region which encompasses from the place where the anchor was dropped up to and including the current cursor position. A non-inclusive mark is the same but does not include the end position of the area.

A line mark selects entire lines, and allows for easy movement of text from one part of a buffer to another.

A column mark is similar to a normal mark, except it displays differently on screen, and allows rectangular sections of the current buffer to be marked.

The currently selected marked area can be found, via *inq_marked()*. Column operations are not directly supported by CRiSP internally, but instead are supported by the macros supplied with CRiSP.

Regions are nestable, i.e. multiple *drop_anchor()*'s can be issued without any intervening *raise_anchor()*.

The marked region can be cleared by calling *raise_anchor()* or performing a *copy()* or *cut()* operation on the buffer.

### Return value

Nothing.

### See also

end_anchor()(pg. 82)., get_region()(pg. 96)., mark()(pg. 142)., swap_anchor()(pg. 203)., raise_anchor()(pg. 157).

## drop_bookmark([bookmark], [yes], [buf], [line], [col])

### Description

This macro is used to set a bookmark. A bookmark is like a normal bookmark - it is a place holder in a buffer.

| | |
|---|---|
| **bookmark** | is a string or integer used to identify the bookmark so you can jump back to it later on. |
| **yes** | is a string which if set to **'y'** or **'Y'** will cause any previous bookmark with the same identifier to be overwritten. |

If buf, line & col are non-NULL, then bookmark **bookmark** is set to the specified buffer at the designated line and column position. If buf, line or col are NULL then the bookmark is set from the current buffer, line and column position.

### Return value

Greater than zero if bookmark dropped, or zero if operation failed.

### See also

bookmark_list()(pg. 52)., delete_bookmark()(pg. 69)., goto_bookmark()(pg. 100).

## echo_line([flags], [screen_id])

### Description

The *echo_line()* macro is used to control which fields are visible in the character mode status line.

If **flags** is omitted then the current setting is returned.

**flags** is an integer expression, whose bit pattern is interpreted as follows:

| | |
|---|---|
| 0x01 | **Line:** field enabled. |
| 0x02 | **Col:** field enabled. |
| 0x04 | Percentage through file enabled. |
| 0x08 | Current time enabled. |
| 0x10 | **RE**member/**PA**use reminders enabled. |
| 0x20 | Cursor type displayed. |
| 0x40 | Echo line is not updated. |

**screen_id** specifies which CRiSP screen window to use. If not specified the current screen is used.

The cursor type is only used if the screen cannot change the cursor type to indicate insert/overtype and normal/virtual space.

### Return value

Previous value of flags.

### See also

inq_screen()(pg. 121).

## edit_control([ctrl-flags], [edit-flags], [vfy-timer], [vfy-keycount])

### Description

This primitive is used to control various aspects of the editing session. The first argument, *ctrl-flags* is used to control the following:

If the ECF_GET_VERIFY_TIMER bit (0x01) is set then the current file sanity verify timer value is returned. If the ECF_GET_VERIFY_KEY_COUNT bit (0x02) is set then the current file sanity verify keycount is returned. The file sanity checker is used to detect changes to files on disk which are currently being edited.

*edit-flags* is an optional argument which is currently reserved. The following flags are defined:

ECF_VERIFY_ENABLED
: If this flag is enabled CRiSP will perform a sanity check when either a sub-shell is spawned, CRiSP is stopped (^Z) or receives the input focus (GUI versions only).

ECF_FOCUS_CLICK
: This flag enables CRiSP to flush mouse button events when receiving input

focus. This is useful if you find it annoying when clicking on the CRiSP window to receive input focus, but at the same time, CRiSP moves the cursor to the position clicked. By enabling this flag, CRiSP will ignore the mouse button events which occur at the time input focus is received. There is not guarantee that this flag will have any affect - network latency can affect the successfulness of this mechanism.

ECF_ENABLE_CUA
This flag is used to enable CUA like behaviour of menu bar menus. Normally menu bar items can have mnemonic accelerators associated with them by using the '&' character prior to the letter in the menu bar entry which is to be the accelerator. If this flag is enabled then these accelerator letters will be displayed underlined. However, enabling this feature can 'steal' some of the Alt-key combinations from the main editing area.

ECF_DELETE_REGION
If this flag is set then inserting text with a highlight active will cause the highlighted text to be deleted first.

ECF_FONT_8BIT
This bit set means to display all 0..255 characters in an X11 font even if the font-metric information doesn't appear to be correct. This bit was added because some X11 fonts don't provide enough information to determine if the top half of the character set is visible or not.

ECF_BACKSPACE_DEL_SWAP
This bit is used to enable swapping of the <Backspace> and <Del> key and is primarily aimed at DEC VT100 style keyboards where there is no explicit or easy to reach DEL key but the BACKSPACE key acts like the DEL key.

ECF_PC_KEYBOARD
This bit enables certain aspects of keyboard mapping to promote more consistent keyboard use on certain platforms. Originally designed for the PC keyboard supplied with HP/UX systems, this makes the numeric keypad act the way each key is labelled. Thus the '1' key acts as an <End> key rather than inserted the character 1.

ECF_DISABLE_LINKS
Controls the appearance of text link buttons in the display. If this flag is set, then links will have no special appearance or function.

ECF_DISABLE_SPELL
Allows you to globally disable the spell checker.

ECF_MBCS Reserved.

ECF_NO_HILITE_CUR_LINE
If set, then current line highlighting for non-colorized buffers is disabled.

ECF_NO_SET_FOCUS
Used to disable input focus grabbing when a dialog box is popped up.

ECF_LEFT_HAND_SCROLLBAR
If enabled, then vertical scrollbars are placed on the left of the edit buffer.

ECF_HILITE_CUR_LINE_COLOR
If set then current line highlighting is enabled for colorized buffers.

ECF_AUTO_NEWLINE_DETECT
Reserved.

ECF_NUMLOCK_MODIFIER
If enabled then the **<NumLock>** key acts as a key modifier, rather than having its usual meaning. This means that if it is set then any keypad keys which are pressed will be returned as their non-NumLocked value ORed with the KEYMOD_NUMLOCK value. This is expressly designed to be used for the EDT keyboard support emulation.

The ***vfy-timer*** and ***vfy-keycount*** values specify the values to use for the file sanity checking code.

### Return value

If ECF_GET_VERIFY_TIMER is specified then the current value of the verify timer (in seconds is returned). If ECF_GET_VERIFY_KEY_COUNT is specified then the current value of the verify keystroke counter is returned. If neither of these flags are specified then the current value of the edit-flags parameter is returned.

## edit_file([mode,] [file1], [file2], ..)

### Description

This macro is the usual way of editing a file; if a buffer is already allocated for the file, then that is set as the current buffer. Otherwise a new buffer is created, the contents of the named file are read into that, and the filename is associated with that buffer (so that later on a *write_buffer()* can be used to save any changes made to the buffer).

The argument list is a set of strings and optional mode specifications. Mode specifications are integers which are used to control how subsequent files are read in. The default is '*normal*'. In normal mode, CRiSP treats the file as a text file suitable for the current operating system, e.g. under Windows carriage return and linefeed are assumed to terminate a line, whilst under Unix, only a linefeed is treated as a line terminator.

For ASCII files, newlines preceded by a carriage return are treated as DOS files and the carriage return is removed and a flag is set for the buffer so that when the file is subsequently written back to disk, the carriage returns will be automatically added on. Only the first line is checked for the CR character before a LF. If there isn't a CR before a LF at the end of the first line, then the file is a normal Unix ASCII file. Any subsequent CR's detected in the file are kept.

If a filename starts with a '|' character then everything after the '|' character is assumed to be a command which is passed to the *popen()* call and the data from the pipe is read in to a new buffer, whose name includes the pipe-character at the start of the file name.

If the mode specification flag is specified then it applies to all subsequent files in the argument list, or until another mode specification is met. The mode specification flag can be used to 'override' the default heuristic described in the previous paragraphs and is a set of bits defined as follows:

| Value | Name | Description |
|---|---|---|
| 0x00 | EDIT_NORMAL | This is the default. |
| 0x01 | EDIT_BINARY | Force file to be treated as a binary file. |
| 0x02 | EDIT_ASCII | Force file to be treated as an ASCII file. |
| 0x04 | EDIT_STRIP_CR | Remove trailing carriage-return characters at the end of each line. This is most appropriate for MS-DOS text files. |
| 0x08 | EDIT_STRIP_CTRLZ | Remove trailing Ctrl-Z character at the end of the file. This is most appropriate for MS-DOS text files. |
| 0x10 | EDIT_PRIVATE | Where possible do not rely on disk file to remain stable whilst editing. Err on the side of caution and buffer data from file internally. This is an optimisation only. |

This macro supports wild-cards and the ability to edit multiple files at once. If more than one argument is specified, then a separate *edit_file()* is performed on each file, and the current buffer is set to the last file read in.

The normal c-shell wildcards are supported:

| | |
|---|---|
| * | wild-card -- matches any number of characters. |
| ? | wild-character -- matches any single character. |
| [l-m] | any character in range at this character. |

| | |
|---|---|
| ~/ | as first part of filename matches home directory |
| ~user/ | matches 'user's home directory. |

If no filenames are specified, the user is prompted for a filename.

### Return value

Returns 1 if new file edited; zero if edit_file aborted at the Edit file prompt; returns < 0 if its a new file or permission denied. On an error, the global variable *errno* will be set to the reason for the failure.

### Example

The following example can be used to read in all .c files in the current directory:

**edit_file("*.c");**

### See also

create_buffer()(pg. 62)., create_nested_buffer()(pg. 64)., inq_buffer()(pg. 105)., read_file()(pg. 162).

## enable_interrupt([rel-val], [abs-val])

### Description

This primitive is used to enable keyboard interrupts (usually the ^C key) to abort certain operations which may take a long (or infinite) amount of time. Examples of operations which can take a long time are the search command, or playing back a keystroke macro. CRiSP maintains an internal *interrupt level* counter. When this is zero keyboard interrupts are disabled; when this is greater than zero then keyboard interrupts are enabled. Only certain primitives check this flag and allow an abort to be generated.

Either **rel-val** or **abs-val** should be specified. **rel-val** adds the value to the current interrupt level. **abs-val** sets the current interrupt level to the value specified.

By only allowing certain operations to be aborted ensures that macros do not leave CRiSP in an inconsistent state.

### Return value

The previous value of the interrupt mask.

## end_anchor([line], [col])

### Description

*end_anchor()* is used to define the ending co-ordinates of a highlighted region which is not necessarily bounded by the cursor. Normally one corner of a selected region is bounded by the cursor, the other co-ordinate being supplied at the time the *drop_anchor()* primitive is called.

Using this primitive allows a highlighted region to be created, allowing the cursor to move within a bounded region. (Cursor movement is not restricted). This primitive enables complicated user interface components to be implemented, such as the quadruple-mouse click feature which highlights the entire buffer without moving the cursor.

### Example

The following example can be used to select the entire buffer without moving the cursor.

```
save_position();
end_of_buffer();
drop_anchor(MK_LINE);
end_anchor(1, 1);
restore_position();
```

### Return value

-1 if no anchor or current buffer defined. 0 otherwise.

### See also

drop_anchor()(pg. 78)., get_region()(pg. 96)., mark()(pg. 142)., swap_anchor()(pg. 203)., raise_anchor()(pg. 157).

## end_of_buffer()

### Description

Moves the cursor to the end of the last line in the current buffer.

### Return value

Nothing.

## end_of_line()

### Description

Moves the cursor to the end of the current line.

### Return value

Returns 1 if cursor moves; zero if cursor already at end of line.

### See also

beginning_of_line()(pg. 51).

## end_of_window()

### Description

Moves the cursor to the last line of the window. The cursor stays in the same column position.

### Return value

Returns 1 if cursor moved; 0 if cursor stayed in the same position.

### See also

end_of_buffer()(pg. 83)., top_of_window()(pg. 208).

## error(fmt, [arg1], [arg2], ..)

### Description

This macro is used to print a message on the status *prompt* line. *fmt* is a string with possible embedded printf-like % options. arg1, arg2, .. are optional arguments (up to 4 parameters) which either evaluate to strings or integer values.

The % options are the same as for the underlying printf() which the system supports. (Internally the string coding is handled by sprintf). Please see the description of the *message()* macro for a list of the minimum supported set of % options.

This macro is similar to *message()* except that the message string is classified as an error. Error messages are printed in the error colour (see *color()*).

In addition, if the pause_on_error flag has been set (see *pause_on_error()*) then the error message is displayed suffixed with a '..' and CRiSP waits for the user to type any key to continue. This is useful in debugging.

Error messages are truncated if they are too long.

### Return value

Nothing.

### Example

```
string  mac = "fred";
error("Macro %s not loaded", fred);
```

### See also

message()(pg. 143)., printf()(pg. 153).

## evaluate(str)

### Description

This function evaluates the string expression '*str*' as if it were written in the CRUNCH language.

NOTE: This function currently only supports symbol lookup.

### Return value

Result of evaluating the expression contained in the string '*str*'.

### See also

execute_macro()(pg. 84).

## execute_keystroke(key, flags)

### Description

This macro is used to execute a macro associated with a keystroke, as if the user had pressed the key. This is designed for macros which need to intercept function keys but need to forward the execution on if they shouldn't have received the callback.

For example, this is used by the template editing macros. When <Enter> is pressed, if template mode is enabled then the appropriate processing is taken. If however, templates are not enabled for <Enter> then the key is forwarded as if templates were not in place.

| | |
|---|---|
| **key** | Keycode to execute; use **inq_kbd_char(0)** to retrieve the current key typed. |
| **flags** | Integer; if set to zero, then the key is simply re-interpreted (be careful of infinite loops). A value of 1 causes the current buffer local keymap to be ignored (hence this acts as a forwarding function). |

### Return value

Nothing

### See also

inq_kbd_char()(pg. 111).

## execute_macro([macro], [args])

### Description

This macro is used to execute another macro. It is needed because it is the only way to execute a macro which is defined by a string expression, rather than by virtue of the CRiSP language syntax.

macro should be a string expression and if omitted is prompted for; args are optional and are the arguments passed to the macro.

If the string 'macro' has the form 'var=value' then the expression is parsed into an assignment statement. The value parameter is treated as a number if it starts with a digit or a minus sign.

If the 'macro' parameter is specified as a list, then each string in the list will be executed in turn. (This avoids having to implement a for-loop to execute each string in the list).

### Return value

Returns the value of the last macro executed.

### See also

evaluate()(pg. 84).

## exist(filename, [canon])

### Description

Checks for existence of file. The existence of the file is performed by doing a stat() call on the named file. Therefore this can be used for all file types (including character/block devices, etc).

If *canon* is not specified, then the filename is canonised first. This involves expanding any tilde prefixes and converting DOS style filenames to Unix style ones. If canon is specified then this conversion is not performed. (This is designed to be used to distinguish operating system types).

### Return value

Non-zero if filename exists; zero if file does not exist. If file does not exist, then the global variable *errno* is set to the reason for the error.

### See also

is_directory()(pg. 132)., stat()(pg. 201).

## exit(str)

### Description

This macro is used to exit from CRiSP, or from a recursive *process()* macro invocation.

If called at the bottom level of CRiSP, CRiSP goes through its exit procedure and prompts the user if he/she is sure, if there are any buffers which have been modified.

If called from within a nested invocation of *process()*, on the next attempt to read keyboard input the calling *process()* macro will return.

***str*** is an optional string argument which can be used when exiting CRiSP. It should be a single character from the set: **[nyw]** to cause CRiSP to exit and answer the "Buffers modified" prompt, e.g. exit("w") would exit CRiSP and automatically save any modified buffers.

### Return value

Nothing.

### See also

abort()(pg. 44).

## exp(x)

### Return value

Returns the irrational number 'e' to the power of x as a floating point value.

### See also

log()(pg. 140)., log10()(pg. 140).

## _extension()

### Description

This macro is a callback macro called whenever CRiSP edits a file via the edit_file() primitive. It is provided to allow users to write their own hooks, e.g. to set tab stops on a file extension basis.

This macro is called with no arguments.

The use of this macro is discouraged as CRiSP provides a more powerful and efficient mechanism for handling files based on the model buffer mechanism. Consult the macro file **packages.cr** to see how this works.

## extern TYPE var1, var2, ..;

### Description

The extern keyword is used to declare variables which are not defined within the current file or function, but are available elsewhere. Essentially the extern keyword creates a place holder in the symbol table to avoid undefined symbol reference errors.

## fabs(x)

### Return value

Absolute value of x, |x| as a floating point value.

## _fatal_error()

### Description

This is a callback macro. It is called when CRiSP detects a segmentation violation internally, and exists to allow the called macro to take emergency actions and save any modified buffers to disk. Segmentation violations should not normally occur, and may occur due to bugs in either a users macro, or some rarely used primitive. The macro that is called should use very few facilities of CRiSP to avoid hitting the same bug again.

Refer to the **core.cr** macro for an example of how this macro is used.

## filename_extension(filename)

### Description

This macro extracts the trailing extension from a filename and returns it. If no extension is present then a null string is returned. If multiple extensions are present then the last extension only is returned.

### Return value

File extension or null string if no extension present.

### See also

inq_extension()(pg. 109)., inq_names()(pg. 118).

## file_glob(string, [flags])

### Description

This macro implements a shell-style globbing facility, and facilitates writing macros which want to perform wild-card name expansion.

If *flags* is not specified then the expression *string* is expanded as if typed to the shell (i.e. '*', '?', and '[..]' are supported).

If *flags* is specified then it is a collection of bits, which indicate how to expand the expression.

| | |
|---|---|
| GLOB_ALL_DIRS | Specifies that all directories should be scanned - even if they do not match the regular expression. Without this, and doing a recursive scan, might cause sub directories to be missed because the parent directory name does not match the wild card. |
| GLOB_DIRS | specifies that directories will be included in the output list. |
| GLOB_FILES | specifies that normal files will be included. |
| GLOB_RECURSIVE | specifies that a recursive directory scan will be performed, e.g. similar to the Unix *find* command. |
| GLOB_SYMLINK | specifies that files will be lstat()'ed instead of stat()'ed. The difference is that without GLOB_SYMLINK you can cause CRiSP to run out of memory if a symbolic link is encountered which creates a loop in the filesystem directory graph (i.e. a symlink pointing to a parent directory). |
| GLOB_DOT_FILES | says to include filenames or directories which start with a  dot. |

For example,

**file_glob(".", GLOB_RECURSIVE | GLOB_DIRS | GLOB_FILES)**

is equivalent to the Unix command "find . -print".

### Return value

A list of strings corresponding to the filenames which match the wild card expression in string.

### See also

filename_match()(pg. 89).

## file_pattern(filespec)

### Description

This macro is used in conjunction with the *find_file()* macro to implement a directory lookup mechanism.

filespec is a string expression which should evaluate to a file name or a wild-card filename. *find_file()* can then be used to read the filenames of files which match the file-spec.

### Example

See example in *find_file()* for an example of how to iterate through all files in the current directory.

### Return value

Nothing.

### See also

find_file()(pg. 90)., file_glob()(pg. 86).

## filename_append(dir, filename)

### Description

This function is a portable way of creating a filename which consists of the directory, **dir,** concatenated with the filename, **filename.** If **filename** is an absoluite filename, then a copy of **filename** is returned.

This function is designed to hide the differences and peculiarities of Microsoft Windows and Unix filenames. Although Microsoft Windows can use Unix style forward slashes in filenames, it does not like multiple redundant forward-slashes. Unix can handle these quite happily.

This macro makes it easier to avoid these peculiarities and the need to ensure invalid filenames are not created.

### Example

The following example returns `"/usr/local/crisp"` under Unix. Under Windows, the result would be `"\usr\local\crisp"`:

**filename_append("/usr/local/", "crisp")**

### Return value

String containing a new filename which is the concatenation of the directory and filename,.

### See also

filename_delimiter(),(pg. 88)., simplify_filename()(pg. 196).

## filename_delimiter()

### Description

This function returns a string which is suitable for use in generating filenames. The string is designed to be used as a delimiter between one directory name and another.

This function is designed to aid portability of macros between Unix and Microsoft Windows.

### Return value

Returns "/" under Unix. Under Microsoft Windows, returns "\".

### See also

filename_append(),(pg. 87)., filename_has_dirspec()(pg. 88)., simplify_filename()(pg. 196).

## filename_first_component(filename)

### Description

This function returns an integer corresponding to the first directory name delimiter in the input string.

This is similar to the macro code:

```
string filename;
filename = index(filename, "/");
```

but handles the portability issue of forward and backward slashes depending on the operating system, i.e. Unix versus Microsoft Windows filenames.

### Return value

Returns position of first "/" (or "\" if Microsoft Windows) in the specified filename. Zero if no slashes are present.

### See also

filename_last_component()(pg. 89)., simplify_filename()(pg. 196).

## filename_has_dirspec(filename)

### Description

This function returns a TRUE or FALSE value depending whether the specified filename (a string value) contains a directory separator or not, i.e. whether the filename refers to a file in the current directory or a relative or absolute filename.

This primitive is designed to make it easier to write machine independent macros which need to process filenames.

### Return value

Returns 1 if "/" is present under Unix. Under Microsoft Windows, returns 1 if "\" or "/" is present.

### See also

filename_append()(pg. 87)., simplify_filename()(pg. 196).

## filename_is_absolute(filename)

### Description

This function returns a TRUE or FALSE value depending whether the specified filename is an absolute or relative filename. Under Unix, an absolute filename starts with a "/". Under Windows, an absolute filename may start with a "/" or a "\" and may optionally be preceded by a drive designator.

This primitive is designed to make it easier to write machine independent macros which need to process filenames.

### Return value

Returns 1 if filename is absolute. Zero otherwise.

### See also

filename_has_dirspec()(pg. 88)., filename_append()(pg. 87)., simplify_filename()(pg. 196).

## filename_last_component(filename)

### Description

This function returns an integer corresponding to the last directory name delimiter in the input string.

This is similar to the macro code:

```
string filename;
filename = rindex(filename, "/");
```

but handles the portability issue of forward and backward slashes depending on the operating system, i.e. Unix versus Microsoft Windows filenames.

### Return value

Returns position of last "/" (or "\" if Microsoft Windows) in the specified filename. Zero if no slashes are present.

### See also

filename_first_component()(pg. 88)., simplify_filename()(pg. 196).

## filename_match(file, pattern)

### Description

This function can be used to compare a filename to see if it matches a filename regular expression. A filename regular expression is a regular expression similar to that accepted by the command line shells on Unix systems (e.g. you can use * for wildcard, ? for wild-character, and [..] to select a range of characters).

**file** is the filename being tested; **pattern** is the regular expression or expressions to test against. **pattern** can be a list or regular expressions or a string containing a single regular expression

### Example

The following example returns 1:

**filename_match("abc.txt", "*.txt")**

### Return value

If **pattern** is a string, then 1 if the filename matches; 0 otherwise. If **pattern** is a list, then the index into the list that matched the expression or -1.

### See also

find_file()(pg. 90)., file_glob()(pg. 86)., re_search()(pg. 158).

## find_distribution_file(filename)

### Description

This primitive is used to aid in the support of the CRiSP user environment. A relative filename is specified and the function checks to see if the file exists in the users private configuration directory or in the distribution.

This mechanism is used to allow users to create files in the configuration directory which will override certain CRiSP files, e.g. colorizer files.

### Return value

Full path to the specified file.

### Example

The following example can be used to find the filename of the C colorizer file:

string filename = find_distribution_file("src/keyword/c.kwd");

### See also

find_file()(pg. 90)., filename_match()(pg. 89).

## find_file([name], [size], [mtime], [ctime], [mode])

### Description

*find_file()* is used in conjunction with *file_pattern()* to evaluate filenames in a particular directory. The base directory and wild-card filename pattern are set via *file_pattern()*. Thereafter *find_file()* can be called repeatedly, until it fails, receiving the next file name which matches the pattern set in *file_pattern()*.

name is an optional name of a string variable, which receives the filename which matched the pattern if the *find_file()* was successful. size is the size of the file (in bytes), and is an integer variable. mtime and ctime are the names of integer variables and receive the last modification and creation times, as returned by the Unix stat() system call. mode is the file mode as returned by stat(), and should be the name of an integer variable.

There are two hash-defines in crisp.h which define two of the bit definitions for the mode field. These definitions are taken from the /usr/include/sys/stat.h file. If you use these #define's be careful to ensure they are compatible with your Unix system.

For a more thorough example of the use of these primitives, see the abbrev.m macro file.

### Example

The following example evaluates the names of all files in the current directory and inserts the names and mode information into the current buffer.

```
string  name;
int     mode, size;

file_pattern("*");
while (find_file(name, size, NULL, NULL, mode) > 0) {
    sprintf(buf, "file=%s, size=%d, mode=%x",
            name, size, mode);
    insert(buf + "");
}
```

### Return value

Returns zero if there are no more files; returns 1 if next directory entry successfully received.

### See also

file_pattern()(pg. 87)., file_glob()(pg. 86).

## find_line_flags([buf_id], [line_no], flags, and_mask, or_mask, value)

### Description

This primitive is used to search a buffer looking for lines which have been marked with a particular set of flags. For example, this can be used to find the next set of different lines after a **diff** operation has been performed. Lines can be marked with upto 32 bit flags (see set_line_flags).

| | |
|---|---|
| **buf_id** | Buffer to search; if not specified, use the current buffer. |
| **line_no** | Specify line to start the search; if omitted, current line is used. |
| **flags** | Specifies the type of search. The following values can be used (OR'ed together)<br>LF_FORWARDS: Perform forward search.<br>LF_MATCH_EQ: Search for line where the line flags AND **and_mask** OR **or_value** matches **value.**<br>LF_MATCH_ANY: If specified means to take the line flags and AND them with **and_mask;** if the result is non-zero then search has succeeded. |
| **and_mask, or_mask** | The flags for each line are logically anded with **and_mask** and OR'ed with **or_mask**. If the result matches **value** then the search terminates. |

### Return value

Line number of line matching the search conditions. 0 if search fails.

### See also

inq_line_flags()(pg. 113)., set_line_flags()(pg. 183).

## find_macro(filename)

### Description

This primitive is similar to the load_macro() primitive but it doesn't actually load the specified macro. The current macro path (CRPATH) is searched looking for the specified file. If the specified file has an extension, then an exact match is located. If no extension is specified, then files with the extensions: .cm, .cr, and .m are searched in that order.

This macro is designed to allow user macros to locate macro files using the same algorithm as the load_macro() primitive uses.

### Return value

String containing full path and filename of macro file, or the NULL string if no file could be found.

### See also

autoload(pg. 49)., load_macro(pg. 140).

## find_marker([rsvd])

### Description

This macro is used to locate the next line in the buffer which has a mark set. (See the *mark_line()* macro for more details). This primitive is provided to allow the vi-mode g// and v// commands to rapidly locate text previously marked.

On successful execution the current line is set to the location of the next marker, and the marker is automatically removed.

### Return value

Returns 1 on success and 0 if no more markers found.

### See also

find_line_flags()(pg. 90)., mark_line()(pg. 142).

## find_path(path_spec, file)

### Description

The find_path function is used to scan a list of directories looking for the designated file. For example, this function can be used to locate a variable in the PATH environment variable, by doing something like:

```
string path_list = split(getenv("PATH"), ":");
string real_name = find_path(path_list, "csh");
```

The path_spec variable should be a list of directory names to try. file should be a string containing the name of the file to locate.

### Return value

An integer indicating the index into the list where the directory was found. If the file wasn't found then -1 is returned.

## first_time()

### Description

CRiSP maintains an internal 'first_time' flag for every macro; after the first call to the macro, the flag is set to FALSE. This makes it easier for macros to perform a once-only initialisation.

### Return value

1 if this is the first time a macro has been called; 0 otherwise.

## float var1, var2, ...;

### Return value

Declares a floating point variable. (64-bit size).

### Description

This primitive is used for declaring a floating point variable. Floating point variables are stored using double precision (64-bits).

Future versions of CRiSP may use single precision floating point format for floats. If you want to ensure that your code will use double precision format, use the *double* declaration keyword.

### See also

double(pg. 78).

## floor(x)

### Return value

Largest integer not greater than x, as a floating point number.

### See also

ceil(pg. 54).

## fmod(x, y)

### Return value

Floating point remainder of x/y (x and y both floating point), with the same sign as x.

## font_ctl(cmd, [flags], [name])

### Description

This primitive is used to enumerate fonts, styles and sizes in a platform independent manner and is used by the Font selection dialog.

| | |
|---|---|
| **cmd** | Is a command to invoke. Supported commands at present include:<br>0 - list all font family names available.<br>1 list font sizes available.<br>2 list font styles available. |
| **flags** | If the bottom bit of flags is set then only fixed width fonts will be enumerated. |
| **name** | If specified, then only fonts matching the name will be listed. For example you can use this to enumerate the font styles for a specific font. |

### Return value

A list of values corresponding to the command request.

### See also

inq_font()(pg. 110)., set_font()(pg. 181).

## for (init; cond; post) stmt

### Description

This macro is used to implement a C-like loop structure. init, cond, post and stmts maybe expressions or a list of statements to execute. The init expression is evaluated first. Next the cond condition is evaluated. If it is non-zero, then the stmt expression is evaluated, followed by evaluation of the post expression. The loop continues until the cond expression evaluates to non-zero.

### Return value

Returns the value of the last statement executed in the loop. Normally zero.

## format(fmt-string, ...)

### Description

The *format* primitive is similar to the *sprintf()* primitive but instead of putting the result into a variable, the result is returned as a string. In fact, sprintf() can emulated by doing:

```
string s = format(fmt, ...);
```

### Return value

Formatted string value.

### See also

printf()(pg. 153)., sprintf()(pg. 200).

## frexp(x, exp)

### Return value

This function converts the floating point number x into a normalized fraction in the interval [1/2, 1]. This value is returned as the power of 2 representing the exponent is stored in exp. exp should be an integer variable.

## get_color([screen_id], [flags])

### Description

This macro can be used to find out the values of the color attributes of the current screen (if *screen_id* is not specified), or the specified screen (if *screen_id* IS specified). A list of color names allocated to each object type on the display is returned. The order of color objects is defined in the table below.

If *flags* is specified then a list of integer valued attributes will be returned corresponding to the attributes to be used by the colorizer mechanism.

| **Code** | **Mnemonic** | **Description** |
|---|---|---|
| 0 | COL_BACKGROUND | Background color of the screen. |
| 1 | COL_FOREGROUND | Foreground color for all windows. |
| 2 | COL_SELECTED_WINDOW | Color of selected window title. |
| 3 | COL_MESSAGES | Normal color of prompts and messages and Line:/Col: fields. |
| 4 | COL_ERRORS | Error message color. |
| 5 | COL_HILITE_BACKGROUND | Color of background for a highlighted area. |
| 6 | COL_HILITE_FOREGROUND | Color of foreground for a highlighted area. |
| 7 | COL_INSERT_CURSOR | Color associated with the insert mode cursor. |
| 8 | COL_OVERTYPE_CURSOR | Color associated with the overtype mode cursor |
| 9 | COL_BORDERS | Color associated with the window borders. |

### Return value

Returns a list of color names, which may have been modified by appropriate use of the X11 resource database. Returns -1 if the specified screen does not exist.

If *flags* is specified then the color attribute flags are returned in a list, instead of the color names.

### See also

set_color()(pg. 179).

## list get_exception_info()

### Description

This primitive retrieves information about the current run-time exception and is used to allow the calling macro create informative dialogs about the reason for the current error.

The return value is a list of values (or a **NULL** value if no exception is in progress). The values in the list are exception specific.

### Return value

A list or **NULL** if no exception is in progress.

## gethostname()

### Description

This function returns the name of the current machine.

## get_mem_info(var)

### Description

This function is used to retrieve the current virtual memory settings for CRiSP. This function is needed for the Options->Memory Configuration dialog box.

The variables and types are subject to change and normal macro programmers should not need to use this function.

### Return value

Value of the memory info parameter as described by the *var* parameter.

### See also

mem_info()(pg. 143).

## get_mouse_pos([x], [y], [win_id], [line], [col], [where])

### Description

This primitive is used to process a mouse message. When a mouse button is clicked or dragged, CRiSP generates a callback for a button key. To retrieve information about where the mouse was on the screen at the time of the mouse event, you need to call this function.

### Return value

Time in milliseconds since last mouse event.

### See also

get_mouse_screen()(pg. 94)., process_mouse()(pg. 154)., translate_pos()(pg. 209).

## get_mouse_screen()

### Description

This primitive returns the object id of the screen where the last mouse action occurred. It is designed to be called from a mouse key callback routine, and provides a way for the mouse macro code to determine which of multiple DBOX_SCREEN widgets the mouse was clicked on.

### Return value

Object id of last mouse press.

### See also

get_mouse_pos()(pg. 94)., process_mouse()(pg. 154)., translate_pos()(pg. 209).

## get_parm([arg], var, [prompt], [length], [default])

### Description

*get_parm()* is the mechanism for accessing parameters passed to macros. *get_parm()* can also be used to prompt the user for input, and optionally specify a default value.

Arguments passed to macros are numbered, zero upwards.

| | |
|---|---|
| **arg** | specifies the argument to retrieve - zero is the first, 1 is the next, etc. If it is **NULL**, then the user is prompted for the value, and the **prompt, length** and **default** parameters are used. |
| **var** | is the name of an integer, string or list variable. If the user is being prompted, then only integer and string expressions may be input. List expressions are not supported. |
| **prompt** | If **arg** is **NULL** then the user is prompted to type in a value. This field is the prompt which is used. |
| **length** | Maximum length of text. Omit for indefinite length. Specifying a non-zero value means the user does not need to hit <Enter>. For integer variables, there is no **length** parameter, and instead this parameter is the default value. |
| **default** | When a prompt is displayed, this value is displayed as the default value. Pressing **<Enter>** uses this as the default input value. For integer variables this parameter is ignored (the default is passed in as the length). |

Arguments passed to macros are passed as call by name, i.e. every time a *get_parm()* is issued on a particular parameter, that parameter is re-evaluated. This can be very useful sometimes, and at other times it can cause anomalous side-effects.

### Return value

Zero if user aborted prompt, or argument number does not exist. Greater than zero if successful.

## get_property(obj_id, property)

### Description

This function retrieves the value of a property previously associated with an object via the *set_property()* function. If the specified property does not exist then a value of **NULL** will be returned. It is advisable to use a polymorphic variable when accessing the value of a variable in order to detect errors.

Property names are enclosed in quotes (i.e. are string constants).

Object identifiers are created, for example, when creating a dialog box or a user defined dictionary (e.g. via create_dictionary()).

The crunch macro language provides a special notation for accessing properties. You can use the dot operator to refer to a property, so long as the property name is a valid crunch symbol. For example the following two statements are equivalent.

```
get_property(dict, "word_count");

dict.word_count;
```

### Return value

Value of the specified property or **NULL** if the object or variable does not exist.

### See also

create_dictionary()(pg. 63)., dialog_box()(pg. 72)., dict_exists()(pg. 73)., dict_list()(pg. 74)., remove_property()(pg. 168)., set_property()(pg. 185).

## get_region([buf_id], [entire_buffer])

### Description

This primitive returns a string containing the highlighted region of the specified buffer, or the entire buffer. Newlines are inserted at line boundaries.

If ***buf_id*** is not specified then the current buffer is used.

If ***entire_buffer*** is specified and is non-zero, then get_region() will grab the entire buffer, avoiding the need for dropping a highlight and setting the cursor to the end of the buffer.

At present, column regions are NOT supported -- they are treated as normal ones.

### Return value

String containing highlighted region.

### See also

inq_marked()(pg. 116).

## get_selection([vartype], [type])

### Description

This primitive is used to get a copy of the X11 selection buffer. Because the X11 cut & paste mechanism works asynchronously, so does this. Calling get_selection() causes CRiSP to request the current version of the selection buffer. The actual data will arrive later. When the data arrives, CRiSP triggers a registered macro (value 13 -- REG_SELECTION). The calling macro should register a callback for this trigger, and call get_selection() again. Inside the trigger, get_selection() returns the string containing the data to be pasted into the buffer. The calling macro is free to do whatever it wishes with this data.

If no application currently holds the scrap then an error message will be displayed, and the bell will sound indicating the failure.

This primitive is available under the X11 version of CRiSP only. Under other systems it always returns a NULL string.

The two arguments are optional and are used to mark the type of selection which will be returned. When a macro wants to get the selection it should specify the *type* parameter to indicate the region type required. Normally, this is either omitted or specified as the symbolic constant MK_NORMAL. However the user may want to paste the selection buffer in as a column region, in which case this variable should be specified as MK_COLUMN.

The actual pasting type is made available to the registered callback routine. It can get the paste type by specifying the first argument, *vartype*, as the name of an integer variable to receive the marked type specified in the original call to *get_selection()*.

This mechanism allows the first caller to get_selection() to pass information back to the registered macro.

For an example of how this is used in practise, refer to the mouse.cr and xwin.cr macros supplied in the source directory (/usr/local/crisp/src/crunch).

### Return value

This macro returns a NULL string when called outside the triggered macro, and the value of the primary selection when called from inside the triggered macro.

### See also

clipboard_owner()(pg. 56)., register_macro()(pg. 163)., put_selection()(pg. 156).

## get_system_resources(val, [name])

### Description

This macro is platform dependent and is used to retrieve system specific information. For example, on the Windows platform it can be used to retrieve information about the GDI and USER heap spaces.

### Return value

System specific.

## get_term_characters([top_left], [top_right], [bot_left], [bot_right], [vertical], [horizontal], [top_join], [bot_join], [cross], [left_join], [right_join])

### Description

This macro can be used to find out the current settings for the various characters which make up the special characters needed to display the window borders on the screen.

Each parameter is the name of a string variable to receive the string expression. Any of the parameters may be omitted.

This macro is designed to allow macros to be written which make it easier for the user to tailor the display.

Refer to *set_term_characters()* for further detail on the meaning of each of these characters.

### Return value

Nothing.

## get_term_features(index)

### Description

This function is used to retrieve the value of a terminal feature as previously set by the set_term_features() primitive. *index* is a number corresponding to the feature to retrieve. If *index* is out of range then a **NULL** value is returned. Otherwise the type of the parameter depends on the type of the feature requested.

### Return value

NULL if index out of range; otherwise an integer or string corresponding to the feature value.

### See also

set_term_features()(pg. 187).

## get_term_keyboard(..)

### Description

This macro is designed to allow the calling macro retrieve the assignments for each key, so that a user interface can be written to examine and display the key bindings.

See *set_term_keyboard()* for more details of how this macro is supposed to work.

This macro has not been currently implemented.

### Return value

Nothing.

## getenv(name)

### Description

This function looks up '*name*' in the environment and returns the value of name. It is similar to the C function getenv().

### Return value

Returns a string which is the contents of the named environment variable, or the null string if the variable does not exist.

### See also

putenv()(pg. 156).

## geteuid()

### Description

Returns effective current user id.

This function is system dependent.

### Return value

User id or -2 if function not supported.

### See also

getuid()(pg. 99)., getgid()(pg. 98)., setgid()(pg. 193)., setuid()(pg. 194).

## getgid()

### Description

Returns current group id.

This function is system dependent.

### Return value

Group id or -2 if function not supported.

### See also

getuid()(pg. 99)., setgid()(pg. 193)., setuid()(pg. 194).

## gethostid()

### Description

Returns systems hostid (or 0 if one is not defined).

### Return value

32-bits of host id.

## getpid([buf_id])

### Description

This function is used to find the process ID of CRiSP, or the process ID of the child process of a PTY buffer. If buf_id is not specified then the PID of CRiSP is returned. If the specified buffer does not exist or if the buffer is not currently connected, then an error is returned.

### Example

The following example generates a filename for a temp file in the /tmp directory.

```
string  filename;
sprintf(filename, "/tmp/cr.xx%05d", getpid());
```

### Return value

Returns the current process ID or the process ID of the attached buffer. -1 is returned on an error.

### See also

connect()(pg. 59). , ipc_create()(pg. 128).

## getrlimit(int type, [rlim_cur], [rlim_max])

### Description

This function is machine dependent. On systems that support it, it can be used to find the current users resource limits. *rlim_cur* and *rlim_max* are integer variables to receive the current and maximum values for that resource.

*type* is the resource limit to inquire about.

Refer to your systems manual pages for more information.

### Return value

0 on success; -1 on error. -2 if call not supported.

### See also

setrlimit()(pg. 194).

## getuid()

### Description

Returns current user id.

This function is system dependent.

### Return value

User id or -2 if function not supported.

### See also

geteuid()(pg. 98)., getgid()(pg. 98)., setgid()(pg. 193)., setuid()(pg. 194).

## getwd(ignored, dir)

### Description

This function returns the current directory in the string variable dir. The ignored parameter is reserved for future use.

### See also

cd()(pg. 53).

### Return value

Returns 1.

## global var1, var2, ..;

### Return value

Nothing.

### Description

The global macro is used to make local variables into global variables. Local variables are destroyed when the macro executing them terminates. Global variables maintain their value across macro invocation and occupy permanent static storage.

A variable must have been specified in a previous *int*, *string*, *list*, *float* or *declare* statement before it can be made into a global.

This primitive is normally unnecessary by following the normal placement and storage rules for identifiers. (Identifiers declared outside of a function automatically have a global scope).

### See also

static(pg. 201).

## goto_bookmark([bookmark], [bufnum], [line], [col])

### Description

This function is used to jump to a previously saved bookmark.

| | |
|---|---|
| **bookmark** | This is an integer or a string which is the name of the bookmark to jump to. If it is not specified then the user is prompted for a bookmark name. |
| **bufnum** | If specified  receives the buffer id associated with the specified bookmark. |
| **line** | If specified, receives the line number associated with the bookmark. |
| **col** | If specified, receives the column number associated with the bookmark. |

This macro is used to move the cursor to a previously saved bookmark. If bufnum, line & col are all NULL or omitted, then the buffer and cursor information associated with book_no are selected.

If any of bufnum, line or col are specified, then these variables are modified to have that information associated with the designated bookmark.

### Example

The following example gets the buffer, line & col associated with bookmark 5.

```
int     buf, line, col;

goto_bookmark(5, buf, line, col);
```

The following example jumps to bookmark 5:

```
goto_bookmark(5);
```

### See also

bookmark_list()(pg. 52)., delete_bookmark()(pg. 69)., drop_bookmark()(pg. 79).

### Return value

1 if macro was successful; 0 if bookmark invalid or prompt was aborted.

## goto_line([line])

### Description

Moves the cursor to the beginning of the specified line. If line is omitted it is prompted for.

### See also

goto_old_line()(pg. 100).

### Return value

Nothing.

## goto_old_line([line])

### Description

When a file is read into a buffer, each line is internally numbered. These internal line numbers are maintained even if lines are inserted and deleted from the buffer. *goto_old_line()* moves the cursor to the line whose original line number was line, or as near as possible if the original lines have been deleted.

This macro gets over the problem of editing a source file to correct compilation errors, and the lines with errors moving as text is inserted or deleted.

If line is omitted, it is prompted for.

### See also

goto_line()(pg. 100).

### Return value

Nothing.

## gsub(pattern, replacement, string)

### Description

The *gsub()* macro is used to perform a global substitution on a string. *pattern* is the string to search for. *replacement* is the replacement text. *string* is the string to apply the translation to.

This function is designed to emulate the function of the same *awk(1)* name.

This primitive is actually a function defined in **crisp.cr**. It is implemented as a wrapper around the *re_translate* primitive.

### Example

```
gsub("e", "", "fred had a banana");
```

This deletes all letter 'e's from the string.

```
gsub("[0-9]+", "(1)", "1+2+3");
```

This encloses in parenthesis all numbers in the input string. (Note the need for the double backslash).

### Return value

The translated string is returned.

### See also

re_translate()(pg. 160)., re_search()(pg. 158)., sub()(pg. 203).

## hide_ctl(cmd)

### Description

This function is used to perform miscellaneous functions which affect the outline mode of editing. These functions are designed to be faster than the alternative of coding the operations manually using the other primitives.

**cmd** is a  integer value indicating the operation to perform.

|  |  |
|---|---|
| 0 | Make current line visible. If the current line is not visible, unhides the parent node which contains this line. |
| 1 | Same as above, but marks all lines in the hidden node as being visible. |

### Return value

Returns the current version number * 100 plus the minor level.

## hilite_create([int buf_id], [int type], [int timeout], [int start_line], [int start_col], [int end_line], [int end_col])

### Description

A hilite is a way of marking a buffer in a similar way that the anchor primitives do. The major difference is that hilites are not editable. Under normal circumstances, when an anchor is dropped, attempting to edit will cause the marked area to be deleted (or cut/copy/pasted). A hilite looks the same as an anchor but is not directly editable. Hilites are designed to be used to show things in the edit window, such as a matching search string or intra-line differences.

This primitive creates a new hilite. Any number of hilites can be associated with the current buffer, so you can hilite all occurences of a word rather than being limited to one. Also hilites are visible even when the window is not the current one.

Anchors exist for compatibility with the original BRIEF editor.

|  |  |
|---|---|
| **buf_id** | Optional buffer id. If none is specified then the highlight is created in the current buffer. |
| **type** | Optional type. A type can be used to classify hilites so that a macro can be written without interfering with existing hilites in the buffer. For example, type 1 hilites are used by the differencing engine to show intra-line modifications.<br><br>By giving all hilites the same type, you can easily remove all of them in one go with the **hilite_destroy** primitive. |
| **timeout** | Specifies a timeout in seconds. If specified, then the hilite will automatically be destroyed after this timeout. Useful for temporarily highlighting strings in a buffer. |

**start_line, start_col, end_line, end_col**

Range for the highlight. If **start_line** or **end_line** are omitted then they will default to the current cursor position.

### Return value

Returns zero on success. -1 if specified buffer is invalid or no current buffer available.

### See also

drop_anchor()(pg. 78)., end_anchor()(pg. 82)., hilite_destroy()(pg. 102).

## hilite_destroy([int buf_id], [int type])

### Description

This primitive is used to remove one or more hilites previously created via **hilite_create()**.

| | |
|---|---|
| **buf_id** | Optional buffer id. If none is specified then highlights are destroyed for the current buffer. |
| **type** | If specified then only hilites matching the specified type are removed. If omitted then all hilites are removed. |

### Return value

Returns zero on success.

### See also

drop_anchor()(pg. 78)., end_anchor()(pg. 82)., hilite_create()(pg. 101).

## if (expr) true-clause [else false-clause]

### Description

The expression 'expr' is evaluated, and if it is non-zero (true) or is a non-null string, then the true-clause statements are evaluated. Otherwise, if the false-clause is specified, then it is evaluated.

## index(search-string, pattern)

### Description

This function is used to see whether search-string contains the substring pattern. A literal substring match is performed. This function returns the place in search-string where pattern has been located or 0 if it does not occur in the search-string.

If the user needs to search for regular expressions, then the function search_string should be used instead.

### Example

The following example checks to see if a string is a lower case alphabetic character:

```
string  letter;

if (index("abcdefghijklmnopqrstuvwxyz", letter) == 0)
      message("%c is upper case.", letter);
else
      message("%c is lower case.", letter);
```

### Return value

Returns 0 if pattern cannot be found in search-string; otherwise returns the position pattern starts at in search-string.

### See also

rindex()(pg. 171)., search_string()(pg. 173).

## inherit_buffer_attributes(extension, buf_id)

### Description

This primitive is used by the packages mechanism to enable a buffer with one file extension to be treated as if it had a different extension. This is used when a file without an extension is intercepted with the REG_DEFAULT_FILE trigger. When the calling macro has determined the type of the file (e.g. by examining the first line of the buffer) then this primitive can be called to allow the model-buffer inheritance mechanism to take place.

The effect of this macro is to search for a *model* buffer with the appropriate file extension and to inherit all buffer attributes and buffer local variables from that buffer. This allows, for example, the template editing mode to be set up automatically for a buffer.

Note that the effect of this primitive may have slightly different semantics than just plainly editing a file with the same extension, since a part of the file may have already been read in (e.g. the TAB conversion mode may not have been applied to the first part of the file, but after calling this primitive, the TAB mode may be changed).

### Return value

Nothing.

### See also

set_model_buffer()(pg. 183).

## input_mode(char, flag)

### Description

This macro is used to configure the way certain control characters are interpreted when typed in and to affect various configuration flags. The following characters can cause a problem when typed in from a serial terminal, although they work as function keys when used in the GUI versions of CRiSP: Ctrl-S (0x13), Ctrl-Q (0x11) and Ctrl-Z (0x1a). Normally CRiSP does not see these characters and they perform the actions of XON/XOFF and the Job control stop.

There are two ways to use this macro - one is to set the mode for these particular control characters and the other is to affect certain keyboard operations.

**Serial terminal**

This mode is controlled by specified a value for **char** - which is the control character to be affected.

On serial terminals it is important to enable flow control because the terminal may not be able to keep up with CRiSP when it is redrawing the screen. Some people want to use these characters as keys for macro assignments. This macro exists to allow users to turn on and off these characters, and thus let CRiSP see them.

**char** should be set to the ASCII value of the character to be changed. **flag** should be set to TRUE to allow CRiSP to see the character or to zero to let the operating system take its normal action.

Note, setting either of Ctrl-S and Ctrl-Q causes both to be set together. Setting Ctrl-Z is only meaningful on systems that support Job control.

Note that this primitive is only operational on Unix systems that support the TERMIO feature.

**Keyboard flags**

If **char** is NULL, then the **flag** parameter can be used to set the current keyboard mode flags. The previous value of the flags is returned. **flag** is a set of bits with the following definitions.

| | |
|---|---|
| 0x0001 | KBDF_CURSOR_KEYS. X Windows only. Controls the interpretation of the arrow keys on the keyboard. By default CRiSP treats the arrow keys and the keypad keys 2, 4, 6, 8 as being the same key. Enabling this mode allows the keys to be treated differently and hence you can assign different macros to the keys. |
| 0x0002 | KBDF_KEYPAD_KEYS. X Windows only. When enabled, the keys on the keypad acts as normal typeable keys rather than being treated as function keys. |

### Return value

If **char** is specified then returns 1 if character previously enabled; zero otherwise.

If **char** is NULL, then returns the current keyboard mode flags.

## inq_assignment(key, [convert], [kbd_id])

### Description

This macro can be used to find out what macro a particular keyboard character is assigned to, or to which key or keys a particular macro is assigned to.

**key** is a string denoting the key sequence to be decoded or an integer representing the internal key code. If key is a string then it should be of the form described under assign_to_key()(pg. 47).

**convert** is an integer expression controlling what type of key lookup is performed.

| | |
|---|---|
| 0 | if omitted or zero, then key is treated as a keyboard character (as defined for *assign_to_key()*), and the name of the macro assigned to that keystroke is returned. If the character is not assigned to a macro, and the character is inserted into the buffer when the user hits it, then the string "**self_insert**" is returned. If there is no key assignment for that key, then "nothing" is returned. |
| 1 | **'key'** is taken as the name of a macro and the keys assigned to invoke this macro are returned. The key assignment returned is returned using the portable key definitions defined for "assign_to_key".<br><br>If more than one key stroke is assigned the same macro invocation, then the returned string will contain the string **"<-also>"** separating the key sequences. |
| 2 | **key** should be a string containing a sequence of key stroke definitions, e.g. as returned by inq_keytroke_macro(). It returns a string containing macro code which can be used to create a new macro. |

**kbd_id** is an optional keyboard identifier. If it is not specified then the current local or global keyboards will be searched for the assignment.

### Example

The following example displays the message **'<Alt-H>'**:

```
message("%s", inq_assignment("help"));
```

### Return value

Returns the name of a macro assigned to a keystroke or the name of a keystroke assigned to a macro.

### See also

int_to_key()(pg. 128)., key_to_int()(pg. 134)., use_local_keyboard()(pg. 212).

## inq_borders()

### Description

Not currently implemented.

### Return value

Returns zero if borders are off; non-zero if borders are on.

## inq_brief_level()

### Description

This macro is supposed to return the level of nesting of sub-shells running BRIEF/CRiSP and is equivalent to doing: getenv("BLEVEL").

### Return value

Returns zero if this is a top level CRiSP; otherwise returns 2, 3, 4, ... to indicate level of nesting.

### See also

getenv()(pg. 97)., putenv()(pg. 156).

## inq_buffer([filename])

### Description

Every buffer has an identifier (an integer value) associated with it. These identifiers are allocated when the buffer is created. *inq_buffer()* can be used to get the identifier for the current buffer, e.g. when saving the current buffer in a macro, so that after the macro has finished the buffer selected on entry to the macro can be re-instated.

If filename is specified then the buffer id of the buffer which has the file loaded is returned instead.

If *filename* is an integer then this primitive can be used to test whether the specified buffer actually exists.

### Example

The following example is typical of the code sequence used in the CRiSP macros on entry and exit to each macro.

```
int     curbuf;

curbuf = inq_buffer();
.
.
set_buffer(curbuf);
```

### Return value

Returns the buffer ID of the current buffer or the buffer which has filename loaded into it. Returns zero if no buffer has the filename loaded.

If filename is an integer then it returns < 0 if the buffer no longer exists or zero if the buffer does exist.

### See also

create_buffer()(pg. 62)., set_buffer()(pg. 176).

## inq_buffer_flags([bufnum], [flag-set])

### Description

This macro returns the flags associated with the status of the current buffer (if bufnum is omitted), or the buffer whose identifier is bufnum.

The *flag-set* parameter is used to return one of the flag words associated with the buffer. Because there are potentially more than 32-bits of flag status information, excess information has been spilled over into additional flags. If this parameter is zero or not specified then the default flag-set is returned.

Flag set #0:

| Flag | Meaning |
|---|---|
| 0x01 | If set, buffer has been modified. |
| 0x02 | If set, buffer will be backed up when written away. |
| 0x04 | Buffer is marked as read-only. |
| 0x08 | Reserved. |
| 0x10 | Underlying file has execute permission. |
| 0x20 | Process attached. |
| 0x40 | Buffer is in binary mode. |
| 0x80 | ANSI mode -- color escape sequences will be processed. ANSI mode will only work if the current character map has the ESCAPE and BACKSPACE characters defined appropriately. |

| | |
|---|---|
| 0x100 | Buffer does not map tabs to spaces. |
| 0x200 | Buffer is a system buffer. |
| 0x400 | Make all characters in window visible. |
| 0x800 | Dont save undo info for buffer. |
| 0x1000 | File is new -- so it will be saved on next write_buffer(). |
| 0x2000 | Append <CR> to end of each line on save. |
| 0x4000 | (BF_TITLE) Label window title with full path name. |
| 0x8000 | (BF_HIDE) Text hiding enabled. |
| 0x10000 | (BF_STRIP_CTRLZ) Strip ^Z at end of file. |
| 0x20000 | (BF_APPEND_CTRLZ) Append ^Z when writing file. |
| 0x40000 | (BF_COLORIZE) Apply colorization algorithm. |
| 0x80000 | RESERVED |
| 0x100000 | (BF_MOD_LINES) Show modified lines. |
| 0x200000 | (BF_OLD_LINES) Show old line numbers when line numbers are enabled. |
| 0x400000 | (BF_LOG_FILE) File is a log file which may get extended. |
| 0x800000 | (BF_IS_PIPE) File is a piped command. |
| 0x01000000 | (BF_VOLATILE) File is a volatile buffer. |
| 0x02000000 | (BF_HEX_MODE) Buffer is being operated on in hex mode. |
| 0x04000000 | (BF_ICVT_TAB_TO_SPACES) Convert tabs to spaces on input. |
| 0x08000000 | (BF_ICVT_SPACES_TO_TABS) Convert spaces to tabs on input. |
| 0x10000000 | (BF_EOF_DISPLAY) Display [EOF] marker at end of buffer. |

Flag set #1:

| **Flag** | **Meaning** |
|---|---|
| 0x0001 | BF2_SCROLL_TITLE. If this bit is set then when the contents of the buffer inside a window scrolls to the left and right, then the title of the window will scroll with it. This allows you to display fixed column headings on a buffer. |
| 0x0002 | BF2_LEFT_TITLE. Put window title on left side of window. |
| 0x0004 | BF2_RIGHT_TITLE. Put window title on right side of window. |
| 0x0008 | BF2_OCVT_TAB_TO_SPACES. Convert TABs to spaces on output. |
| 0x0010 | BF2_OCVT_SPACES_TO_TABS. Convert spaces to TABs on output. |
| 0x0020 | BF2_OCVT_EMBEDDED. Convert inline spaces/tabs on output. |
| 0x0040 | BF2_OUTLINE. Enable outline display of buffer. |

| | |
|---|---|
| 0x0080 | BF2_ASCII_COLUMN. When in hex/ascii display mode, cursor is in the ASCII column. |
| 0x0100 | BF2_SPELL. Enable spell checking. |
| 0x0200 | BF2_OCVT_PRIVATE_TABS. When writing file, use buffer tab stops for conversions, not the default of 8 columns. |
| 0x0400 | BF2_LINE_NO. Display line numbers for buffer. |

### Return value

Flags associated with designated buffer, or -1 if the designated buffer does not exist.

### See also

set_buffer_flags()(pg. 176)., set_window_flags()(pg. 191).

## inq_buffer_title([bufnum])

### Description

Returns the title associated with the specified buffer, or current buffer if **bufnum** is not specified.

The buffer title can be set using the *set_buffer_title()* primitive and is used to set a preferential title to be used when the buffer is displayed in a buffer, rather than defaulting to the filename.

### Return value

Buffer title.

### See also

set_buffer_title()(pg. 178).

## inq_byte_pos([buf_id], [line], [col])

### Description

This primitive is used to return the byte offset from the start of the specified buffer.

If buf_id is not specified then the current buffer is used.

If *line* is not specified then it defaults to the current line.

If *col* is not specified then it defaults to the current cursor position.

### Return value

Byte offset from start of buffer.

## inq_called()

### Description

This macro exists so that macros can be written which need to differentiate between being called directly from the keyboard, or being called from another macro.

This is especially useful when writing a *replacement* macro.

### Return value

A string containing the name of the macro which called the current macro, or "" if the macro was called from the keyboard.

### See also

set_calling_name()(pg. 179).

## inq_changed_buffers([buf_id])

### Description

This function retrieves a list of buffers for which CRiSP has detected a state change on the underlying file, e.g. the file has been modified or deleted whilst editing it. If the *buf_id* parameter is specified then the reason detected for that buffer is changed. If *buf_id* is not specified then a list of buffer IDs is returned.

This function is used in the _file_error() callback macro to list the changed files/buffers in the popup dialog boxes.

### Return value

If *buf_id* is not specified then a list of changed buffers is returned.

If *buf_id* is specified then the reason for the file state change is returned.

### See also

inq_modified()(pg. 117).

## inq_char_map([win_id])

### Description

If win_id is specified, then that character map ID of the underlying buffer is returned (if it has one), otherwise the ID associated with the specified window. If win_id is not specified the current window is used.

This function is used to get the character map as the user perceives a buffer through a window.

### See also

create_char_map()(pg. 62)., set_buffer_cmap()(pg. 176)., set_window_cmap()(pg. 190).

### Return value

Returns the character map ID used to display the current.

## inq_clock()

### Return value

Returns the amount of CPU time used by CRiSP since the first call to inq_clock(), measured in microseconds.

## inq_cmd_line()

### Description

This macro returns the partially typed in text which the user is typing on the prompt line. It is used by various macros to perform command completion.

### Return value

Returns the field currently being typed in by user on the prompt line.

### See also

inq_command()(pg. 108).

## inq_command()

### Description

This macro can be used to find the name of the last macro which was invoked by virtue of it being bound to a keyboard key. Macros which start with a leading '_' are ignored.

### Return value

A string containing the name of the last macro called by the user.

### See also

inq_cmd_line()(pg. 108).

## inq_debug()

### Description

This macro returns the value of the CRiSP trace flags.

### Return value

An integer number representing various debugging bits.

### See also

debug()(pg. 67).

## inq_environment(name)

### Description

THIS FUNCTION IS PROVIDED FOR COMPATIBILITY VIA THE brief.cr MACRO FILE.

This function looks up '*name*' in the environment and returns the value of name. It is similar to the C function getenv().

### Return value

Returns a string which is the contents of the named environment variable, or the null string if the variable does not exist.

### See also

getenv()(pg. 97).

## inq_extension([buf_id], [mode])

### Description

This function is used to retrieve the file extension associated with a buffer. If the file in the buffer has no file extension then this macro will try and use the colorizer name as a candidate for the extension.

The idea behind this macro is to provide an easy and consistent way to get a file type from a filename for use with various setup macros.

| | |
|---|---|
| **buf_id** | Id of the buffer to use. If not specified then the current buffer is used. |
| **mode** | Specifies how the file type is to be determined.<br><br>If 0 or unspecified then returns the filename extension, or **"default"** if none is specified.<br><br>If 1, then returns the filename extension. If none is specified, then returns the colorizer language type. If none is specified then a blank string is returned.<br><br>If 2, then returns the colorizer language type. If this is not defined then uses the filename extension. If none is specified then a blank string is returned. |

### Return value

File extension or file type associated with the specified buffer.

### See also

filename_extension()(pg. 86)., inq_names()(pg. 118).

## inq_filesize([buf_id])

### Description

This primitive returns the size of the file written to disk when the specified (or current) buffer is written to disk. This size is an estimate only, and may be affected by, for example, tab/space conversions on outputs.

### Return value

Approximate size of output file in bytes.

## inq_filesystem()

### Description

This primitive can be used to retrieve attributes about filesystems on the system you are running. It is designed to make it easier to write portable macros which need to handle some of the esoteric features of different operating systems and filenames.

The return value is a set of flags with the following meanings:

| | |
|---|---|
| **FSYS_MONOCASE** | The file system supports monocased filenames. Files can be created in mixed case, but only one copy of a file may be present. This is typified by the Windows/95 and Windows/NT FAT file system using long filenames |
| **FSYS_DOS_DRIVES** | Filesystem supports a prefix to identify a drive, as in **C:** for instance under Microsoft operating systems. |

### Return value

Integer value containing flags which describe a filesystem.

## inq_font(type)

### Description

This primitive returns the name of the default font used for the editing area or the default font used for creating dialog boxes. It simply returns the value as previously set by the set_font() primitive.

If *type* is zero, then the current editing font is returned. If *type* is 1, then the current default dialog font is returned.

### Return value

String containing the default font string.

### See also

set_font()(pg. 181).

## inq_idle_default([flag])

### Description

This function returns the interval for the idle keyboard interval timer. The idle keyboard interval timer is a timer which is started after the last keystroke. If no keys are hit for *inq_idle_default()* seconds then the type 4 (REG_IDLE) registered macros are called.

The *flag* parameter can be used to return the idle timer value or the idle keystroke count. If flag is not specified or is zero then the idle timer is returned. If it is non-zero then the idle keystroke count is returned.

The default idle timer can be set by the '-i' command line switch, however the user usually configures it from the Setup->Autosave & Backups setup menu. If it is not set on the command line explicitly, then CRiSP uses a default value of 60 seconds.

### Return value

Returns the value of the idle keyboard interval timer (in seconds).

### See also

set_idle_default()(pg. 182)., register_macro()(pg. 163).

## inq_idle_time([flag])

### Description

This function returns the number of seconds since the last key was pressed. This macro is used by the autosave macros to see if it is time to save the editing buffers.

If *flag* is specified and is non-zero then the number of keys pressed since the last REG_IDLE trigger is returned instead. The autosave() macro checks both the timer and the key press count to see if either have expired before forcibly saving files.

### Return value

Returns the number of seconds since the last key was pressed, or if *flag* is not zero, then the number of keys pressed since the last autosave.

### See also

set_idle_default()(pg. 182)., inq_idle_default()(pg. 110)., register_macro()(pg. 163)., register_timer()(pg. 166).

## inq_index([int buf_id])

### Description

Returns the index into the current line. Since a buffer can contain virtual spaces, the current column position may not correspond with a physical character. This function returns how many characters into the line the cursor is.

If **buf_id** is omitted, then the current buffer is used.

### Return value

Returns the current character position in the current line.

### See also

inq_position()(pg. 119).

## inq_kbd_char([pos], [physical])

### Description

This macro provides a simple way of determining whether a character is available to be read from the keyboard without actually reading it. (See *read_char()*).

**pos** is an integer which if specified allows you to read the last key typed.

If ***physical*** is specified then CRiSP will destructively check to see if any keys have been hit on the keyboard and bypass the internal checks on the playback buffer or the push back buffer.

If you want to read raw scancodes, refer to the **inq_raw_kbd_char()** primitive.

### Return value

Returns non-zero if there is a character to be read from the keyboard; 0 if no character is available.

### See also

inq_kbd_flags()(pg. 111)., inq_keys_pressed()(pg. 112)., inq_raw_kbd_char()(pg. 120)., read_char()(pg. 161).

## inq_kbd_flags()

### Description

This macro can be used to read the current state of the keyboard modifier keys. This function only works in the GUI versions of CRiSP.

This macro is designed to be compatible with the BRIEF v3.1 function of the same name but care should be taken since the original BRIEF definition is very much Windows 3.x specific and some of the flag settings are not portable, e.g. to X Windows or Windows/95.

The flag settings are

| | |
|---|---|
| 0x01 | SHIFT key pressed. |
| 0x02 | SHIFT key pressed. (0x01 & 0x02 should correspond to the left and right shift keys, but on some platforms they cannot be distinguished so it is better not to rely on detecting either SHIFT key being pressed). |
| 0x04 | Control key is pressed. |
| 0x08 | Alt key is pressed. This may or may not include the right (AltGr) key. You should |

| | |
|---|---|
| | not rely on being able to detect the difference between the left and right Alt keys. |
| 0x10 | ScrollLock key pressed. |
| 0x20 | NumLock key pressed. |
| 0x40 | CapsLock key pressed. |

### Return value

Integer set of flags corresponding to the modifier keys which are currently pressed.

### See also

inq_keys_pressed()(pg. 112)., read_char()(pg. 161).

## inq_kbd_name()

### Description

This macro returns the current name of the key bindings in effect. This name is normally set by a **name** clause in the **.kbd** file.

### Return value

Returns string version of the keyboard name, e.g. **"crisp"** or "**cua".**

### See also

load_keyboard_file()(pg. 139)., set_kbd_name()(pg. 182).

## inq_keyboard()

### Description

This macro may be used to find out what the current keyboard identifier is so that it may be restored later.

See keyboard_push, keyboard_pop for further details on keyboard identifiers.

### Return value

Returns the identifier associated with the current keyboard.

### See also

keyboard_pop()(pg. 134)., keyboard_push()(pg. 135).

## inq_keys_pressed([total], [incremental])

### Description

This primitive can be used to retrieve a count of how many keys have been pressed so far.

The two arguments are optional, and if specified should be the names of integer parameters. The first (total) will receive a count of how many keys have been typed so far. The second (incremental) returns the number of keys pressed since the last idle time trigger.

The primary aim of this macro is to aid in macros which need to detect consecutive key presses (e.g. the functions 'home' and 'end' in misc.cr act differently for consecutive presses of the same key).

### Return value

Returns zero.

### See also

read_char()(pg. 161)., inq_kbd_char()(pg. 111).

## inq_keystroke_macro([km_id], [buf_id], [name])

### Description

This macro can be used to get the keycodes typed as a result of a keyboard macro so that they can be saved and restored.

If **km_id** is specified and is an integer, then it represents the keyboard macro id (as returned from a previous remember() macro). If not specified the last keyboard macro defined will be used.

If **km_id** is specified as a string, then **inq_keystroke_macro()** will return the keystroke identifier of the macro named.

**buf_id** is an optional integer variable which receives the buffer ID of a system buffer associated with the keyboard macro. This buffer contains the macros which were executed during the remember process (as opposed to the raw keystrokes). This buffer may be useful as the starting point when writing your own customised macros. Consult the **remember.cr** macro file to see how this buffer can be saved.

**name** is a string variable which gets the value of the user name associated with the keystroke macro.

### Return value

If **km_id** is an integer, then a string containing characters (in int_to_key() format) of characters making up a keyboard macro.

If **km_id** is a string, then the keystroke id of the named macro.

### See also

delete_keystroke_macro()(pg. 71)., list_of_keystroke_macros()(pg. 138)., load_keystroke_macro()(pg. 139)., playback()(pg. 151)., remember()(pg. 167).

## inq_line_col()

### Description

This primitive returns as a string the current contents of the Line:/Col: status area.

### See also

echo_line()(pg. 79).

## inq_line_flags([buf_id], [line_no])

### Description

This primitive is used to retrieve the line flags for the specified line. The line-flags are a set of 32 single-bit flags associated with a line. These can be used for any purpose although CRiSP uses some of them for its own purposes.

| | |
|---|---|
| **buf_id** | Buffer to retrieve flags from. If not specified, use the current buffer. |
| **line_no** | Line number to retrieve flags for. If omitted, current line is used. |

### Return value

Line number of line matching the search conditions. 0 if search fails.

### See also

find_line_flags()(pg. 90)., set_line_flags()(pg. 183).

## inq_line_length([bufnum], [win_id])

### Description

This command is used returns the length of the longest line in the specified buffer (or the current buffer if bufnum is omitted). The length corresponds to the column position the cursor would be in if the cursor were to be placed at the end of the longest line, i.e. it takes into account any embedded tabs and control characters, etc.

If the designated buffer has a marked region, then only those lines within the marked region are looked at to find the longest line.

If win_id is specified, then the maximum line length of the lines within the window will be returned.

### Example

The following example returns the length of the longest line in the current buffer in between the current line and the end of the buffer.

```
# include "crisp.h"


 .
int     length;

drop_anchor(MK_LINE);
save_position();
length = inq_line_length();
restore_position();
raise_anchor();
message("Longest line = %d cols", length);
```

### Return value

*inq_line_length()* returns the length of the longest line, or -1 if the specified buffer does not exist.

### See also

inq_line_length()(pg. 113).

## inq_lines([bufnum], [perc_flag])

### Description

This macro is used to inquire how many lines there are in the current (or specified buffer) -- or to inquire how far through the buffer the current line is (as a percentage).

If perc_flag is not specified or is non-zero then the percentage through the file is returned; otherwise the number of lines in the file is returned.

It is advisable where possible to avoid inquiring how many lines there are in the buffer because this will force CRiSP to read in the entire input file. The percentage figure does not involve reading the file in. This can significantly affect performance the first time this macro is called on buffers containing very large files.

### Example

Following example prints number of lines in buffer:

```
message("Lines = %d", inq_lines());
```

### Return value

Number of lines in the specified buffer - or the current buffer if bufnum is not specified. Returns -1 if buffer does not exist.

### See also

inq_lines()(pg. 114).

## inq_local_keyboard()

### Description

This function returns the identifier of the current local keyboard; this may be needed for example if the keyboard is to be changed or deleted.

Refer to *use_local_keyboard()* and the help section "Local Keyboards" for more information on what a local keyboard is.

### Return value

The keyboard identifier associated with the current local keyboard, or 0 if there is no local keyboard.

### See also

keyboard_pop()(pg. 134)., keyboard_push()(pg. 135)., use_local_keyboard()(pg. 212).

## inq_macro(macro, [primitive])

### Description

This macro is used to determine whether the passed macro name refers to a macro, builtin keyword or is undefined.

If **primitive** is not specified or is zero, then this then the function returns 1 if the macro exists, 0, if the specified macro does not exist but a primitive by that name exists, or -1 if no primitive or macro exists of that name.

If **primitive** is 1 then the function returns 1 if the macro exists or 0 if the macro does not exist.

If **primitive** is 2 then the function returns the object id associated with the static symbols of the macro file which defines the macro, or -1 if the macro does not exist.

This function can be used to detect erroneous calls to the execute_macro() primitive without the error actually appearing.

Passing the name of a CRiSP builtin will return 0 unless the built-in has been redefined by a replacement macro.

### Return value

Returns 1 if *macro* refers to a macro (or replacement macro); returns 0 if the specified macro does not exist.

Returns -1 if the specified macro does not correspond to a defined macro or a builtin primitive name. (*primitive* argument is set to 1).

### See also

autoload()(pg. 49)., load_macro()(pg. 140).

## inq_macro_files([macro])

### Description

This macro is used retrieve a list of the macro files which are loaded or the object identifier associated with the static symbols of a macro file.

If **macro**  is omitted, then a list of all macro files is returned. Some of these macro files may be **autoload** definitions, in which case you will only have access to the partial filename specified in the original autoload definition. If the macro file has been loaded, then the full path to the macro path is available.

If **macro** is specified, then this should be the name of a macro file (e.g. as previously returned by **inq_macro_files()**) and in this case the object id of the dictionary (symbol table) associated with the macro file is returned.

This macro is provided to aid in the implementation of the CRiSP macro debugger (debug.cr).

### Return value

List of filenames corresponding to loaded macros (if **macro** is not specified).

Object identifier of the symbol table for static variables if **macro** is specified and the macro file exists. 0 otherwise.

### See also

inq_macro()(pg. 115)., create_dictionary()(pg. 63).

## inq_mark_size()

### Description

This macro can be used to find out how many characters are in the currently marked region. This number is the string length of a string which would be necessary to hold the characters.

### Return value

The number of characters in the currently marked region.

## inq_marked([start_line], [start_col], [end_line], [end_col], [at_top_left], [dropped])

### Description

All parameters are optional integer variables. The first four parameters receive the boundary of the currently marked area (if any). The *at_top_left* parameter is used to determine whether the cursor is the start of end of a marked area.

For column blocks, the at_top_left parameter has a value corresponding to which corner of the column area the cursor is in.

If **dropped** is specified then it receives an indication of whether the current region has been dropped using the **end_anchor** primitive. A dropped region is one where you can move the cursor outside of the block.

### Return value

Returns 0 if buffer does not have a region set; otherwise returns current region type.

### See also

end_anchor()(pg. 82)., get_region()(pg. 96)., mark()(pg. 142).

## inq_menu_bar()

### Description

This function returns the window ID of the menu bar window, or zero if none is present. Macros should use this function when creating windows on the screen with absolute co-ordinates, e.g. to fill the screen.

Note: the menu bar temporarily disappears during a screen resize to aid in window co-ordinate calculations.

A menu bar window is created by setting the WF_MENU_BAR attribute of the window.

### Return value

Window ID of the menu bar or zero if none is present.

### See also

set_window_flags()(pg. 191).

## inq_message()

### Description

This macro returns any message or prompt which is currently displayed on the status line. It is used by the various command completion and abbreviation macros to see what command is currently executing.

### See also

inq_cmd_line()(pg. 108).

### Return value

Returns the message currently on the status line plus any type in from the user.

## inq_mode()

### Description

This macro returns the current state of the insert/overtype modes.

### Return value

Returns 1 if in insert mode; 0 if in overtype mode.

## inq_modified([bufnum])

### Description

This macro returns a value indicating whether the current buffer (bufnum omitted) or the buffer specified by bufnum has been modified.

If *bufnum* is a negative number then this primitive will return a count of how many modified buffers there are. This can be used by macros which wish to check if it is safe to exit.

### Example

The following example prints a message saying whether the buffer has been modified or not.

```
message("Buffer has %sbeen modified.",
inq_modified() ? "" : "not ");
```

### Return value

Returns non-zero if current buffer has been modified; 0 if buffer has not been modified (or has been written away).

If specified buffer id is < 0, then count of modified buffers is returned.

### See also

inq_changed_buffers()(pg. 108).

## inq_module()

### Description

Returns the module name associated with the calling macro. When a macro file is loaded, macros are associated with a module name. These module names are used to implement name spaces for static macro functions or to provide a way of creating a macro package split amongst source files.

The idea for this function is to allow macros to pass the name of local functions to macros defined externally to the calling module.

### Example

The following example shows how you can create a static function callback:

```
setup_callback()
{
        do_something(inq_module() + "mycallback");
}
void
do_something(string func
{
        execute_macro(func);
}
static void
mycallback()
{
}
```

By using the inq_module() function above, it does not matter whether the **do_something()** macro is defined in the current file or not.

### Return value

Name of current module, or a blank string. The return value can be concatenated with a macro function to provide a way of addressing **static** macro definitions from outside the defined file.

### See also

module()(pg. 146).

## inq_msg_level([level])

### Description

This primitive may be used to retrieve the current state of the message level flag. The message level flag is used to specify what level of error and informational messages are to be printed. This allows things like replacement macros to operate in a silent manner.

If *level* is specified then this will cause the current message level to be set to the specified value.

To see the meanings of the different levels, refer to set_msg_level.

### Return value

The current value of the message level flag (0-3).

### See also

set_msg_level()(pg. 183).

## inq_names([full_name], [ext], [buf_name], [buf_id])

### Description

This macro is used for determining the names of the file and buffer associated with the current buffer. If buf_id is specified, then that buffer is used to fetch the information rather than using the current buffer. Any of the parameters may be omitted. All the parameters are the names of string variables.

full_name is the full path name of the file associated with the buffer, and is the file which is written to when *write_buffer()* is called; ext receives the extension (if any) of full_name; buf_name is the title of the buffer - as displayed at the top of the window when the buffer is attached (see attach_buffer) to a window.

If *buf_id* is specified then the names extracted will be for the specified buffer rather than defaulting to the current buffer.

### Example

The following macro executes a file-dependent macro for setting up initialisations, etc.

```
string  ext;

inq_names(NULL, ext);
execute_macro(ext + "_init");
```

### Return value

Nothing.

### See also

filename_extension()(pg. 86).

## inq_objects([obj_id])

### Description

The inq_objects() primitive has a number of uses, and is used to inquire the values of objects or sub-object attribute values. If can be used to return a list of all window based objects (scrollbars, menus, buttons, etc) which have been created; it can be used to return the state of a specific dialog box; and it can also be used to return the object-id associated with the current DBOX_SCREEN (i.e. editing window).

If *obj_id* is not specified then a list of all user created objects is returned.

If *obj_id* is an integer >= 0, then a value of -1 is returned if the specified object does not exist, or a value > 0 if the value does exist. If *obj_id* is < zero then the object-id of the current input window will be returned.

If *obj_id* is a string then an integer will be returned corresponding to the object id of the named dialog box.

### Return value

If no arguments are specified then a list of integers corresponding to all window based objects created.

If obj_id is an integer > 0 then a return > 0 indicates the object still exists. If obj_id < 0 then the current input window object id is returned.

If obj_id is a string then the object-ID of the specified dialog box is returned, or -1 if no such dialog box exists.

### See also

change_object()(pg. 54)., delete_object()(pg. 71)., inq_window_system()(pg. 125)., popup_object()(pg. 152).

## inq_offset()

### Description

This primitive returns the current offset into the current line. (The offset is the number of characters to skip over assuming the cursor were at the start of the line to reach the current position).

The number returned does not take into account whether the cursor is on a virtual space or not (it assumes the cursor is at the start of a character position). If the cursor is beyond the end of the line then the line length is returned.

The current display mode is ignored thus allowing the calling macro to accurately determine where the cursor is after a display mode change (e.g. setting the BF_ANSI attribute does not change the physical column the cursor is in, but using this macro the cursor can be position nearest the appropriate character).

### Return value

Current offset into the current line.

### See also

inq_position()(pg. 119).

## inq_playback()

### Description

This macro can be used to determine if a keystroke macro is being recorded.

### Return value

Returns 1 if a keystroke macro is being recorded; zero otherwise.

### See also

playback()(pg. 151)., remember()(pg. 167).

## inq_position([line], [col], [old_line], [block])

### Description

This macro is used to find where the cursor is within the current buffer. line and col are the names of string variables which receive the current line and column number (both start at 1).

This function returns an indication of whether the cursor is past the end of the buffer.

The optional integer symbols *old_line* and *old_block* are used to retrieve the 'old' line number for the current line, and the block number associated with the current line. (Block numbers are for use internally and reliance on this facility is not a good idea to ensure future compatibility; this facility is provided for internal testing only).

### Return value

Returns 0 if cursor is not past end of buffer; otherwise it returns the number of lines past the end of the buffer.

### See also

inq_offset()(pg. 119).

## inq_process_level()

### Description

Returns the current depth of nesting due to calls of the *process()* primitive.

### See also

process()(pg. 154).

## inq_process_position([line], [col])

### Description

This function is similar to *inq_position()* but it returns the current cursor position for the underlying process. line & col are optional integer variables to receive the line & column, respectively.

The process position is used for output from the process; rather than inserting the output from the process where the users cursor is, a separate cursor is maintained instead. This is done because it allows the user to move around the buffer whilst the process is generating output without the process output being sprinkled through the buffer.

For more details of operation, refer to the shell.cr macro file.

### Return value

Returns -1 if current buffer is not attached to a process; returns 0 otherwise.

### See also

connect()(pg. 59)., inq_position()(pg. 119).

## inq_prompt()

### Description

This primitive returns a value indicating whether or not we are currently prompting the user on the command line.

### Return value

Returns zero if the user is not currently being prompted; non-zero otherwise.

### See also

get_parm()(pg. 95).

## inq_raw_kbd_char([pos], [physical])

### Description

This macro is similar to **inq_kbd_char()** and can be used to read the corresponding system dependent raw scan code for the last key pressed. Specify -1 for **pos.**

### Return value

Returns non-zero if there is a character to be read from the keyboard; 0 if no character is available.

### See also

inq_kbd_char()(pg. 111)., inq_kbd_flags()(pg. 111)., inq_keys_pressed()(pg. 112)., read_char()(pg. 161).

## inq_record_size([buf_id])

### Description

This primitive returns the record size attribute as set by the *set_record_size()* primitive, which is normally an inherited value.

### Return value

Current record size or zero if line mode behaviour is being used.

### See also

set_record_size()(pg. 184).

## inq_ruler([buf_id])

### Description

This returns a list of integers corresponding to the current ruler marker positions in the buffer. If **buf_id** is omitted then the current buffer is used.

### Return value

List of column marker indicators (integers).

### See also

set_ruler()(pg. 185).

## inq_scrap([last_newline], [type])

### Description

This macro returns various internal information about the scrap buffer.

**last_newline** is an optional integer variable which is used to indicate whether the scrap has a newline at the end of the last line of the buffer (used when inserting text via *paste()*). This is currently not implemented.

type is an integer variable which receives the type of the marked area which was most recently copied to the scrap.

### Return value

Returns the buffer identifier associated with the scrap.

## inq_screen()

### Description

Returns the screen ID with the current input focus (or the last accessed screen).

A screen is an instance of a main CRiSP editing work area (a top-level shell in X11 terminology).

### Return value

Current screen id.

### See also

set_screen()(pg. 185).

## inq_screen_flags([screen_id])

### Description

Returns the **DBOX_USER_FLAGS** associated with the current or specified screen. screen_id, if specified, should be a valid value as returned from the inq_screen() primitive.

The screen flags are user defined flags used to define the types of CRiSP windows in use, and this facility is mainly provided for callback routines to determine legal mouse activities in a callback routine.

### Return value

Screen flags.

### See also

set_screen()(pg. 185)., inq_screen()(pg. 121).

## inq_screen_size([lines], [cols])

### Description

lines and cols are optional integer variables. If specified they receive the current dimensions of the physical screen.

### Return value

The return value is a magic *cookie* indicating the screen type. Bit 0 is non-zero if this is a color screen of any type. Bits 1-7 are reserved. Bits 8-15 indicate the screen type. The actual value returned is designed to allow calling macros distinguish between different types of colored screens. At the present time this value is a magic cookie. The actual value, for example, is used as a magic *cookie* for the screen setup handling code.

### See also

set_screen_size()(pg. 186).

## inq_system([buf_id])

### Description

This macro can be used by other macros which need to look at arbitrary buffers, e.g. *buffer_list()*.

### Return value

Returns non-zero if the current or specified buffer is a system buffer; returns 0 if the current buffer is a normal buffer.

### See also

inq_buffer_flags()(pg. 105).

## inq_time([buf], [abs_time])

### Description

This primitive returns the time in seconds at which the specified buffer was last modified. If *buf* is not specified then it defaults to the current buffer. The time is in seconds from the time at which CRiSP was invoked.

**abs_time** is the optional name of an integer variable. If specified, then the absolute time at which the buffer was modified is returned in the variable **abs_time.**

Note that the change time may be non-zero even if the file has not been modified (as returned by *inq_modified()*). For example, after saving a modified file, *inq_modified()* will return FALSE (0) but the change time will still indicate the last time the file was modified.

This primitive is used by the autosave mechanism to avoid saving files on each autosave tick if the file has not changed since the last autosave.

### Return value

Returns -1 if buf does not refer to a valid buffer; 0 means the buffer has not been modified. Any other value is the time at which CRiSP was modified (measured from when CRiSP was started).

## inq_top_left([topline], [topcol], [win_id], [csrline], [csrcol], [buf_id])

### Description

This macro returns information about the window **win_id**, or the current window if **win_id** is not specified.

**topline** is the name of an integer variable receiving the line number of the top line in the window. **topcol** receives the column position of the left hand side of the screen.

**csrline** receives the current cursor line within the window and **csrcol** receives the column position of the cursor.

**buf_id** receives the buffer ID of the buffer attached to the window.

### Return value

Nothing.

### See also

set_top_left()(pg. 189).

## inq_views([buf], [win])

### Description

This macro can be used to determine if it is safe to delete a buffer (via *delete_buffer()*).

If *win* is specified then it should be the name of an integer variable to receive the window id of the first window displaying a buffer.

### Return value

Returns the number of windows which have the specified buffer on display. Returns 0 if buffer is not on display.

## inq_window([win_id])

### Description

This macro returns the id of the current window. This is useful for macros which wish to move the cursor to another window but when they have completed wish to restore the cursor to the original window.

If win_id is specified then this function returns -1 if the specified window does not exist or the value of win_id.

### Return value

An integer representing the identifier of the current window or -1 if there is no current window.

### See also

inq_window_buf()(pg. 123)., set_window()(pg. 190).

## inq_window_buf([win])

### Description

This macro can be used to find the buffer ID associated with a window. Given the buffer ID, the actual file name of the buffer can then be found.

### Return value

Returns the buffer ID associated with the current window or the window indicated by the win argument.

### See also

inq_buffer()(pg. 105)., inq_window()(pg. 123).

## inq_window_color([win_id])

### Description

This macro returns the background and foreground colors of a window. If win_id is not specified then the colors of the current window are returned.

If the window color is set to < 0, then the default background color will be used. If the color is set to >= 0 then the specified color will be used.

### Return value

The background color (as a number) ORed with the foreground color in the upper 16 bits of the value.

### See also

window_color()(pg. 215).

## inq_window_flags([win_id])

### Description

This macro returns the flags associated with the specified window, or the current window if *win_id* is not specified. The flags are defined as follows:

| **Flag** | **Description** |
|---|---|
| 0x0001 | WF_LINE_NO. Display line numbers in the window. |
| 0x0002 | This flag is used to allow the cursor to *disappear* from view. It is primarily supported to allow the GUI scrollbar to scroll around the current buffer without forcing the cursor to move. When this flag is set it will cause the screen to reflect the result of a previous call to the set_top_left() primitive even although the cursor may not be visible as a result of doing this.<br><br>See the register_macro() description of event 20 for information on the REG_KEY_ACTIVITY trigger which may be used to detect keyboard input so that the cursor-ghosting can be turned off. |
| 0x0004 | WF_NO_SHADOW. Used to turn off the shadow around a popup window. |
| 0x0008 | WF_SELECTED. Pretend window is the selected window (i.e. make the title appear as if the window were selected). |
| 0x0010 | WF_STICKY_TITLE. Dont change the window title when the buffer being viewed in it changes. |
| 0x0020 | WF_BUTTON. Window is acting as a button. |
| 0x0040 | WF_NO_BORDER. This flag indicates that the window is not to have any borders, irrespective of the *borders()* setting. |
| 0x0080 | WF_HILIGHT. Show currently hilighted region in this window even if this window is not the current window. |
| 0x0100 | WF_DELAY_UPDATE. Delay updating the window until the next refresh. Used to allow the current window contents to be *frozen* until a new keystroke, for example. This is used by the search highlighting macro to display the hilighted matched string, but will cause the hilight to automatically disappear on the next key press. |
| 0x0200 | WF_SYSTEM. A system window is similar to other windows but is ignored under certain circumstances, e.g. you cannot directly navigate into it using the change_window() primitive. It is mainly provided for the implementation of the character mode menu bar facility. |
| 0x0400 | WF_MENU_BAR. Window is a menu bar. CRiSP will take into consideration the dimensions of this window when the main window is resized. |
| 0x0800 | WF_HEX_MODE. Reserved for future implementation. |
| 0x1000 | WF_BLOCK_NUM. Display current internal block number for each line in window. The semantics of this bit are subject to change and are used for internal debugging purposes only. Programmers should avoid using or setting this bit. The functionality underlying this bit is subject to removal at a future date. |
| 0x2000 | WF_AUTO_WRAP. Lines too long to fit in the window will wrap around to the next line. (Not currently supported). |
| 0x4000 | WF_EOF. Display an [EOF] marker at the physical end of a buffer so its clearer to see exactly where the buffer ends. This attribute overrides the default DC_EOF setting for the |

display_mode() control flags.

### Return value

Display flags for the specified or current window.

### See also

inq_buffer_flags()(pg. 105)., set_buffer_flags()(pg. 176)., set_window_flags()(pg. 191).

## inq_window_info([win_id], [buf_id], [lx], [by], [rx], [ty])

### Description

This macro returns the information about the current window. All the parameters are optional names of integer variables to receive the information.

win_id receives the window id of the window.

### Return value

Returns zero if window is tiled, or non-zero if window is a popup.

## inq_window_size([lines], [cols], [right])

### Description

This macro is used to determine the dimensions of the current window. lines, cols & right are optional integer variables. lines and cols receive the number of lines and columns in the current window.

right receives the amount the current buffer is shifted within the window for sideways scrolling. If the current buffer hasn't been sideways scrolled, then right will receive 0.

### Return value

Nothing.

## inq_window_system([winsys])

### Description

This primitive is used to return a string identifying the windowing system under which CRiSP is currently running. This primitive returns the NULL string if CRiSP is not a windowing version of CRiSP.

The strings currently supported are: "Motif", "XView", "Windows/3.x".

This primitive is provided to allow macros to be customised for the different GUI environment in which they might run.

If the variable *winsys* is specified then a string representing the current windowing system version is returned, e.g. "Motif 1.2.3".

### See also

delete_object()(pg. 71).,  inq_objects()(pg. 118)., popup_object()(pg. 152).

## insert(expr, ...)

### Description

*insert()* is used for inserting text into a buffer. (The others are *self_insert()* and *paste()*). *expr* is a string expression which is to be inserted into the current buffer, as if typed by the user. Backslash-n characters are treated as new-lines.

If more than one argument is specified then *expr* is treated as a printf-style format specification. (I.e. %-sequences are interpreted; if no arguments follow expr, then % characters are inserted literally).

If *expr* evaluates to an integer then the character whose ascii value is 'expr' is inserted into the buffer. This is a special feature in that inserting 0x0a inserts a <Line-feed> character (^J) into the buffer without ending the current line. This feature is used by the quote() macro.

### Return value

Nothing.

### See also

format()(pg. 93)., insert_buffer()(pg. 126)., list_to_string()(pg. 139).

## insert_buffer(buf, expr, ...)

### Description

*insert_buffer()* is identical to the *insert()* primitive but in addition allows the buffer id to be specified rather than defaulting to the current buffer. *expr* is a string expression which is to be inserted into the current buffer, as if typed by the user. Backslash-n characters are treated as new-lines.

If more than one argument is specified then *expr* is treated as a printf-style format specification. (I.e. %-sequences are interpreted; if no arguments follow expr, then % characters are inserted literally).

If *expr* evaluates to an integer then the character whose ascii value is 'expr' is inserted into the buffer. This is a special feature in that inserting 0x0a inserts a <Line-feed> character (^J) into the buffer without ending the current line. This feature is used by the quote() macro.

### Return value

Nothing.

### See also

insert()(pg. 125)., list_to_string()(pg. 139).

## insert_mode([mode], [get_flag])

### Description

This macro allows the user to set and/or get the current value of the insert mode. The insert mode is applied when the user types in text. By default CRiSP comes in insert mode, i.e. characters typed in cause characters to the right of the cursor to be shifted over. Overtype mode causes the character under the cursor to be deleted and replaced by the inserted characters.

ins_mode is zero if overtype mode should be turned on, and non-zero if insert mode should be turned on.

If ins_mode is omitted, then the current value is toggled.

Note that the insert/overtype mode only affects *self_insert()* not *insert()* or *paste()*, etc.

If the display terminal is not able to change cursor shape, then the echo status line will display the current status of the insert/overtype mode. (Note that it has to compete with other values on the display line so it may disappear, for example, if percentage through file is turned on. (See *echo_line()*).

If the parameter *get_flag* is specified then the current insert mode setting only will be returned. The 1st argument will be ignored.

### Example

The following example prints the state of the insert/overtype mode without changing the mode.

```
int     mode;

mode = insert_mode();

/* Toggle mode back to original value. */
insert_mode();

message(mode ? "Insert mode" : "Overtype mode");
```

### Return value

Returns non-zero if the previous value was insert mode; returns zero if the previous value was overtype mode.

## insert_object(obj_id, name_id, flags, ...)

### Description

*insert_object()* is used to modify an existing dialog box object by adding in a new set of widget objects, before or after the specified **name_id**.

### Return value

-1 if specified **obj_id** does not exist.

-2 if **obj_id** is not a dialog box.

## insert_process(expr, [num])

### Description

*insert_process()* is similar to *insert()* except the inserted text is sent to an attached process instead of being inserted into the text buffer. No auto echo is performed - it is the calling macros responsibility to echo any input sent to the process.

If num is specified, then the string, expr, is inserted num times. If omitted, it defaults to 1.

### Return value

Nothing.

## inside_region([win_id], [line], [col])

### Description

This function can be used to test whether the specified **line** and **col** arguments correspond to a position before, inside, or after the currently selected region of a window.

**win_id** is the window identifier of the window to check. If not specified then the current window is used.

If **line** or **col** argument are not specified, then the current line and/or column number are used.

### Return value

| | |
|---|---|
| 0 | no region selected. |
| 1 | cursor is before the region. |
| 2 | cursor is inside the region. |
| 3 | cursor is after the region. |
| 4 | cursor is to the left of a column region. |
| 5 | cursor is to the right of a column region. |

### See Also

drop_anchor()(pg. 78).

## int var1, var2, ..;

### Description

This macro is used to define local variables which are to contain only integer values. The variables defined are local variables, and are destroyed when the macro executing this declaration terminates.

Integers are two's complement integers, 32-bits in length on all machines.

## int_to_key(key)

### Description

This function takes an integer argument, which is a key-code, e.g. as returned by *read_char()*. The integer key-code is converted to a string in the canonical form understood by assign_to_key. This allows macros to be written portably, rather than returning the raw key-code.

*int_to_key()* and *key_to_int()* are inverses of each other.

### Return value

Returns a string in the format of *assign_to_key()*.

### See also

key_to_int()(pg. 134).

## ipc_accept(chan_id)

### Description

This function is used to accept an incoming connection from another task on the network. It can only be used with the IPC_TCP type of IPC mechanism. It should only be called when the existing channel is indicated as being readable, either via a trigger for the IPC_READ_TRIGGER or some other indication. If the IPC channel is marked in non-blocking mode then you can attempt to accept any outstanding connections at any time, but an error will be returned unless the operation is successful.

On a successful return a new IPC channel is returned.

### Return value

-1 on an error or >= indicating a channel identifier.

### See also

ipc_close()(pg. 128)., ipc_create()(pg. 128)., ipc_read()(pg. 130)., ipc_write()(pg. 132)., ipc_status()(pg. 131)., ipc_register()(pg. 130)., ipc_unregister()(pg. 131).

## ipc_close(chan_id)

### Description

Closes a previously opened IPC channel. Note that any underlying processes are not killed, so if you want an underlying process to be killed you will need to explicitly send a signal via the kill() primitive.

The process ID of an associated process can be obtained by the ipc_status() primitive.

### Return value

-1 on error. Zero on success.

### See also

ipc_create()(pg. 128)., ipc_read()(pg. 130)., ipc_write()(pg. 132)., ipc_accept()(pg. 128)., ipc_status()(pg. 131)., ipc_register()(pg. 130)., ipc_unregister()(pg. 131).

## ipc_create

## ipc_create(IPC_ANON_PIPE | flags, program)
## ipc_create(IPC_NAMED_PIPE | flags, pipein, pipeout, program)
## ipc_create(IPC_TCP | flags, "host:port")
## ipc_create(IPC_TCP | flags, "port")
## ipc_create(IPC_UDP | flags, "host:port")
## ipc_create(IPC_UDP | flags, "port")
## ipc_create(IPC_PTY | flags, program)
## ipc_create(IPC_DDE | flags, "SERVICE:TOPIC")

### Description

The *ipc_create()* primitive is used to create an external IPC connection. This function is a more functional version of the connect() and disconnect() primitive and can be used to create a connection to an external task or server application irrespective of any existing buffer.

A number of common IPC mechanisms are supported. In all cases, the first parameter is an integer value specifying a set of flags which control the creation of the IPC connection. The value of this expression must include one of the IPC_ #define's listed above to indicate the type of connection required.

The remaining arguments are dependent on the connection type and are discussed below.

The function returns a *connection identifier* (like a file descriptor) which is used in subsequent IPC functions (such as *ipc_read()*, *ipc_write()*, etc). A macro can be invoked automatically when there is data to read or write on the connection.

The number of connections which can be created is operating system dependent and is limited by the hard file descriptor limit on your system.

**IPC_ANON_PIPE**
This mechanism uses anonymous pipes and allows you to spawn a subprocess to which you can read and write using the *ipc_read()* and *ipc_write()* primitives. The *program* argument is a string corresponding to the name of the program to run. This argument is passed to the shell so you rely on your SHELL environment variable being honoured and you should be able to use the normal shell wild-card metacharacters.

**IPC_NAMED_PIPE**
This mechanism is similar to the IPC_ANON_PIPE mechanism but allows you to use two named pipes for I/O. The second and third arguments correspond to the names of the named-pipes to be used for input and output respectively. If either of these is **NULL** then the current *stdin* or *stdout* will be used. When the sub-process is created the output named-pipe will be used for *stdout* and *stderr*.

**IPC _TCP** This mode allows you to create a TCP connection. Depending on the second parameter, you can either create a server connection or a client connection. The second parameter is used to encode the port to connect to, for a client connection, or the port to bind to for a server connection. For a server channel, the string should simple be a port number or a service name listed in the /etc/services file. For a client connection, the string should contain a host and port in the format: "host:port", where port can be a number of an entry listed in the /etc/services file. The hostname part can either be a hostname or an IP address in standard dotted notation. The port number must be a number or a service name as listed in /etc/services.

If a server connection is created then the return from the function is immediate and you will be notified of incoming connections by the IPC_READ_TRIGGER. When the IPC_READ_TRIGGFER is fired you can issue an *ipc_accept()* function call to accept an incoming connection. (*ipc_accept()* creates a brand new connection for you).

For a client connection, the function will block until a connection is made or an error will be returned indicating the reason for the error (you can access the global variable *errno* to retrieve the system dependent reason for the error).

**IPC_UDP** This mechanism is similar to the TCP mechanism described above but creates a connectionless end-point.

**IPC_PTY** This mechanism is similar to the IPC_ANON_PIPE mechanism described above but utilises a PTY for input and output. This IPC mechanism is most suited to processes which need to run from a pseudo-tty but would otherwise break if run from a pipe.

**IPC_DDE** This mechanism is only available on Microsoft Windows platforms. This mechanism provides a simple way to access interprocess communication via the DDE protocol. Reading from a DDE connection is not currently supported, although writing is.

When using an IPC mechanism, you can read and write data with the ipc_read() and ipc_write() functions. You can create callback (registered macro) to indicate when data is available to be read, so you can write non-blocking macros.

An EOF is indicated on an IPC channel by reading a zero length string on the channel.

When you have finished with a channel you should *ipc_close()* the channel.

In addition to the basic IPC mode types described above, you can OR in the following flags:

IPC_NON_BLOCKING
: Put channel into non-blocking mode. In this mode, attempting to read from the IPC channel will not block but will return a null string. Also, when connecting to a remote service using IPC_UDP or IPC_TCP the macro will not block and no failure of the connection will occur until an attempt is made to read/write on the connection. This mode of working is best suited for the callback mechanism.

### Return value

Less than zero on failure. For certain error conditions you can examine the global variable *errno* to discover the system dependent reason for the failure.

| | |
|---|---|
| -2 | Missing parameter. |
| -3 | Out of system resources. |
| -4 | Socket couldn't be allocated. |
| -5 | bind() system call failed. |
| -6 | connect() system call failed. |

Greater than or equal to zero on success, in which case this is the channel identifier for use in the other IPC function calls.

### See also

ipc_close()(pg. 128)., ipc_read()(pg. 130)., ipc_write()(pg. 132)., ipc_accept()(pg. 128)., ipc_status()(pg. 131)., ipc_register()(pg. 130)., ipc_unregister()(pg. 131).

## ipc_read(chan_id, [len])

### Description

This primitive is used to read data from an IPC connection. All IPC connections operate in a non-blocking mode unless explicitly set to be blocking (see ipc_status()).

If no data is available to be read, then a zero length string will be returned, as will an EOF indication. Therefore you should avoiding reading data unless data is actually available. You can use *ipc_register()* to register a callback when data is available to be read.

If *len* is not specified then as much data will be read as possible. If *len* is specified that at most that number of characters will be read.

NOTE: If you read or write binary data (i.e. data containing embedded NULLs) then the data may be truncated.

### Return value

String of data read.

### See also

ipc_close()(pg. 128)., ipc_create()(pg. 128)., ipc_write()(pg. 132)., ipc_accept()(pg. 128)., ipc_status()(pg. 131)., ipc_register()(pg. 130)., ipc_unregister()(pg. 131).

## ipc_register(chan_id, trigger, macro)

### Description

The *ipc_register* primitive is used to register callback routines to handle asynchronous I/O on an IPC connection. Because CRiSP is a full blown GUI application it is generally unwise to wait for data. By using the registered callbacks you can arrange for a macro to be called only when the specific trigger is available.

The *chan_id* parameter refers to an I/O channel previously opened via *ipc_create()*. *trigger* indicates under what circumstance the macro can be called. *macro* is the name of a macro to call. The macro is called with one argument - the channel ID of the channel. (This allows the same callback macro to be used for multiple connections).

The following is a list of the callback reasons:

IPC_TRIGGER_READ. Data is available to be read via *ipc_read()*. If a read returns an empty string, then the underlying connection has been lost and the calling macro should treat this as an EOF indication.

IPC_TRIGGER_WRITE. Data can be written to the underlying channel without blocking. It is fairly rare to use this trigger unless you are planning to send large amounts of data.

IPC_TRIGGER_EXCEPTION. Reserved for future use.

IPC_TRIGGER_PROCESS_DEATH. This can be used to intercept child process death (only applicable to the IPC_ANON_PIPE, IPC_NAMED_PIPE and IPC_PTY connection types). It can be used to recognise the death of the process attached to the endpoint of an IPC connection even when no data is available to be read. In practise you may get an IPC_TRIGGER_READ callback with an empty string rather than having this callback invoked.

A callback can be removed by calling *ipc_unregister()*.

NOTE: You can register multiple callbacks for the same trigger condition, in which case the callbacks will be called in order. This of course may not be meaningful, e.g. reading data would cause the next callback to hang until more data is available, but it is worth bearing in mind.

### Return value

Returns -1 on error, zero on success.

### See also

ipc_unregister()(pg. 131)., ipc_create()(pg. 128)., ipc_close()(pg. 128)., ipc_read()(pg. 130)., ipc_write()(pg. 132)., ipc_status()(pg. 131)., ipc_accept()(pg. 128).

## ipc_status(chan_id, [flags], [pid])

### Description

This function can be used to retrieve and/or set the value of the IPC status flags on the connection.

If *flags* is specified then the flags on the channel will be set to this value. (You can retrieve the current value of the flags by calling *ipc_status()* without specifying a *flags* parameter).

If *pid* is specified and is the name of an integer variable then it will receive the process-id of the associated sub-process for the channel. This field is only valid for the IPC_ANON_PIPE, IPC_NAMED_PIPE and IPC_PTY IPC mechanisms.

### Return value

Current value of the flags associated with the channel or -1 on an error.

### See also

ipc_accept()(pg. 128)., ipc_close()(pg. 128)., ipc_create()(pg. 128)., ipc_read()(pg. 130)., ipc_write()(pg. 132)., ipc_register()(pg. 130)., ipc_unregister()(pg. 131).

## ipc_unregister(chan_id, trigger, macro)

### Description

The *ipc_unregister* primitive is used to remove a previously registered callback issued with the *ipc_register()* function. In order to remove a callback, exactly the same arguments should be passed to the ipc_unregister() primitive.

### Return value

Returns -1 on error, zero on success.

### See also

ipc_register()(pg. 130)., ipc_create()(pg. 128)., ipc_close()(pg. 128)., ipc_read()(pg. 130)., ipc_write()(pg. 132)., ipc_status()(pg. 131)., ipc_accept()(pg. 128).

## ipc_write(chan_id, string)

### Description

This primitive is used to write data to an IPC connection. All IPC connections operate in a non-blocking mode unless explicitly set to be blocking (see ipc_status()).

### Return value

-1 on an error. Number of bytes written.

### See also

ipc_accept()(pg. 128)., ipc_close()(pg. 128)., ipc_create()(pg. 128)., ipc_read()(pg. 130)., ipc_status()(pg. 131)., ipc_register()(pg. 130)., ipc_unregister()(pg. 131).

## is_directory(path)

### Description

Test to see if the specified path name is a directory or not.  Be careful about using this primitive as a boolean predicate function because if the specified file or directory does not exist, then an error code (-1) is returned.

### Return value

Returns -1 if path doesn't exist or on an error; 0 if path is not a directory; 1 if path is a directory.

### Example

```
string file = getenv("HOME") + "/.file";

if (is_directory(file) > 0)
    message("File is a directory: %s", file);
```

## is_integer(var)

### Description

This macro can be used to test the data type of a polymorphic variable, for example when walking down a list.

### Return value

Returns 1 if var is an integer variable; zero otherwise.

### See also

is_list()(pg. 132)., is_null()(pg. 132)., is_string()(pg. 133).

## is_list(expr)

### Description

This macro can be used to test the data type of a polymorphic variable.

### Return value

Returns 1 if var is an list variable; zero otherwise.

### See also

is_integer()(pg. 132)., is_null()(pg. 132)., is_string()(pg. 133).

## is_null(expr)

### Description

This macro can be used to test if a particular element of a list actually exists (i.e. to avoid walking off of the end of a list).

### Return value

Returns 1 if var is contains a null value; zero otherwise.

### See also

is_integer()(pg. 132)., is_list()(pg. 132)., is_string()(pg. 133).

## is_pipe_filename(filename)

### Description

This macro returns 1 if the specified filename refers to a pipe (i.e. it starts with a '**!**' or '**|**' character).

### Return value

Returns 1 if filename is a pipeline command or zero

### See also

is_url_filename ()(pg. 133).

## is_string(expr)

### Description

This macro can be used to test the data type of a polymorphic variable.

### Return value

Returns 1 if var is a string variable; zero otherwise.

### See also

is_integer()(pg. 132)., is_list()(pg. 132)., is_null()(pg. 132).

## is_url_filename(filename)

### Description

This macro returns 1 if the specified filename is a universal resource locator filename, e.g. it can start with **ftp:** or **http:**.

### Return value

Returns 1 if filename is filename in the form of a URL, zero otherwise.

### See also

is_pipe_filename ()(pg. 133).

## key_list([kbd_id], [self_insert], [buf_id])

### Description

This macro is used to get a list of all key-bindings in a keyboard map. The macro returns a list of strings where the even elements in the string are the key names, and the odd elements are the macro names associated with those keys.

Upto two keyboard maps can be specified. If **buf_id** is specified, then the local keyboard assigned to that buffer is used. If not specified then the local keyboard map of the current buffer is used.

If **kbd_id** is omitted then the current keyboard is used. If specified then that keyboard is used.

If **self_insert** is specified and is non-zero, then the keys for the self-inserting keys are returned as well. The default is to omit them.

When scanning a local and a global keyboard, and an entry is defined for a keycode in the local and global map, then only the local keyboard entry is returned.

This primitive is defined so that the help macro can display a list of all valid keys currently mapped in both the current local and global keyboard maps.

### Return value

Returns a list containing pairs of strings where the first string in each pair is the key name, and the second string is the macro assigned to that key.

### See also

inq_keyboard()(pg. 112).

## key_to_int(key, [raw])

### Description

This function is the inverse of int_to_key. *key* is a string to be converted to an internal keycode number. If *raw* is not specified or is zero, then the internal keycode associated with the key sequence is returned. In this case, key should be a valid key-sequence which can be passed to *assign_to_key()*.

If *raw* is specified and is non-zero, then *key* is a raw key stroke, as entered by a function key on the keyboard. In this case, the keycode assigned to that key is returned.

### Example

The following example returns the keycode for the F1 function key.

```
key_to_int("<F1>");
```

The following example returns the keycode for the key whose raw key sequence is that specified. If this keycode corresponds to the <F1> key, then the internal keycode for the <F1> key will be returned:

```
key_to_int("x1B[224z");
```

### Return value

Returns an integer representing the internal key code assigned to the specified key sequence, or -1 if the sequence does not correspond to an valid key definition.

### See also

assign_to_key()(pg. 47)., int_to_key()(pg. 128).

## keyboard_flush()

### Description

keyboard_flush can be used to flush any pending input.

### Return value

Nothing.

## keyboard_pop([save])

### Description

This macro is used to pop the current keyboard off the keyboard stack; **save** is an integer expression. If it is non-zero, then the current keyboard is saved for later use (by keyboard_push). If it is zero, then the keyboard map is discarded.

### Example

See *keyboard_push()* for an example of creating a temporary keyboard map.

### Return value

Nothing.

### See also

keyboard_push()(pg. 135).

## keyboard_push([kbd_id])

### Description

keyboard_push is used to create a new keyboard table. The current keyboard map is saved on a stack, and can be retrieved by using keyboard_pop.

**kbd_id** is an integer expression. If it is omitted, then a new keyboard map is pushed on the stack, and by default all the keys in the map are assigned to *nothing()*, i.e. pressing them has no effect. If kbd_id is specified, it should be a value which was obtained from a previous *inq_keyboard()* call. This will push the keyboard whose identifier is kbd_id on top of the stack.

A keyboard map is a table of all possible keys, together with the macros that are bound to them. By default CRISP comes up with the usual editing keys, but some macros need to temporarily bind macros to keys for their operation. For example, the buffer_list macro creates a new keyboard and binds various keys to it, so that it can intercept the users requests to delete and write away buffers.

Having keyboard maps saves the macro writer from having to write complicated, and CPU expensive keyboard parsers.

### Example

The following example shows how to create a temporary keyboard mapping for use within a macro.

```
int     curkbd;

curkbd = inq_keyboard();

keyboard_push();
assign_to_key("<Alt-H>", "help");
..
..
process();
keyboard_pop();

keyboard_push(curkbd);
```

### Return value

Nothing.

### See also

keyboard_pop()(pg. 134).

## keyboard_reset([kbd_id])

### Description

This primitives clears out all key bindings for the current keyboard, or the specified keyboard id.

### Return value

1 on success. 0 if the specified keyboard does not exist.

### See also

keyboard_pop()(pg. 134)., keyboard_push()(pg. 135).

## keyboard_typeables()

### Description

keyboard_typeables is a quick method of setting up a keyboard map so that the usual insertable characters (ASCII space to ASCII tilde, carriage return, and line-feed) do their default actions of being insertable.

It is a shorthand way of doing the following:

```
assign_to_key("A", "self_insert");
assign_to_key("B", "self_insert");
.
.
```

It is useful after creating a new blank keyboard.

### Return value

Nothing.

### See also

assign_to_key()(pg. 47)., keyboard_push()(pg. 135)., keyboard_pop()(pg. 134).

## kill(pid, sig)

### Description

This primitive can be used to send signals to processes on the system. The arguments are machine dependent, and on those platforms that support process signals, then the first parameter is the process ID, and the second is the signal number to send the signal to.

### Return value

Machine dependent. Usually zero on success, -1 on error. If an error occurs the error reason is in the global integer variable errno.

### See also

send_signal()(pg. 174).

## ldexp(x)

### Return value

Returns x*2^n as a floating point value.

## left(num)

### Description

Moves the cursor one character to the left or num characters to the left, if specified. num may be negative.

If the cursor is moved past the beginning of the current line, then the cursor wraps around to the end of the previous line.

### Return value

Nothing.

## length_of_list(list_expr)

### Description

*length_of_list()* returns the number of atoms in a list; it only counts the top level atoms. For example, if (car list_expr) is a list expression itself, then this only counts as one atom, not (length_of_list (car list_expr)).

### Return value

The number of atoms in the list.

## line_add([int buf_id], [int line_no], [int n])

### Description

This function is used to perform line number arithmetic. At it's simplest it simply returns **line_no + n**. However, if outlining is in effect, i.e. some lines are hidden on screen, then determining line_no+n is quite tedious. This macro primitive handles that by taking into account invisible lines if outlining is enabled.

| | |
|---|---|
| **buf_id** | Buffer to refer to. If omitted the current buffer is used. |
| **line_no** | Starting line number. Current line number of buffer if omitted. |
| **n** | Number to add or subtract (if negative). |

### Return value

**line_no + n** if successful, or -1 if specified buffer does not exist.

## list var1, var2, ..;

### Description

This macro is used to define list variables. List variables may be defined as global (see *global()*). List variables are given an initial value of the NULL list (i.e. *length_of_list()* returns 0).

Lists are data types similar to arrays, except each element of a list (referred to as an atom) may have any of the CRiSP primitive data types (including list), and lists can be expanded at the end simply by concatenating entries on the end (see *append_list()* and *put_nth()*).

## list_extract(lst, start, end, incr)

### Description

This macro can be used to extract alternate entries from a list. The **start, end** and **incr** arguments control the data to be extracted.

### Return value

New list with the extracted entries.

## list_of_bitmaps(file, [width], [height])

### Description

This primitive lists the pixmap icons available in a .**xpl** archive library file. *file* is the path to the pixmap library file. If *width* and/or *height* are specified and are non-zero, then only those pixmaps matching the specified dimension will be returned.

### Return value

List containing names of pixmaps, which can subsequently be used as argument to a **DBOX_TOOL_BUTTON** object type.

## list_of_buffers([sys_bufs])

### Description

This primitive is used to return a list of all buffer identifiers. By default only non-system buffers are returned in the list. If *sys_bufs* is specified and is non-zero then all system buffers will be reported as well.

This primitive makes it easy to write macro code which needs to iterate over all buffers without having difficulty maintaining the current buffer status.

### Return value

List of integers corresponding to all loaded buffers.

### See also

list_of_windows()(pg. 139).

## list_of_dictionaries([non_empty])

### Description

Returns a list of integer *dictionary* identifiers. Dictionaries are explicitly created via the create_dictionary()(pg. 63). primitive or implicitly by various other primitives (e.g. create_buffer(pg. 62)., create_object(pg. 64).).

By default, a list of all dictionaries are returned. If **non_empty** is specified and is non-zero, then only dictionaries containing at least one symbol are returned.

### Return value

List of all defined dictionaries.

### See also

create_dictionary()(pg. 63)., dict_delete()(pg. 73)., dict_list()(pg. 74)., get_property()(pg. 95)., set_property()(pg. 185).

## list_of_keyboards()

### Description

Function which returns a list of integers corresponding to the keyboard mappings currently defined.

Mainly of use for writing diagnostic macros.

### Return value

List of integers corresponding to all defined keyboards.

### See also

keyboard_push()(pg. 135)., keyboard_pop()(pg. 134).

## list_of_keystroke_macros()

### Description

This primitive is returns a list of integers corresponding to all the currently defined keystroke macros..

### Return value

List of integers corresponding to all defined keystroke macros.

### See also

delete_keystroke_macro()(pg. 71)., inq_keystroke_macro()(pg. 113)., load_keystroke_macro()(pg. 139).

## list_of_objects(obj_id, parent)

### Description

Returns a list of children of the specified parent. *parent* is a string name corresponding to the parent object, e.g. a DBOX_CONTAINER widget.

## list_of_screens([mask], [value])

### Description

This primitive returns a list containing all the valid screen id's. A screen is an instance of a main CRISP window (e.g. under X11, a top level window, containing a menu bar and scrollbar).

The optional arguments *mask* and *value* are used to get a list of only certain types of screens. Both arguments must be specified to have any effect. When both arguments are given, then the user flags associated with a screen are ANDed with the mask value and the screen ID will only be returned in the resultant list if this masked value is equal to the value of the *value* parameter.

This primitive is used when a peel off window is terminated, e.g. via Alt-X to determine whether we are the last *top-level* CRISP window and hence to terminate CRISP or simply the current peel off window.

### Return value

A list of the screen id's or NULL if no screens have been created.

### See also

create_object()(pg. 64).

## list_of_windows()

### Description

This primitive is used to return a list of all windows on the current screen.

This primitive makes it easy to write macro code which needs to iterate over all windows without having difficulty maintaining the current window status.

### Return value

List of integers corresponding to all windows on current screen.

### See also

list_of_buffers()(pg. 137)., list_of_screens()(pg. 138).

## list_to_string(fmt, list, [incr])

### Description

This function can be used as a convenient way to format all the strings in the specified list, and is a useful alternative to the following construct:

```
string  s, tmp;
int     i;

for (i = 0; i < length_of_list(lst); i++) {
    sprintf(tmp, fmt, lst[i]);
    s += tmp;
}
```

### Return value

A new string corresponding to the format specification applied to the list.

### See also

printf()(pg. 153)., sprintf()(pg. 200)., insert()(pg. 125).

## load_keyboard_file(filename)

### Description

This function is used to load a file containing a set of key bindings. CRISP stores key-bindings in ***.kbd** files in the **etc/** distribution directory.

### Return value

-1 on error; 0 on success.

### See also

assign_to_key()(pg. 47)., inq_kbd_name()(pg. 112).

## load_keywords(filename)

### Description

This function is used to get CRISP to load a colorizer file in the .KWD file format.

### Return value

-1 on error; 0 on success.

## load_keystroke_macro([id], [name], [str])

### Description

This macro provides a way of creating or modifying keyboard macro without using the keyboard. This macro is designed to allow user macros to load macros from external files.

**id** is a keystroke macro identifier. If its not specified, then a new keystroke macro will be created; otherwise the specified macro will be modified.

**name** is an identifier for the keystroke macro which is designed to allow the user associate a meaningful name with the keyboard macro.

**str** is a string in a similar format returned by inq_keystroke_macro() which is parsed and stored as a keystroke macro.

### Return value

Keystroke macro ID associated with the keystroke macro.

### See also

delete_keystroke_macro()(pg. 71)., playback()(pg. 151)., remember()(pg. 167).,
inq_keystroke_macro()(pg. 113).

## load_macro([file])

### Description

This macro is used to load a .m or .cm macro file into CRISP. file is a string expression which specifies the name of the macro file. If omitted, it is prompted for.

Macros in source form (.m files) can be loaded with this macro.

The extension can be omitted, in which case CRISP looks for either file.m or file.cm If the extension is supplied, then only a file matching the extension will be searched for. Otherwise CRISP looks for a .cm file before looking for a .m file.

If file does not contain a path specification, then CRISP searches all the directories in the CRPATH environment variable (or uses an internal default if CRPATH is not specified).

When searching for a .m or .cm it searches the directories in order, i.e. if a .cm file exists in a later path than the .m then the .m file will be loaded first.

When loading .cm files, CRISP checks the version number of the file to see if it is compatible with the current version of CRISP, and issues an error message if the version number does not agree.

If a .m file has a syntax error then the loading of the file is aborted with an error message displayed.

If a macro called '**_init**' exists in the macro file being loaded, then that macro is executed immediately. This facility exists to allow macros to perform initialisation functions.

### Return value

Returns 1 if macro file was successfully loaded; 0 otherwise.

## log(x)

### Return value

Returns the natural logarithm of x, ln(x). x is a floating point value which must be > 0.

### See also

log(pg. 140)., exp(pg. 85).

## log10(x)

### Return value

Returns the base-10 logarithm of x, x is a floating point value which must be > 0.

### See also

log10(pg. 140)., exp(pg. 85).

## lower(string)

### See also

upper

### Return value

Returns a copy of string with all upper case letters mapped to their lower case equivalents.

## ltrim(string)

### Description

This function is used to remove leading characters from the specified string. The default is to remove all tabs, spaces and newline characters. If delim is specified, then all characters in the delim string are removed from the beginning of string.

### See also

trim(pg. 210)., compress(pg. 59).

### Return value

Returns a copy of string with all leading white space characters removed. (spaces, tabs and newlines).

## (macro name list)

### Return value

Result of last statement executed in the macro.

### Description

(macro) defines a new macro. It is normally only processed when loading a .m or .cm file. The macro is given the name 'name' and consists of the executable statements in the list, list.

If a macro is given the special name _init, then it is automatically executed when the macro file is loaded.

If the *name* parameter is a two-element list consisting of an integer number and a macro name, then the macro is created as a static macro. Static macros are only visible to macros defined in the same file, and cannot normally be accessed by any other file/macro. This special syntax is generated by the CRUNCH compiler automatically for static macros.

THIS IS A LOW LEVEL PRIMITIVE NOT AVAILABLE IN THE CRUNCH LANGUAGE. THE CRUNCH COMPILER AUTOMATICALLY PRODUCES MACRO DEFINITIONS FOR EACH DEFINED CRUNCH FUNCTION.

## macro_list([wildcard])

### Description

This macro is similar to *command_list()* but simply returns a list of all the currently defined macros. If **wildcard** is specified, then this allows you to specify a filename style wildcard for returning a list of matching macro names.

### Return value

A list of strings containing the names of all currently defined macros (not CRISP keywords), in alphabetic order.

### See also

command_list()(pg. 58).

## make_list(expr1, expr2, ..)

### Description

This macro is used to create a new list constructed from the argument list. Each argument is evaluated and appended to the next list. Contrast this with quote_list() which does not evaluate the arguments.

### Example

```
int i = 4, j = 5;
list l = make_list(i, j);

/* l = {4 5} */
```

### See also

quote_list()(pg. 157).

### Return value

List constructed from arguments.

## make_local_variable(var1, var2, ..)

### Description

This macro is used to make existing local variables into *buffer* local variables. CRISP supports the concepts of three storage classes for variables - local, which go out of scope when the current macro terminates; global, which never go out of scope; and buffer-local, which go out of scope when the current buffer is changed.

Buffer-local variables are useful for saving state information on a per buffer basis.

When searching for a variable, CRISP searches the symbol tables in the following order:

1. Local variables.
2. Buffer variables.
3. Global variables.

Therefore be careful when overloading symbol names. CRISP allows local variables, global variables and buffer variables to all be in scope at once, in which case the variable at the highest level in the above list is only accessible.

### Return value

Nothing.

### See also

create_dictionary()(pg. 63).

## mark([type])

### Description

*mark()* is similar to *raise_anchor()*/*drop_anchor()*. If a marked region is currently displayed, then *mark()* raises the anchor; if no anchor is displayed, then *mark()* drops an anchor.

type is an optional integer variable which is used to define the mark type if one is to be dropped. If it is omitted, it defaults to a normal marked region.

See *drop_anchor()* for a description of the marked region types.

### Return value

Nothing.

### See also

drop_anchor()(pg. 78).

## mark_line(flag, [toggle])

### Description

This macro is used to set a user-defined *marker* on the current line. User markers are special marks for macro writers. The primary reason for this macro is to allow the vi-mode g// command to be implemented by marking all lines that match a regular expression and then performing the associated command after the lines have been marked.

Flag is a value to set the marker to. If it is non-zero then the marker is set, otherwise it is cleared.

If the toggle parameter is specified then the *flag* parameter is ignored. If specified, it toggles the state of all lines, i.e. unmarked lines become marked and marked lines become unmarked. This option is used to implement the vi-mode v// command by reversing the matching sense of the search.

There is no guarantee that a line marked will maintain its mark if text is deleted and undone, or if a line is modified in any way (e.g. by splitting a line). Markers are only meant to be short term objects.

### Return value

If the mark was already set then it returns 1; if it was not set then it is zero. If the current line is beyond the end of the buffer then -1 is returned.

### See also

find_marker()(pg. 91).

## max(v1, [v2], [pos])

### Description

This function returns the maximum value of the specified expressions. If both values are integers, then an integer result will be returned. If either (or both) values are floating point then a floating point value will be returned.

If *v1* is a list, then this function will return the largest value in the list. Non-numeric items in the list will be ignored. In the case of finding the largest value in a list, the parameter *v2* is optional. If specified then this value is the *span* when searching the list, e.g. a value of 2 would look at each successive element.

If a list operation is being performed then the optional third parameter, *pos* may be specified. In this case it receives the index of the largest element found in the list.

### See also

abs()(pg. 45)., min()(pg. 144).

## mem_info([flags], [blk_size], [num_lines], [chunks], [num_bytes]);

### Description

This primitive is used to configure the virtual and physical memory used by CRISP, and/or return statistics on memory usage.

This primitive is not designed for general use, but is more often used for regression and internal testing purposes only.

### Return value

Amount of memory used by CRISP (actually the return value of the *sbrk(2)* system call -- on those systems that support it).

### See also

get_mem_info()(pg. 94).

## message(fmt, [arg1], [arg2], ..)

### Description

This macro is used to display a message on the status *prompt()* line at the bottom of the screen. fmt is a string and may contain printf-like % formatting characters. arg1, arg2, .. are integer or string expressions used to satisfy the % formatting options. Upto 6 arguments are allowed.

This macro can be used to display informational messages on the bottom of the screen; if error messages are to be displayed then the *error()* macro should be used instead.

The following format identifiers are supported:

| Format | Description |
|---|---|
| %c | Print a character. |
| %d | Print an integer expression. Options such as %03d can be used to perform zero insertion and supply a field width. |
| %e | Used for printing a floating point values. |

| | |
|---|---|
| %f | |
| %g | |
| %o | Print integer expression in octal. |
| %p | If its less than a second since the last call to message (or printf or error), then don't display the message. This is used by macros which want to print progress messages and want to avoid a costly call to other macros. |
| %s | Print a string. Field width and alignments are allowed. If a list is specified as the argument to this format specified, then the entire list will be printed in a canonical manner. |
| %u | Print integer expression in an unsigned format.. Options such as %03d can be used to perform zero insertion and supply a field width. |
| %x | Print integer expression in hex. |

The following prefixes are supported:

| **Prefix** | **Description** |
|---|---|
| %l... | User long format for printing, e.g. floats are printed with full double-precision. |
| %-.. | Value is printed left justified. |
| %* | Take the next argument as the field width specification. Numeric values are treated as below for %NNN. |
| %NNN | Where NNN is a numeric value. This specifies the width of the field to be printed, adding spaces as appropriate to pad the output field. If NNN is a negative value then field is left-justified. If NNN contains a leading zero, then leading zeros will be used to pad the field. |

### Example

```
message("The %s %s %s %s %s.",
      "cat", "sat", "on", "the", "mat");
message("%pMacro completed: %d%%", (val * 100) / total);
```

### Return value

Nothing.

### See also

error()(pg. 83)., printf()(pg. 153).

## min(v1, [v2])

### Description

This function returns the minimum value of the specified expressions. If both values are integers, then an integer result will be returned. If either (or both) values are floating point then a floating point value will be returned.

If *v1* is a list, then this function will return the smallest value in the list. Non-numeric items in the list will be ignored. In the case of finding the smallest value in a list, the parameter *v2* is optional. If specified then this value is the *span* when searching the list, e.g. a value of 2 would look at each successive element.

If a list operation is being performed then the optional third parameter, *pos* may be specified. In this case it receives the index of the smallest element found in the list.

### See also

abs()(pg. 45)., max()(pg. 143).

## mkdir(path, [mode], [tree])

### Description

This function creates a new directory. The path specified should be a string variable and is passed directly to the *mkdir(2)* operating system call.

If **tree** is specified and is non-zero, then the mkdir() primitive will create all the sub-directories leading to the the path as well. (Similar to **mkdir -p** Unix command).

**mode** is the protection codes used to create the directory. If omitted, the value 0755 (rwxr-xr-x) will be used.

### Return value

On success, zero is returned. Otherwise -1 is returned and the global variable ***errno*** is set to the reason why the call failed. (Refer to the system manual page for the error codes).

### See also

cd()(pg. 53)., is_directory()(pg. 132).

## mkfifo(path, perms)

### Description

This function can be used to create a named-pipe. This will work only on systems that support named-pipes.

*path* is the name of the FIFO to create, and *perms* is the system dependent permission bits.

### Return value

-1 on error, 0 on success. On an error the global variable *errno* will contain the reason for the error.

### See also

mkdir()(pg. 145).

## mktemp(string template)

### Description

This function is used to generate a temporary filename, using **template** as the template for the filename. The string **template** should contain a full path name with 6 trailing 'X's. On return the trailing 'X's are substituted for a unique number.

### Return value

New string with trailing 'X's substituted for a unique number.

### See also

tempnam()(pg. 206)., tmpnam()(pg. 207).

## mode_string(int perms)

### Description

This function returns a string corresponding to the file permission bits as defined by **perm.** The format of the string is similar to that generated by the Unix *ls* command. E.g. "-rwxrwxr-x".

### Return value

String version of the permission bits.

### See also

stat()(pg. 201).

## modf(x, ip)

### Return value

Returns integer part of x as a floating point number, and stores the fractional part of x in the floating point variable ip. Both the return value and the value stored in ip have the same sign as x.

## module("module-name")

### Description

The purpose of this function is to provide data hiding within a macro file. If you are writing a set of macros some of which are internal and some for external use, then you can use the CRUNCH '*static*' declaration specifier to restrict the visibility of a macro. However a static function is only 'visible' within the current macro file, and this presents a problem with dynamic callbacks (e.g. as a result of *assign_to_key* or *create_object*).

This primitive allows you to refer to static functions defined in another macro file explicitly, using the "::" naming modifier, and also allows static macro functions to be accessed in callbacks.

When specifying macros in callbacks, if a null module name is specified (e.g. "::function") then CRISP will automatically *bind* the potential macro call at the point of reference. This is a quick way of referring to the named function within the current file without having to explicitly define a module name and referring to it.

The module name should be a string containing a valid sequence of identifier symbols (e.g. [A-Za-z_][A-Za-z_0-9]* describes the set of valid identifiers).

Multiple source files can contain the same module() identification in which case the caller does not really care where a particular function is located. Each file will be checked in turn.

Once a module statement is executed, you can use the syntax "<module-name>::<function>" to refer to a function in callbacks. For example:

```
void
main()
{
    module("searching");
    assign_to_key("s", "searching::find_it");

    /* Similar but not necessarily */
    /* identical behaviour to the above */
    assign_to_key("s", "::find_it");
}

static void
find_it()
{
    message("Now what?");
}
```

It is strongly recommended not to call private static functions in other macro files unless absolutely needed. Doing so will break the object orientatedness of macro implementations and will make it difficult to implement future upgrades to the macros. The module mechanism is designed to avoid pollution of the name space and hence cause erroneous macro execution due to naming conflicts in different and possibly unrelated files.

### Return value

-1 if called outside the context of any macro file; 0 on success, 1 if module already existed.

### See also

inq_module()(pg. 117).

## move_abs([line], [col])

### Description

This macro moves the cursor to the specified line and column position. If either line or col are unspecified or zero, then the line or column position is left unchanged.

### Return value

Returns non-zero if cursor moved; 0 if cursor didn't move.

### See also

goto_line()(pg. 100)., move_rel()(pg. 147).

## move_edge([direction], [amount])

### Description

This macro is used to expand or contract a window. direction specifies the edge which is to be moved. The expansion is performed interactively. This macro is designed to support tiled windows.

If direction is specified, it should have one of the following values:

| | |
|---|---|
| 0 | Up |
| 1 | Right |
| 2 | Down |
| 3 | Left |

Amount is an optional integer expression. If not specified, the user will be prompted to move the edge with the arrow keys. If it is specified it specifies the number or characters or lines to move the window. The number is relative to the top left of the screen, so positive numbers move further away and negative numbers move nearer.

### Return value

Returns <= 0 if unsuccessful; > 0 otherwise.

### See also

create_edge()(pg. 63)., create_window()(pg. 65)., delete_window()(pg. 72).

## move_rel([line], [col])

### Description

This macro moves the current line or column plus or minus the number of lines specified, i.e. the move is relative to the current cursor position.

### Return value

Returns non-zero if cursor moved; 0 if cursor didn't move.

### See also

goto_line()(pg. 100)., inq_position()(pg. 119)., move_abs()(pg. 147).

## next_buffer([system], [prev])

### Description

*next_buffer()* is the mechanism for enumerating all buffers. It returns the buffer id of the next buffer in the list.

system is an optional integer which if present and non-zero allows the calling macro to walk through each system buffer as well (see create_buffer). If it is omitted, then system buffers are stepped over. This facility is used by the *buffer_list()* macro if the systems option is turned on.

If prev is specified and non-zero, then the previous buffer can be found.

### Example

The following example shows how to walk down each buffer, including system buffers:

```
int     curbuf, buf;

curbuf = inq_buffer();
buf = curbuf + 1;
```

```
while (curbuf != buf) {
...
    buf = next_buffer(1);
}
```

### Return value

Returns the buffer id of the next buffer in the buffer list.

## next_char([num])

### Description

This function is similar to *right()* except it moves the cursor over physical characters, e.g. tabs are treated as single characters, rather than virtual spaces.

If num is specified then that number of character positions are skipped over.

If the cursor is moved past the end of the line, then the cursor wraps around to the start of the next line.

When skipping characters for text files, the implicit newline at the end of each line is included in the count. When skipping over characters in binary files, the end of line is not treated as a character. Thus to go to an absolute offset in a binary file, it is merely necessary to go to the top of the buffer and perform a next_char() function the appropriate number of times.

### Return value

Returns non-zero if the cursor changed position; zero if it didn't.

### See also

prev_char()(pg. 152).

## next_window([win_id], [tiled_flag])

### Description

This macro is used to move to the 'next' window. The definition of next is related to the order the windows appear in the screen. Windows internally are sorted in top-left to bottom-right order. This primitive changes window and sets the current buffer up appropriately.

If win_id is specified then the next window from the specified window is returned.

If *tiled_flag* is specified and is non-zero then tiled (popup) windows will be included as candidates for the *next_window()* primitive.

### Return value

New window ID or -1 if window cannot be found.

## NLS(string)

### Description

This macro is designed to support nationalised text strings inside a macro. By default this function returns the passed in string. If the CR_LANG environment variable is set to point to a nationalised string database, then this primitive will look up the string in the database and return the translation for the string in the alternate language.

### Return value

**string** or translated version of the string.

## nothing()

### Description

This macro is a no-op. It exists simply as a place holder for the language and is valid anywhere any other macro is valid.

### Return value

Nothing.

## nth(expr, list_expr)

### Description

This macro allows the caller to extract individual atoms from a list. The first element of a list is element 0; the *nth()* macro allows lists to be treated like arrays. The last element of a list is *length_of_list()*-1.

nth is a more efficient way of accessing random elements in a list, than for example, using *car()* and *cdr()*, since the list expression doesn't need to be copied so much.

This primitive is not normally accessed from the CRUNCH language. When the user uses array subscripts to access a list, CRUNCH converts the subscripts into calls to this macro.

### Return value

Returns the n'th atom in list_expr, where n is the value of the integer expression expr.

## object_value([obj_id], [subobj_id], attribute);

### Description

This primitive is used to retrieve attributes of items in a dialog box or the current *screen*.

| | |
|---|---|
| obj_id | is the object-identifier of a user defined dialog box, or NULL if we are inquiring about the current screen. |
| subobj_id | is an integer representing the sub-object within a dialog box to be inquired about. Alternatively it can be a string in which case the sub-object within the dialog box that has the given DBOX_NAME attribute of the same value will be examined. |
| attribute | is a value indicating which value we are after. |

### Return value

Depends on the object and attribute. Refer to the **Programmers Guide** for more details on GUI programming.

### See also

dialog_box()(pg. 72)., change_object()(pg. 54)., parent_of_object()(pg. 150).

## open(path, mode, [prot])

### Description

This primitive can be used to explicitly open a file. The semantics of this function are exactly the same as the underlying operating system.

This primitive is provided so that intrinsic aspects of the operating system are available, such as file locking.

### Return value

On success a file descriptor is returned. On failure, -1 is returned.

### See also

close()(pg. 57)., read_from_file()(pg. 160)., write()(pg. 215).

## output_file([filename])

### Description

This macro is used to change the filename associated with the current buffer. By default the filename associated with the current buffer is the filename specified on an *edit_file()* or *create_buffer()* call. *output_file()* allows this to be changed to the value of the string expression, filename.

If filename is omitted, then the user is prompted for the new file name.

### Return value

On success, a value of 1 is returned. On failure, one of the following errors may occur:

| | |
|---|---|
| -1 | A blank or missing filename was specified. |
| -2 | The output filename when wildcard expanded, expanded into a blank filename. |
| -3 | The filename is the same as a buffer already loaded for editing. |
| -4 | The filename corresponds to a file on disk which the same device/inode number (e.g. a hard link to an already loaded buffer). |

## page_down()

### Description

Moves the cursor one window-full towards the end of the buffer.

### Return value

Nothing.

## page_up()

### Description

Moves the cursor one window-full towards the beginning of the buffer.

### Return value

Nothing.

## parent_of_object(obj_id)

### Description

This function can be used to return the object identifier of the parent of the specified object. At present this can only be used to map a DBOX_SCREEN value (e.g. as returned by inq_screen()) into the owning object.

This primitive is used when exiting from a peel-off window so that the macro can figure out which window to destroy.

### Return value

An integer valued object ID, or -1 if the passed object ID is invalid.

### See also

object_value()(pg. 149)., dialog_box()(pg. 72).

## paste()

### Description

This macro copies the contents of the scrap buffer and inserts it into the current buffer where the cursor is located. The cursor is moved to the end of the pasted region.

When using CRiSP on a windowing system (e.g. X11), this key should be bound to the <Ins> key. When a mouse paste operation is performed, CRiSP checks to see if this macro is bound to the <Ins> key and if not will not paste the selection. This is done to avoid allowing the user to paste data with the mouse into a popup window.

### Return value

Returns 1 on success; 0 on failure.

### See also

cut()(pg. 66)., copy()(pg. 61)., inq_scrap()(pg. 121)., set_scrap_info()(pg. 184).

## pause()

### Description

*pause()* is used to pause a keyboard macro definition. Usually all keyboard input typed during a *remember()* sequence is saved in the keyboard macro buffer. Pressing a key assigned to *pause()* causes the *remember()* sequence to suspend saving the characters.

### Return value

Nothing.

## pause_on_error([pause])

### Description

This macro is used to set or toggle the pause on error flag. This flag is tested after every *error()* message is displayed. If this flag is on then the error message is displayed with a '..' suffix added to the end of the error message and CRiSP waits for the user to type any key on the keyboard to continue. This allows the user to see error messages before they get overwritten by other messages.

By default this is off. If *pause_on_error()* is called with no argument, then the current state is toggled and CRiSP prints the message 'Pausing errors on.' or 'Pausing errors off.' If pause is specified then the current setting is set to the value of the integer expression pause.

### Example

The following example shows how *pause_on_error()* can be used around a call to a macro so that the user can see the errors being displayed by the macro without resorting to sight-reading.

```
pause_on_error();
fred();
pause_on_error();
```

### Return value

Non-zero if previous setting was paused messages on; 0 is previous setting was no pausing of error messages.

## playback([macro], [no_exec], [repeat])

### Description

*playback()* causes the saved keyboard macro to be replayed. If macro is not specified, then the last macro defined is played back.

If *no_exec* is specified then the specified keyboard macro will not be executed, but instead the playback() macro will remember the keystroke macro identified by *macro* so that the next time the user attempts to playback a macro, the specified macro will be executed, rather than the last keystroke macro which was defined.

*playback()* is not usually called from within a macro, but instead is usually bound to the <F8> key.

*repeat* specifies how many times to repeat the playback. If omitted then the playback is executed once. If *repeat* is zero then the playback will be repeated whilst the cursor is before the last line in the file. (Be careful when using this as you may end up in an infinite loop).

### Return value

Returns greater than or equal to zero if playback was successful. Less than zero otherwise.

## popup_object(obj_id, [x], [y])

### Description

The popup_object() primitive is provided to allow popup menus to be displayed when running windowing versions of CRiSP. The object id specified should correspond to a menu previously created with the create_object() primitive.

By default, **popup_object()** will popup a menu based on the last mouse position clicked in an edit window. **X** and **y** can be specified to force an absolute location for the popup.

The macro returns straightaway, i.e. it does not wait for the user to make a selection, and macro writers should be wary of this.  You should associate appropriate callbacks with the menu items.

### See also

create_object()(pg. 64)., delete_object()(pg. 71)., inq_objects()(pg. 118)., inq_window_system()(pg. 125).

## (post++ var)

### Description

This macro is used to access a variable and perform an auto increment on the variable AFTER its value has been used (c.f. (++)).

THIS PRIMITIVE IS NOT DIRECTLY ACCESSIBLE FROM THE CRUNCH LANGUAGE AND IS USED WHEN COMPILING THE POSTFIX INCREMENT OPERATOR (x++).

### Return value

Returns the value of the variable var.

## (post-- var)

### Description

This macro is used to access a variable and perform an auto decrement on the variable AFTER its value has been used (c.f. (--)).

THIS PRIMITIVE IS NOT DIRECTLY ACCESSIBLE FROM THE CRUNCH LANGUAGE AND IS USED WHEN COMPILING THE POSTFIX DECREMENT OPERATOR (x--).

### Return value

Returns the value of the variable var.

## pow(x, y)

### Return value

Returns x to the power of y (x^y). x, y and the return value are floating point values.

## prev_char([num])

### Description

Moves the cursor backwards by num characters. The movement is by physical characters, and treats tabs, etc. as single characters.

If the cursor is moved beyond the beginning of the current line then the cursor wraps around to the end of the previous line.

When skipping characters for text files, the implicit newline at the end of each line is included in the count. When skipping over characters in binary files, the end of line is not treated as a character. Thus to go to an absolute offset in a binary file, it is merely necessary to go to the top of the buffer and perform a next_char() function the appropriate number of times.

### Return value

Non-zero if successful. Zero if cursor didn't move.

## print()

### Description

This macro is used to send the currently marked region to the printer. The internal primitive is defined to do nothing. At the user interface level, the print macro is assigned to a macro which will send the currently hilighted region to the printer or the entire buffer if no region is hilighted.

### Return value

Nothing.

### See also

print_block()(pg. 153)., printer_dialog()(pg. 153).

## print_block([flags], filename, [lpp], [cols], [inc_filename], [from_page], [to_page], [font_size])

### Description

This primitive is used to print the entire current buffer or a section of the current buffer.

| | |
|---|---|
| **flags** | is a set of flags which control the printing. |
| **filename** | is the name of a file to write the output to. This can be an external process (e.g. the Unix printer spooler) by preceding the command with a pipe symbol, e.g. "\|lp". |
| **lpp** | is an optional number of lines per page. |
| **cols** | is the number of columns to print (1 and 2 column printing only are supported). |
| **inc_filename** | is the name of a file to be included at the top of the PostScript output file (normally `/usr/local/crisp/etc/crisp.ps`). |
| **from_page, to_page** | If *from_page* and *to_page* are specified then these are the page numbers to restrict printing to. |
| **font_size** | Size of font for printing. If omitted, a default of 10pt is used. |

### Return value

Number of pages printed, or < 0 if an error occurs.

### See also

print()(pg. 153)., printer_dialog()(pg. 153).

## printer_dialog([flags])

### Description

This primitive is used to access the native system printer support, and is primarily designed to gain access to the Microsoft Windows print common dialog box.

### See also

print()(pg. 153)., print_block()(pg. 153).

## printf(fmt, [arg1], [arg2] ..)

### Description

This macro can be used for debugging macros. It causes the formatted string to be sent directly to stdout, bypassing all of CRiSPs internal screen manipulations.

Under Unix, stdout is used for all display purposes, so the printf output stream would be intermingled with the display stream if you are using the character mode of CRiSP.

Under Microsoft Windows, using CRiSP.EXE a new console window is created to list the printf output.

If you need to track the progress of a macro, you can also use *debug()* since it gives you a much finer granularity of output.

Refer to the message()(pg. 143). primitive for details on the formatting options available.

### Return value

Nothing.

### See also

debug()(pg. 67)., format()(pg. 93)., sprintf()(pg. 200).

## process()

### Description

The *process()* macro is the mechanism for causing CRiSP to nest keyboard invocations. CRiSP maintains a process level counter which is normally set to 1. When in the process mode, CRiSP accepts keystrokes and executes the macros bound to those keystrokes.

A call to *process()* is usually made after creating buffers and windows for display on screen, and having set up a temporary keyboard map. Return from a call to *process()* only occurs as a result of a macro calling the *exit()* macro.

The current process level is terminated via a call to *exit()*. When the process level goes to zero, CRiSP exits (possibly after prompting the user to save buffers).

Process nesting may occur to any depth, within the bounds of CRiSPs maximum nesting level.

The term 'process' is a misnomer and in this context should not be confused with process buffers, which are a completely different concept.

### Return value

Nothing.

### See also

inq_process_level()(pg. 120).

## process_mouse(but1, but2, but3, x, y)

### Description

This macro exists to allow user macros to interface to CRiSP so that internal mouse processing can be performed in a generic manner for different types of mice. Some mice require internal support whereas others can be totally handled by user level macros, but at an execution time cost.

This macro gets called with the current state of the mouse buttons, zero means the button is up, and non-zero means the button is down.

The x and y parameters specify the co-ordinates of the mouse on the screen, with (0,0) being the top left of the screen.

When the macro is called, the states of the buttons are compared with the last recorded state and the REG_MOUSE registered macro is called with information about what type of object the mouse was clicked on.

Refer to the mouse.cr macro for details.

### Return value

Nothing.

### See also

translate_pos()(pg. 209).

## profile([expr])

### Description

*profile()* exists solely to help profile CRiSPs own execution so that it may be optimised. By default a mon.out file is created whenever a program exits which was compiled with the *cc -p* flag. When CRiSP exits, it usually avoids calling exit() so that the mon.out file does not get written. This allows CRiSP to be compiled with the -p flag and installed without causing every user to inadvertently create mon.out files all over the file system.

In order to get a mon.out file on exit from CRiSP, either run CRiSP with the -p flag, or call this macro.

If expr is omitted, the profile option is toggled. If expr is non-zero then a mon.out file will be created; if it is zero, then mon.out will not be created.

Note that this option only works if CRiSP was compiled with the profiling code included.

### Return value

Nothing.

## _prompt_begin(string)

### Description

This macro is a callback macro called whenever the user is about to be prompted on the command line. This function is called with the string which forms the prompt, and exists to allow the command line history and abbreviations mechanisms to be implemented.

### See also

_prompt_end()(pg. 155).

## _prompt_end()

### Description

This macro is a callback macro called after the user has finished entering text on the command line. It exists to allow the history facility of the command line to be implemented. The called macro normally saves the string the user typed in so it can be recalled.

### See also

_prompt_begin()(pg. 155).

## push_back(key, [front])

### Description

This macro is used to push a character or string back into the typeahead buffer. The character(s) pushed back sit at the front of the input buffer and is read before any user typeahead.

If key is an integer, then it represents an internal key code, usually derived from a *read_char()* or *key_to_int()* macro call. This can be used to push back arbitrary key sequences. If *key* is a string then the ASCII characters constituting the string are pushed back.

*front* is an optional integer which if present and is non-zero means that the characters pushed back are to be pushed at the front of any previous characters pushed back via the *push_back* function. If this is not specified or is zero then the characters are pushed back after all previously pushed back characters.

### Return value

Nothing.

## put_nth(expr list_var value)

### Description

*put_nth()* is used to modify or insert a new atom into a list. list_var is the list; expr is the atom number to be inserted (0 is the first atom). value is an integer, string or list expression which is to be inserted.

If the atom to be inserted already exists, then that atom is deleted, and replaced by the new value.

If *expr* is beyond the end of the list then the list will be padded with NULL values.

### Example

The following example creates a list of the ten integers, 1..10:

```
int      i;
list     l;

for (i = 1; i <= 10; i++)
    l += i;
```

### Return value

Nothing.

## put_parm(num, value)

### Description

This macro is used to modify a calling parameter. **num** specifies the argument number, and value is the value to place in that argument. value may be a string or an integer expression. (Lists are not currently supported).

This macro allows the calling macro to pass multiple values to its calling macro. The calling macro should call this macro with the name of an integer or string macro in the **num**'th position.

### Return value

Returns 0 if successful; greater than zero if argument num does not exist.

## put_selection(string text)

### Description

The put_selection() primitive is used when running the X11 versions of CRiSP. It is used to specify a string which is a candidate for pasting into other X11 client applications, conforming to the ICCCM.

The string specified as an argument is copied, and CRiSP asks the X server for ownership of the selection. If this succeeds, then the primitive returns zero. If it fails, then the primitive returns 1.

Because the string is copied, the calling macro is free to change the value after this call.

It is not possible to detect loss of the selection.

This primitive is used so that when the user selects the Copy/Cut functions from the popup menu, then the hilighted region will be available to other applications.

### Return value

-1 if ownership of the selection fails; 0 if the selection now belongs to CRiSP.

### See also

clipboard_owner()(pg. 56)., get_selection()(pg. 96).

## putenv(env_var)

### Description

This function is used to add or modify an environment variable. Environment variables are available via the getenv() function and also to any sub-processes created by CRiSP.

env_var should be a string of the form 'name=value'.

### Return value

None

### See also

getenv()(pg. 97).

## quote_list([list_expr])

### Description

The *quote_list()* macro is the mechanism for creating list literal constants. It is normally used for assigning initial values to lists, or for passing an anonymous macro to another macro.

For example, the following sets the list, l, to the two element list ("one" "two"):

```
list    l;
l = quote_list("one", "two");
```

If quote_list() wasn't used, and the expression appeared as:

```
l = "one", "two";
```

CRISP would assign the value "two" to the list, since in this context the comma is a binary operator.

Any CRISP expression can appear after the quote_list, including a sequence of commands.

### Return value

The quoted list, list_expr.

### See also

make_list()(pg. 141).

## quote_regexp(string, [unix_syntax])

### Description

This macro is used to quote a string so that it can be used in searches. The first parameter, *string* is the pattern to be quoted. The second optional argument may be used to indicate which type of regular expression (Unix or CRISP) is present. This affects certain quoting characters. If not present, then it will default to CRISP syntax.

For example, suppose the macro writer wishes to search for the pattern "<>" at the start of a line in the buffer. The correct way to do this is to write something like:

search_fwd("<\<\>")

It is possible for the macro writer to parse the string and insert the appropriate backslashes, but this would be a very expensive operation. It is not adequate to turn off regular expression parsing with *search_fwd()* since then the macro couldn't be sure to match the string at the start of the line.

In this case, *quote_regexp()* does the tedious work of translating the string so that it will match exactly.

### Return value

Returns a string with all regular expression characters quoted.

## raise_anchor()

### Description

*raise_anchor()* is used to raise the current marked region, i.e. remove it. The currently highlighted region is unhighlighted.

### Return value

Nothing.

## rand([num])

### Description

This macro is used to evaluate random numbers. The number returned is an integer in the range 0..2^31 or 0..num. The numbers may or may not be random enough, depending on the underlying implementation of rand(3). CRiSP tries to use the randomness of rand() to add a bit of variety to the low order bits of the returned number. However this may not be effective.

### Return value

Random number in range 0..2^31 or 0..num

### See also

srand()(pg. 200).

## re_search([flags], [pattern], [object], [start], [lensym], [scol], [ecol])

### Description

This primitive is designed to replace the search_fwd(pg. 173)., search_back(pg. 172)., search_string(pg. 173). and search_list(pg. 173). primitives. It provides the same functionality, but is compact and somewhat easier to read and remember. The motivation for this primitive is to allow the syntax mode of a search to be set temporarily for a search rather than for the entire session. This allows macros to be written which do not have to worry about the environment the user may have established.

The actual type of search depends on the type of the expression returned by **object**.

| | |
|---|---|
| **object** | If this is a string, then a string search is performed. If it is a list then a list search is done; otherwise a search on the current buffer is performed. |
| **pattern** | is the regular expression to look for. **start** is used for list searches and indicates the starting point in the list to start the search from. **lensym** is used for string searches to contain the length of the matched string found.<br><br>If ***pattern*** is an integer expression then ***object*** MUST be a list in which case the index into a list where the integer value is found is returned. |
| **flags** | is a set of bits defined in crisp.h which control the options of the search. The flags are defined below: |
| **scol & ecol** | These two fields specify the starting and ending column of a columnar search. They are only meaningful in conjunction with the **SF_COLUMN** flag. |

| **Flag** | **Description** |
|---|---|
| 0x01 SF_BACKWARDS | Perform a backwards search on the current buffer. If not specified a forward search is performed. Only meaningful for buffer searches or translates. |
| 0x02 SF_IGNORE_CASE | Ignore case when comparing strings. |
| 0x04 SF_BLOCK | Limit searches to hilighted block. (Buffer searches only). |
| 0x08 SF_UNIX | Select Unix syntax mode. If not specified, CRiSP mode is used. |
| 0x10 SF_LENGTH | If specified then successful match will return length of matched string + 1. If not, returned length will take into account any escape sequences in the pattern. Buffer search only. |
| 0x20 SF_NOT_REGEXP | Dont treat pattern as a regular expression. |

| | |
|---|---|
| 0x40 SF_GLOBAL | (re_translate only) Perform a global translate. |
| 0x80 SF_PROMPT | (re_translate only) Prompt user for each translation. |
| 0x0100 SF_MAXIMAL | When using CRiSP style syntax match the longest string possible. |
| 0x0200 SF_LINE | (re_translate only) If this is specified then each translation is only applied a maximum of one time per line. This is similar to vi's style of substitution where each line is matched only once even if the pattern occurs more times on each line. |
| 0x0400 SF_STICKY | (re_translate only) Dont restore cursor position after translate has been done. Normally used to execute a single translation without prompting the user, and ensuring cursor moves to the position where the translation occurred. |
| 0x0800 SF_LIST | This flag can be used when searching a buffer for a string. It causes the search to be repeated throughout the buffer and a list of integers containing the matching line numbers is returned. This is a quick way of performing a search for all lines matching a pattern and avoids the avoid of repeated regular expression compilations. |
| 0x1000 | Reserved |
| 0x2000 | Reserved |
| 0x4000 | Reserved |
| 0x8000 SF_COLUMN | Specifies that a column search is to be performed. The column boundaries are specified with the **scol** and **ecol** parameters.<br><br>Column searches currently only work on buffers. |

### See also

re_translate()(pg. 160)., search_fwd()(pg. 173)., search_back()(pg. 172)., search_string()(pg. 173)., search_list()(pg. 173)., re_syntax()(pg. 159).

### Return value

| | |
|---|---|
| buffer search: | 0 if pattern not found.  -1 on an error  length of matched text + 1 |
| list search: | -1 if pattern not found index of atom if found |
| string search: | 0 if pattern not found index of first character of match |
| integer search: | Index of atom where integer was found, or -1 on a failure |

## re_syntax([mode])

### Description

This macro allows the user to select the regular expression syntax mode. By default, the mode is set for CRiSP regular expression syntax. If mode is set to 1, then Unix like syntax is selected. The differences are as follows:

In Unix mode, the '*' and '.' characters have their normal meanings. The CRiSP characters '@' and '?' act as non regular expression characters. Also the { and } characters are replaced by their backslash-( and backslash-) equivalents.

### Return value

Returns the current regular expression syntax mode - 0 for CRiSP syntax, 1 for Unix-like syntax.

## re_translate([flags], [pattern], [replacement], [string], [scol], [ecol])

### Description

This function is a new interface to the translate()(pg. 208). primitive which allows the regular expression syntax mode to be specified temporarily for the translation. This function allows translations to be made in the current buffer or to a string. (Translating a string provides similar functionality to awk's sub() and gsub() functions).

CRiSP provides #define macros for *awk's* **sub()** and **gsub()** functions.

| | |
|---|---|
| **flags** | is a set of flags which control the translation. See description of re_search()(pg. 158). for further details. |
| **pattern** | is the string to search for. |
| **replacement** | is the string to replace the **pattern** with. |
| **scol** | Specifies starting column for searches marked using the SF_COLUMN flag. |
| **ecol** | Specifies ending column for searches marked using the SF_COLUMN flag. |

If **string** is specified then the substitutions are made to the string, and the resulting translated string is returned.

If either **pattern** or **replacement** are not specified then they will be prompted for.

### Return value

For buffer translations, the number of translations made is returned. For string translations, the translated string is returned.

### See also

re_search()(pg. 158)., search_fwd()(pg. 173)., search_back()(pg. 172)., search_string()(pg. 173)., search_list()(pg. 173)., re_syntax()(pg. 159).

## read([num], [flags])

### Description

**num** is a count of characters to read from the current line. If **num** is not specified, then all characters to the right of the cursor are returned, together with the newline character which terminates all lines.

If **num** is specified and greater than zero, then that number of characters are read from the buffer at the current cursor position. If **num** is longer than the remaining characters on the current line, then only the rest of the current line is returned. If **num** is less than zero then the characters before the cursor are read (without moving the cursor)

If ***flags*** is specified and is non-zero, then when reading text from the current line, all leading spaces and tabs are skipped over.

### Return value

Returns a string containing the next **num** characters from the input buffer, or the rest of the line after the cursor if **num** is not specified.

## read_from_file(fd, buf, [len])

### Description

This macro is used to read from a previously opened file. The macro name is inconsistent and should really be *read()* but read already exists and is used to read from the buffer.

This function acts like the standard C library read.

| | |
|---|---|
| **fd** | File descriptor previously returned by **open().** |

| | |
|---|---|
| **buf** | String variable. This receives the value of the data read. |
| **len** | If defined specifies how many bytes to read. If omitted a suitable buffer size is used. |

### Return value

Number of bytes read; zero on EOF; -1 if fd does not refer to a valid file descriptorr.

### See also

open()(pg. 149)., close()(pg. 57)., write()(pg. 215).

## readlink(filename)

### Description

This macro is used to read the contents of a symbolic link. On systems which support symbolic links, this macro returns a string containing the symbolic link, or an error value if (integer) if the file is not a symbolic link.

On systems which do not support symbolic links then this macro will return the integer zero.

When testing this macro, the result should be assigned to a polymorphic symbol so that the type of the variable can be tested first.

### Return value

Contents of symbolic link if filename refers to a symbolic link or -1 (and errno set to the appropriate error) if the file is not a symbolic link.

A value of zero is returned on systems which do not support symbolic links.

### See also

stat()(pg. 201).

## read_char([millisec], [raw], [nostore])

### Description

This macro can be used to see if a character is available from the typeahead buffer. If a character is available, it is returned (in internal key code form), and the character removed from the typeahead buffer.

| | |
|---|---|
| **millisec** | is specified then the read_char() macro will wait the specified number of milliseconds for a key-press before returning. If **millisec** is set to zero, then read_char() will wait until a key is pressed (same as not specifying millisec for CRiSP). If **millisec** is set to -1, then the keyboard will be polled. |
| **raw** | if specified and non-zero, then a raw character will be read from the keyboard. This means that hitting a function key will usually return the first ESCAPE character forming that key code, rather than being translated into the internal keycode. This facility is used by the quote() macro. This is really only needed if using the character mode version of CRiSP as this relies on escape sequences to distinguish function keys. |
| **nostore** | If specified, then the key returned will not be stored in any current keystroke macro definition. |

If a non-destructive peek into the typeahead buffer is required, use *inq_kbd_char()*.

Internal key codes should not be manipulated as is (unless the keys are being inserted into the current buffer), but rather should be converted to the canonical key-code format, via *int_to_key*.

### Return value

Internal key code of character typed or -1 if no character available.

### See also

inq_kbd_char()(pg. 111)., int_to_key()(pg. 128).

## read_file([edit_mode, ] [filename])

### Description

This macro is used to read a file into the current buffer at the current cursor position.

*read_file* takes up to arguments. If two arguments are specified then the first parameter is the edit-mode. (Refer to the *edit_file()* primitive for a list of edit settings; the edit-mode controls how files are read in to the current buffer, e.g. ascii mode vs. DOS mode, etc).

If no arguments are specified then you will be prompted for a filename and the default edit mode will be used.

### Return value

Zero if unsuccessful; non-zero otherwise. On an error, the global variable *errno* will be set to the reason for the failure.

### See also

edit_file()(pg. 81).

## redraw()

### Description

This macro is used to physically redraw every character on the screen, after first clearing the screen. It is sometimes required on serial lines, etc, where the screen has garbage on it, and CRiSP has lost control.

### Return value

Nothing.

## refresh([flag], [process_buttons], [recalc])

### Description

This macro is used to update the screen after making various changes to buffers and/or windows. It optimises the amount of output to the screen.

If **flag** is not specified and there is keyboard typeahead then the screen will not be updated; if **flag** is specified then the screen will be updated even if there is keyboard typeahead.

If **process_buttons** is specified and is non-zero, then any button presses on dialog boxes will be processed. You can use this to capture a click on a **CANCEL** button, in a CPU intensive macro.

If **recalc** is specified and is non-zero, then CRiSP will perform more work to redraw the window. This may be necessary when changing certain attributes, such as the spell-checking dictionaries.

### Return value

Nothing.

## register_file_class(filename-regexp)

### Description

This primitive is used to define a class of filenames which are to be treated the same. This is used by the setup mechanism so that users can treat all files matching the specified regular expression as the same. This is useful, when it is not possible to rely solely on the extension part of a filename.

The *filename-regexp* parameter is a regular expression, which is a superset of the usual C-shell style of expression. The characters ?, *, [] and  are supported to match any character, any string of characters, a range of characters, or any of a sequence of strings, respectively.

For example, to match filenames of the form: `file.c0123` you would specify a regular expression of "*.c[0-9][0-9][0-9]".

Be careful when registering file classes to avoid confusion caused by multiple expression matching the same filename. This can lead to non-obvious behaviour. (You normally do not have control over the order of registration of regexp's since the user interface hides this from you).

### Return value

Nothing.

## reregister_macro(num, macro, [type], [screen_id], [string])

### Description

This macro is identical to the **register_macro** primitive. The only difference is that if the existing registered macro definition exists, it is purged first. This avoids the problem of registering the same macro callback multiple times.

### Return value

Nothing.

### See also

register_macro()(pg. 163)., unregister_macro()(pg. 211)., register_timer()(pg. 166)., unregister_timer()(pg. 211).

## register_macro(num, macro, [type], [screen_id], [string])

### Description

Registered macros are hooks into the execution of CRiSP. CRiSP defines triggers which allow the user's macro's to be activated when certain conditions arise in CRiSP.

The conditions are described by the 'num' parameter, which is an integer expression. The conditions are listed below.

*macro* is the name of a macro to call when the trigger is invoked. *type* specifies the context of the macro. This can be one of the values: REGISTER_GLOBAL (default) -- the macro is executed under all circumstances, REGISTER_BUFFER -- macro is executed for current buffer only; REGISTER_SCREEN -- macro is executed for current screen only.

The *screen_id* parameter is used with the REGISTER_SCREEN type and indicates which GUI screen ID the macro can be invoked in.

The *string* parameter is optional and is context dependent. The meaning of this parameter will vary depending on the actual trigger being used. Refer to the description below for the meaning of this parameter.

Multiple macros may be associated with a particular registered macro. When the registered macro is called, the macros are called in FIFO order.

Registered macros may be removed by using the *unregister_macro()* macro call; a particular registered macro can be triggered via *call_registered_macro()*.

| Trigger | Description |
|---|---|
| 0 | A character has been inserted into the current buffer via *self_insert()*. |
| 1 | This is called every time the current buffer is changed via *edit_file()*. |
| 2 | This is triggered every time the <Alt-H> key is pressed, and is used for the context sensitive help feature. |
| 3 | This is triggered whenever an unassigned key is pressed. |
| 4 | This is triggered when the keyboard idle timer expires. It allows the autosave feature to be implemented. |
| 5 | This is called just before CRiSP exits. It allows macros to tidy up after themselves, e.g. delete temporary files, save editing state. |
| 6 | This is called every time a new file is read in to a buffer (via edit_file). |
| 7 | This is called whenever the user types the interrupt character (by default ^Y). This allows macros to set a flag which can cause them to abort. |

| | |
|---|---|
| 8 | This is called whenever the user types in an invalid key during input on the status/prompt line. This allows the abbreviation and help facilities to be implemented. |
| 9 | An internal error occurred. This used to trap segmentation violations inside CRiSP and allows users to write their own 'emergency' macro to save buffers. (One is already provided -- refer to the core.cr macro). |
| 10 | Called when mouse activity detected. This is not fully supported at present. |
| 11 | Called when process input available from a process buffer. |
| 12 | Called when keyboard buffer empty. This macro MUST return an integer value. If the value is zero, then keyboard input can be read. If it is non-zero, then the keyboard should not be read, but the internal code will check the push-back buffer for input. |
| 13 | Called when data is available from the PRIMARY selection after the get_selection() macro has been called. This will only happen on windowing systems supporting the ICCCM cut & paste mechanism. |
| 14 | This is the *drag'n'drop* trigger. Currently it is only supported in the XView version of CRiSP. When this macro is registered, then the macro will be called when the user drops a file icon into the CRiSP window. The macro is passed a string argument corresponding to the name of the file to be edited. |
| 15 | (REG_INPUT_FILENAME). This is the input-filename trigger. It allows a macro to be written which intercepts files being read into a buffer. The macro should return a string value which is the mapped name of the file (or the original filename unchanged). This trigger allows a macro to be written that allows files to be edited which are dynamically uncompressed when read in, for example. See the crisp.cr (infile_trigger macro) for an example of use.<br><br>The *string* parameter can be specified with this trigger to indicate that the callback macro should only be called if the file extension of the input file matches this string. |
| 16 | This is the output-filename trigger. This allows macros to be written which intercept all files which are written to and allows them to perform some file-name specific action. For example, an attempt to write to a symbolic-link could cause a macro to check if the file is a symbolic link and if so remove the symlink and write a new file. |
| 17 | This is the modified file trigger. It is called when Crisp detects that the corresponding file on disk for a buffer has been modified. This is used to detect situations where Crisp hasn't completely read in a file but maybe another editing session has destroyed the file. The calling macro should return an integer value to indicate whether the buffer should continue to be edited, or truncated at the current load point. A value of zero means to stop further editing. A value greater than zero means to continue editing of the buffer. |
| 18 | This trigger is reserved for GUI related windowing events. The macro will be called with a string. The current values are: "OPEN" will be passed when the window is mapped to the screen (i.e. uniconised); "CLOSE" when the window is iconised (unmapped). |
| 19 | REG_SCROLLBAR. Reserved for future use. |
| 20 | (REG_KEY_ACTIVITY). This is used to detect any keyboard input. It is similar to the REG_TYPED (0) trigger but may be used to detect a key being pressed even if no key is inserted into the buffer. This trigger is only called when CRiSP is waiting in the current process() level input loop. It will not necessarily be triggered by the various |

primitives which wait for a key (e.g. read_char()).

| | |
|---|---|
| 21 | (REG_SCREEN). The current screen has changed, e.g. the user has moved the mouse into another screen window. The macro is called with a single integer argument indicating the screen number which was selected. The calling macro would normally call *set_screen()* to allow the screen change to take place. If this function is not called then the user cannot change to the designated screen. |
| 22 | (REG_PROC_DIED). This triggers calls the specified macro within the context of the buffer which died, indicating that an attached process buffer has terminated. |
| 23 | (REG_UNTYPED). This is the exact converse of REG_TYPED, i.e. a trigger will be called after any keystroke which does NOT insert a character. |
| 24 | (REG_WRITE_FILE). Called whenever a file is successfully written. Used to allow the audit macro to make a log of all files which are modified. The registered macro is called with the filename of the buffer which has just successfully been written. |
| 25 | (REG_LOCKING). Reserved. |
| 26 | (REG_SIGNAL). A SIGUSR1 or SIGUSR2 signal has been received. |
| 27 | (REG_EXCEPTION). This callback is used to intercept macro execution errors. The callback macro is passed an error code and a name indicating the macro or variable causing the exception condition. |
| 28 | (REG_NEW_FILE_CLASS) |
| 29 | (REG_NEW_FILE_INSTANCE) |
| 30 | (REG_BOOKMARK) A bookmark has been dropped or deleted. |
| 31 | (REG_BUFFER_DELETED) A buffer has been deleted. This trigger is called twice when a buffer is deleted. The first time it is called with an argument which is the buffer-id of the buffer about to be deleted. The second time it is called after the buffer is deleted (buffer id arg is blank or zero). |
| 32 | (REG_UNDEFINED_MACRO) An undefined macro was called. Argument passed is name of macro. |
| 33 | (REG_INSERT_MODE) Insert mode has changed. |
| 34 | (REG_REMEMBER) A keystroke macro is being recorded or has stopped recording. |
| 35 | (REG_DELETE_SCREEN) Screen is about to be destroyed. |
| 36 | (REG_BUFFER_MOD) Buffer has been modified. This is in contrast to REG_FILE_MOD which indicates something external to CRiSP has modified a file being edited. REG_BUFFER_MOD will happen the first time you insert or delete characters into a clean buffer. This trigger is also called when the buffer is 'unmodified', e.g. if you keep on undoing back to the point where the buffer no longer needs saving. You should call the inq_modified() primitive to determine which of the two states you are in.<br><br>See also REG_BUFFER_MOD2 (below) which is similar. |
| 37 | (REG_DEFAULT_FILE). This callback is identical to REG_NEW but is called after REG_NEW and only if the input file does not have an extension. This allows a macro to set up things like colorization type to be written based on the contents of the file, if it cannot be determined from the file extension |

| | |
|---|---|
| 38 | (REG_CHANGE_DIRECTORY). The current directory has been changed, e.g. by executing the *cd*() primitive. |
| 39 | (REG_STARTUP). This trigger is called after all the startup code has completed, and gives macros a chance to do something before going *live*. Typically this may be used by pre-loaded macros. |
| 40 | (REG_DEFAULT_EXTENSION). Reserved for future use. |
| 41 | (REG_SELECTION_REQUEST). Another application has requested the clipboard selection from us. The callback function should return the string to be made available. |
| 42 | (REG_LINE_CHANGED) Reserved for future use. |
| 43. | (REG_CRISP_IPC) An internal CRISP IPC message was received. Used under Windows to all the single-instance/multi-instance mechanism to be implemented. |
| 44 | (REG_PRE_COMMAND) Called just before a keystroke is executed. |
| 45 | (REG_POST_COMMAND) Called just after a keystroke macro has been executed. |
| 46 | (REG_BUFFER_MOD2) This is similar to REG_BUFFER_MOD but is called to allow the macro to allow or deny modification to a buffer. The triggered macro should return a FALSE value to deny buffer modifications (as if the file were read-only) or a TRUE value to let modifications succeed. |
| 47 | (REG_BUFFER_RENAME). Buffer has been renamed, e.g. due to the **output_file** primitive. |

### Return value

Nothing.

### See also

reregister_macro()(pg. **Error! Bookmark not defined.**)., unregister_macro()(pg. 211)., register_timer()(pg. 166)., unregister_timer()(pg. 211).

## register_timer(msec, macro, [type], [screen_id])

### Description

This primitive can be used to register a macro to be called at some future time. msec is a time period (in milliseconds) specifying when the macro should be invoked. For example, a value of 1000 for msec means to call the macro in 1 seconds time.

This macro returns a timer identifier which can be used in a call to unregister_timer to cancel the timer.

Note that it is the macro programmers responsibility to ensure that timer programming is done with caution. The timer macro will be invoked at the point CRiSP asks for keyboard input, and for example, this could happen whilst the user has popped up a popup window (e.g. the buffer list). You need to carefully check the calling environment if you need to modify buffers or windows.

Although the time period is measured in milliseconds you should try not to rely on accurate timings and the callback procedure may be called up to one second later than the value requested.

Once the timer has gone off and the macro has been called the timer is removed. If you require repeated timers you will need to re-register the timer in your callback macro.

If *type* is specified and has the value REGISTER_SCREEN then the timer will automatically be cancelled when the owning screen is destroyed. For instance this is used to avoid calling the callback macro responsible for updating the time in the status bar. You can specify an alternate screen to the current one by specifying *screen_id*

### Return value

A timer identifier which can be passed to the unregister_timer function.

### See also

unregister_timer()(pg. 211).

## reload_buffer([buf_id])

### Description

This primitive can be used as a quick way to reload a buffer from disk. This can be useful for example when it has been detected that the file on disk has changed (e.g. by another application) and you wish to see these changes.

CRiSP will attempt to save as much information as possible, e.g. bookmarks, when the file is reloaded.

CRiSP will not allow buffers associated with processes or pipelines to be reloaded.

If buf_id is not specified then the current buffer will be reloaded.

### Return value

-1 if an error occurs, e.g. buffer does not exist or is not allowed to be reloaded. -2 is returned if the filename associated with the current buffer does not exist. 0 on success.

### See also

edit_file()(pg. 81)., read_file()(pg. 162)., connect()(pg. 59)., disconnect()(pg. 75).

## remember([overwrite], [rsvd], [mouse])

### Description

remember is used to record a keyboard macro (a sequence of keystrokes) that can later be played back via playback()(pg. 151).

**overwrite** is an optional string whose first character is examined to see whether an existing keyboard macro should be overwritten. If a keyboard macro already exists, and overwrite is not specified, then the user is prompted to overwrite the macro. If overwrite is specified and the first character of overwrite is a 'y' or 'Y' then the keyboard macro will be overwritten.

If remember is called whilst recording a keyboard macro, then the recording is terminated.

remember is not usually called as part of a macro but is usually bound to a keyboard key (**<F7>**).

remember will also create a buffer called **KBD-MACRO-xx** where **xx** is the keyboard macro number. This buffer contains the top level macros executed by the user typing in the keys. This facility allows the user to save and edit a keyboard macro. Note that not all the macros executed by the user are saved in the buffer - only the top level ones. What this means is that keyboard input to things like dialog boxes (e.g. the buffer list) will not be listed. The buffer so created is a normal buffer but will not be automatically saved when you exit. It is the user's responsibility to save the buffer if you want to keep the macro.

The ***mouse*** parameter should be used when the keystroke macro is being recorded/terminated because of a non-keyboard action. It should be specified and non-zero when using a mouse or menu bar callback. If this is not specified or is zero then the last keystroke of a macro may be discarded.

### Return value

On completion of the recording, a keystroke macro ID is returned.

### See also

inq_keystroke_macro()(pg. 113)., list_of_keystroke_macros()(pg. 138)., load_keystroke_macro()(pg. 139)., playback()(pg. 151).

## remove(string file)

### Description

This function is equivalent to the ANSI C function *remove()* for removing files.

### Return value

On success a return value >= 0 is returned; -1 is returned on an error, and the global variable *errno* is set to the reason for the error.

### See also

del()(pg. 69)., rmdir()(pg. 171).

## remove_object(obj_id, name_id)

### Description

This primitive can be used to delete a tree of objects from a dialog box.

## remove_property(dict, name)

### Description

This function allows you to delete a property from a dictionary/symbol table.

**dict** is the dictionary id as returned, for example, from create_dictionary, or the id of an object, such as a buffer or IPC connection.

### Return Value

Returns 0 on success. -1 if **dict** is not a valid dictionary or object. -2 if **name** is not in the specified dictionary.

### See also

create_dictionary()(pg. 63)., get_property()(pg. 95)., set_property()(pg. 185)., ipc_create()(pg. 128)., create_buffer()(pg. 62).

## rename(old_file, new_file)

### Description

This primitive is used to rename a file. old_file and new_file should be strings containing the names of the file.

### Return value

Returns zero on success and -1 on failure. On failure the global variable *errno* is set to the reason for the failure.

## replacement *macro*()

### Description

The keyword **replacement** is a declaration attribute which may be applied to macros. Normally when CRiSP loads a macro file, any new macros override any prior definition of a macro with the same name.

The **replacement** keyword marks the macro as a replacement for an existing macro. Specifically, CRiSP will keep any existing macro definition. An attempt to call the macro will call the most recently defined replacement macro. The replacement macro can call the prior macro definition by calling the macro with the same name. Visually this looks like a recursive macro call, but CRiSP calls the next oldest macro body instead.

This facility allows users to create macros which overload the functionality provided by any existing CRiSP system macro, without requiring modifications to the system supplied macros.

From the body of a replacement macro, you can call the prior macro definition by calling the macro with the same name. You can specify additional or modified arguments. If no arguments are specified then the parameters passed in to the current function are automatically forwarded to the replaced macro.

### Example

The following example can be used to overload an existing macro, but performs argument validation.

Original macro definition:

```
int divide(int a, int b)
{
      return a / b;
}
```

In another file, the replacement macro is defined:

```
replacement in divide(int a, int b)
{
      if (b == 0)
            return 0;
      return divide(a, b);
}
```

### Return value

Returns zero on success and -1 on failure. On failure the global variable *errno* is set to the reason for the failure.

## require(macro_file)

### Description

The *require()* primitive is used to ensure a macro file is loaded. It is similar to the *autoload()* primitive, except that the autoload mechanism relies on specifying specific macro names.

When using the require() primitive, you can specify a macro filename in a similar way to the autoload() filename. However you should note the following: if two macros both specify the same string for the macro_file, (e.g. "template/xyz"), then the same macro file will be referred to irrespective of the CRPATH or current directory. Specifically CRiSP maintains a table of the 'required' macro files which have been loaded. What it does not do is realise that a macro file may have been preloaded via the autoload() primitive before the require() primitive.

What this all means is that you should be consistent in your macros and avoid using an autoload() and require() primitive for the same macro file, since this can lead to the macro file being loaded more than once. (This may be acceptable for pure functional macro files but not for those carrying state information).

In general it is best to use the require() primitive in the main() function of your macro file.

### Return value

Returns 0 if macro file has already been preloaded; 1 if macro file loaded as a result of this require() function. -1 if macro file could not be autoloaded() for you.

### See also

autoload()(pg. 49)., load_macro()(pg. 140)., inq_macro()(pg. 115).

## restore_excursion()

### Description

*restore_excursion()* is equivalent to restore_position(2) and is #define'd in the crisp.h include file.

*restore_excursion()* is used to restore the buffer, window and position saved with the *save_excursion()* macro. This primitive is used frequently in macros which are going to create new temporary buffers and/or windows for displaying popup menus etc.

### Return value

Returns 0 if there is no saved position on the stack; 1 if position was successfully restored.

### See also

save_position()(pg. 172)., save_excursion()(pg. 171)., restore_position()(pg. 170).

## restore_position([type])

### Description

This macro restores a saved position from the position stack. The argument *type* indicates how to restore the position.

| | |
|---|---|
| 0 | The saved position is discarded. |
| 1 | The position is restored. If the saved position does not refer to the current buffer then the position for that buffer is restored but the buffer is not restored. |
| 2 | The last buffer saved is restored and displayed in the current window at the same point the save was made. |

If *type* is not specified then this is the same as specifying 1, i.e. position is restored but the current buffer and window are untouched unless the last saved position refers to the current buffer.

An argument of 2 is equivalent to the *restore_excursion()* primitive.

The saved position stack is independent of the current buffer.

### Return value

Returns 0 if there is no saved position on the stack; 1 if position was successfully restored.

### See also

save_position()(pg. 172)., save_excursion()(pg. 171)., restore_excursion()(pg. 169).

## return [expr];

### Description

This macro is used to return from a macro, and optionally return a value. If expr is specified, it may be an integer, string or list expression.

The current macro is terminated and control passes to the calling macro.

### Return value

Value of **expr**.

## returns([expr])

### Description

This macro is similar to *return*, except it doesn't cause the current macro to terminate. It simply sets CRiSPs internal accumulator with the value of expr. If any other statements follow the execution of *returns()*, then the accumulator will be overwritten.

Use of this macro is not recommended.

### Return value

Value of **expr**.

## right([n])

### Description

Moves the cursor 'n' positions to the right. This is equivalent to adding 'n' to the current column position. Note that when the cursor moves it doesn't move over characters atomically but one column at a time. This allows the cursor to move beyond the end of the line and into the middle of a tab stop, for example.

If you want to move over physical characters, use the *next_char()* and *prev_char()* macros instead.

n may be positive or negative. negative amounts move leftward.

### Return value

Nothing.

## rindex(search-string, pattern)

### Description

This function returns the index in search-string of the last occurrence of pattern. This function is useful for splitting a filename into its filename and directory components.

If the user needs to search for regular expressions, then the function search_string should be used instead.

(See also *index()*).

### Example

The following example splits a string into a directory and filename part.

```
string  dir, file, filename;
int     i;


i = rindex(filename, "/");
if (i != 0) {
    dir = substr(filename, i - 1);
    file = substr(filename, i + 1);
}
else {
    dir = ".";
    file = filename;
}
```

### Return value

Returns 0 if pattern cannot be found in search-string; otherwise returns the last occurrence of pattern in search-string.

## rmdir(string path)

### Description

This function can be used to remove the named directory.

### Return value

On success a return value >= 0 is returned; -1 is returned on an error, and the global variable ***errno*** is set to the reason for the error.

### See also

remove()(pg. 168).

## save_excursion()

### Description

This primitive is synonymous with *save_position()* and is #define'd as this in the include file crisp.h.

*save_excursion()* tends to be used in conjunction with *restore_excursion()*. It is used to save the current position, current buffer and current window in a stack so that the calling macro can restore the position after possibly creating popup windows etc.

It is synonymous with *save_position()* so that the code looks correct (i.e. for each *save_excursion()* call there should be a matching *restore_excursion()* call).

### Return value

Nothing

### See also

save_position()(pg. 172)., restore_position()(pg. 170)., restore_excursion()(pg. 169).

## save_position()

### Description

This macro is used to save the current buffer and cursor position. It makes use of an internal position stack; *restore_position()* can be used later to restore the cursor position.

Note that the position stack is an independent stack of any buffer. This means that *restore_position()* may be called to pop off the top entry of the stack even when the current buffer is not the same as buffer referenced by the top of the saved position stack.

### Return value

Nothing.

### See also

save_excursion()(pg. 171)., restore_position()(pg. 170)., restore_excursion()(pg. 169).

## screen_dump([file])

### Description

This macro is used to get a snapshot of the current screen image. It will write the current screen to the named file or to the file /tmp/crisp.screen if file isn't specified.

It does not make use of the screen's character attributes and makes an approximation for the screen drawing characters.

### Return value

-1 if cannot write file; 0 if successful.

## search_back([pattern], [re], [case], [block], [length])

### Description

This function searches from the current cursor position backwards towards the top of the buffer looking for pattern.

If pattern is omitted, it is prompted for.

If re is specified and it is zero, then pattern is treated as a pure string; if it is omitted or is non-zero, then pattern is treated as a regular expression. (See section on Regular Expressions for details of the syntax of regular expressions).

If case is specified and is zero, then the pattern is treated non-case sensitively, i.e. A matches a. Otherwise the search is performed with case being sensitive.

If block is specified and is non zero, then the search is confined to the currently marked region.

If length is specified and is non-zero, then it indicates that on a successful return the length of the matched string should be returned + 1. If it is not specified, then the return length will take into account any c parameters in the regular expression.

### Return value

Returns 0 if pattern not found. -1 if an error is detected. Otherwise it returns the length of the matched text + 1.

### See also

re_search()(pg. 158)., search_fwd()(pg. 173).

## search_case([case])

### Description

This macro is used to set the value of the search case flag. By default all searches are case sensitive, i.e. "A" does not match "a". By setting case to non-zero, case sensitivity will be ignored when performing matches.

If case is omitted, the current value is toggled.

### Return value

Returns the previous value of the case flag.

### See also

re_search()(pg. 158).

## search_fwd([pattern], [re], [case], [block], [length])

### Description

This function searches from the current cursor position towards the end of the buffer looking for pattern.

If pattern is omitted, it is prompted for.

If re is specified and it is zero, then pattern is treated as a pure string; if it is omitted or is non-zero, then pattern is treated as a regular expression. (See section on Regular Expressions for details of the syntax of regular expressions).

If case is specified and is zero, then the pattern is treated non-case sensitively, i.e. A matches a. Otherwise the search is performed with case being sensitive.

If block is specified and is non zero, then the search is confined to the currently marked region.

If length is specified and is non-zero, then it indicates that on a successful return the length of the matched string should be returned + 1. If it is not specified, then the return length will take into account any c parameters in the regular expression.

### Return value

Returns 0 if pattern not found. -1 if an error is detected. Otherwise it returns the length of the matched text + 1.

### See also

re_search()(pg. 158)., search_back()(pg. 172).

## search_list([start], pattern, list_expr, [regexp], [case])

### Description

This macro is used to search a linked list for a string pattern.

If start is not specified, the search is started from the beginning of the list; otherwise the search starts from the atom numbered start.

pattern is the string to search for, and may or may not be a regular expression, depending on the value of regexp. list_expr is the list to search.

If regexp is not specified or is zero, then pattern is not treated as a regular expression. If it is specified and non-zero then pattern is considered to be a regular expression.

If case is specified and is non-zero, then the case sensitivity is taken into account.

### Return value

The index of the atom in list_expr where the pattern was found; -1 otherwise.

### See also

re_search()(pg. 158).

## search_string(pattern, string, [length], [re], [case])

### Description

This macro is used to search string for the pattern, pattern. If re is specified and is zero, then pattern is treated as a literal string - not a regular expression (in which case this function is similar to *index()*). If case is specified and is zero, then the search is done with case insensitive.

If length is specified (name of an integer variable), then it will receive the length of the matched pattern, if the search is successful.

If a \c sequence occurs in the pattern, then the return value and setting of the length parameter occur as if the expression start at the point of the \c.

### Return value

Returns the starting character in string where the match was found, or zero if the match failed.

## self_insert([n])

### Description

This macro is the means for the user to type data into the current buffer. It causes the last character to be typed to be inserted into the buffer (or overtyped if overtype mode is on).

If *n* is specified, then the character whose ASCII value is n is inserted into the file instead of the last character typed.

If first parameter is a string then the entire string is inserted as if it were typed. This is useful to simulate input.

### Return value

Nothing.

### See also

insert()(pg. 125)., insert_buffer()(pg. 126).

## send_signal(signal)

### Description

This macro is used to send a signal to a process. The signal is sent to the process attached to the current buffer. If no process is attached to the current buffer then it is a no-op.

### Return value

Nothing.

### See also

kill()(pg. 136).

## set_application_icon([int flags], [string pixlib], [string icon])

### Description

This function is used to set the icon used for an application window. This icon is normally placed at the top left of the application window for Windows applications, and is used when the application is minimised.

By default, a CRiSP application icon is used. This primitive allows you to override that default for custom macros and applications.

**Pixlib** is the name of a pixmaps icon library (normally **pixmaps.xpl**). **icon** is the name of the icon in that library.

### Return value

-1 on error, 0 on success.

### See also

set_application_name()(pg. 175).

## set_application_name([name])

### Description

This function sets the application name. The application name is used internally by CRiSP in various circumstances, e.g. when displaying warning notices. This allows you to configure your application so that the name of CRiSP is hidden and instead you can use your own application name.

If **name** is not specified then the application name is not modified.

### Return value

Current application name.

### See also

set_application_icon()(pg. 174).

## set_backup([flags], [buf], [versions], [backup_dir], [prefix], [suffix])

### Description

This macro can be used to inquire or turn on/off the setting of the backups flag. The backups flag is tested every time a buffer is written away (via write_buffer). If the flag is on, then a backup file is made.

If flags is not specified then the current value of the backup enable flag is toggled. If flags is specified then the current backup flags will be set to this value. (Note that the meaning of these flags is slightly different if the *buf* parameter is specified). The current meanings of the flags is:

| Bit | Meaning |
|---|---|
| 0x01 | **BACKUP_ENABLED**. Backupswill be created if this is set. (Default) |
| 0x02 | **BACKUP_SLOW**. Dont rename() files when creating backups. Copy them instead. (Default) |
| 0x04 | **BACKUP_BAK**. Create .BAK files in preference to backup directory. |
| 0x08 | **BACKUP_REQ_VERSIONS**. Request versions. If this bit is set then set_backup() will return the integer numberof backup versions which will be retained. |
| 0x10 | **BACKUP_REQ_DIR**. Request backupdir. If this bit is set then set_backup() will return the string valued backup directory name. |
| 0x20 | **BACKUP_REQ_FLAGS**. Requests the current backupflag settings. |
| 0x40 | **BACKUP_REQ_SUFFIX**. Requests the current setting for the filename suffix. |
| 0x80 | **BACKUP_REQ_PREFIX.** Requests the current setting for the filename suffix. |

Bit 0x02 is used to preserve file permission semantics on systems where the owning directory of a file does not necessarily have the correct permissions to avoid a user from deleting/renaming a file.

If buf is specified then the backup flag will only affect the specified buffer. When setting the backup flag for a buffer you can only enable or disable backup file generation (bit 0x01). The slow-backup flag (0x02) is a global flag setting.

If *versions* is specified then this is the maximum number of versions of files to keep when creating backups in the backup directory. (Multiple backup files are maintained by creating subdirectories called 0/, 1/, ...).

If *backup_dir* is specified then this is the name of the directory where backup files are to be created. If this is blank or points to a non-existent directory then .BAK files will be created instead. (In addition, if bit 0x04 is set in the flags parameter then .BAK files will be created anyway).

*prefix* is used to set a prefix to be applied to backup files. *suffix* is used to set the suffix of backup files. By default *prefix* is blank and *suffix* is "**bak**". The suffix and prefix are only used when no backup directory is specified for backups.

If called from the keyboard, then one of the following messages are printed:

Backups will be created. Backups will not be created.

If set_backup is called with a single parameter of -1 (e.g. set_backup(-1)) then only the current flag settings will be returned.

### Return value

If the BACKUP_REQ_VERSIONS, BACKUP_REQ_DIR, or BACKUP_REQ_FLAGS flags are set then the appropriate information is returned.

If the specified buffer does not exist then -1 is returned.

Otherwise the previous value of the flags is returned.

## set_buffer(bufnum)

### Description

This macro is used for changing the current buffer; bufnum is a buffer identifier, as returned via a *create_buffer()* or *inq_buffer()* call.

*set_buffer()* does not cause any registered macros to be called (unlike edit_file).

### Return value

Returns the ID of the previous buffer or less than zero if bufnum is an invalid buffer.

### See also

inq_buffer()(pg. 105).

## set_buffer_cmap([cmap_id], [buf_id])

### Description

This macro is used to apply a character mapping to a buffer. A character map associated with a buffer has higher precedence than a map associated with a window.

The character map is used in converting column positions in a line into the actual character pointer in the internal code.

### See also

create_char_map()(pg. 62)., inq_char_map()(pg. 108)., set_window_cmap()(pg. 190).

### Return value

Returns value of cmap_id or -1 if the specified character map does not exist.

## set_buffer_flags([buf_id], [flags1], [flags2])

### Description

This primitive is used to modify the internal flags associated with a buffer. If *buf_id* is not specified then the current buffer is affected.

This allows a calling macro to make a set of changes to a buffer, but not have the buffer marked as being modified for the buffer_list macro or the exit code.

The *flags1* and *flags2* values are a set of bits used to define various attributes.

The flags1 parameter:

| Flag | Meaning |
|---|---|
| 0x01 | If set, buffer has been modified. |
| 0x02 | If set, buffer will be backed up when written away. |
| 0x04 | Buffer is marked as read-only. |

| | |
|---|---|
| 0x08 | Reserved. |
| 0x10 | Underlying file has execute permission. |
| 0x20 | Process attached. |
| 0x40 | Buffer is in binary mode. |
| 0x80 | ANSI mode -- color escape sequences will be processed. ANSI mode will only work if the current character map has the ESCAPE and BACKSPACE characters defined appropriately. |
| 0x100 | Buffer does not map tabs to spaces. |
| 0x200 | Buffer is a system buffer. |
| 0x400 | Make all characters in window visible. |
| 0x800 | Dont save undo info for buffer. |
| 0x1000 | File is new -- so it will be saved on next write_buffer(). |
| 0x2000 | Append <CR> to end of each line on save. |
| 0x4000 | (BF_TITLE) Label window title with full path name. |
| 0x8000 | (BF_HIDE) Text hiding enabled. |
| 0x10000 | (BF_STRIP_CTRLZ) Strip ^Z at end of file. |
| 0x20000 | (BF_APPEND_CTRLZ) Append ^Z when writing file. |
| 0x40000 | (BF_COLORIZE) Apply colorization algorithm. |
| 0x80000 | RESERVED |
| 0x100000 | (BF_MOD_LINES) Show modified lines. |
| 0x200000 | (BF_OLD_LINES) Show old line numbers when line numbers are enabled. |
| 0x400000 | (BF_LOG_FILE) File is a log file which may get extended. |
| 0x800000 | (BF_IS_PIPE) File is a piped command. |
| 0x01000000 | (BF_VOLATILE) File is a volatile buffer. |
| 0x02000000 | (BF_HEX_MODE) Buffer is being operated on in hex mode. |
| 0x04000000 | (BF_ICVT_TAB_TO_SPACES) Convert tabs to spaces on input. |
| 0x08000000 | (BF_ICVT_SPACES_TO_TABS) Convert spaces to tabs on input. |
| 0x10000000 | (BF_EOF_DISPLAY) Display [EOF] marker at end of buffer. |

The flags2 parameter:

| **Flag** | **Meaning** |
|---|---|
| 0x0001 | BF2_SCROLL_TITLE. If this bit is set then when the contents of the buffer inside a window scrolls to the left and right, then the title of the window will scroll with it. This allows you to display fixed column headings on a buffer. |
| 0x0002 | BF2_LEFT_TITLE. Put window title on left side of |

| | |
|---|---|
| | window. |
| 0x0004 | BF2_RIGHT_TITLE. Put window title on right side of window. |
| 0x0008 | BF2_OCVT_TAB_TO_SPACES. Convert TABs to spaces on output. |
| 0x0010 | BF2_OCVT_SPACES_TO_TABS. Convert spaces to TABs on output. |
| 0x0020 | BF2_OCVT_EMBEDDED. Convert inline spaces/tabs on output. |
| 0x0040 | BF2_OUTLINE. Enable outline display of buffer. |
| 0x0080 | BF2_ASCII_COLUMN. When in hex/ascii display mode, cursor is in the ASCII column. |
| 0x0100 | BF2_SPELL. Enable spell checking. |
| 0x0200 | BF2_OCVT_PRIVATE_TABS. When writing file, use buffer tab stops for conversions, not the default of 8 columns. |
| 0x0400 | BF2_LINE_NO. Display line numbers for buffer. |

### See also

inq_buffer_flags()(pg. 105)., create_char_map()(pg. 62)., set_buffer_cmap()(pg. 176).

### Return value

Nothing.

## set_buffer_title([buf_id], title)

### Description

This primitive is similar to *set_window_title()* but is used whenever a buffer is displayed in a window. (By contrast, using set_window_title() can cause the title to be lost as soon as you do an edit_file()).

*buf_id* is the buffer which you wish to affect. NULL specifies the current buffer.

### Return value

-1 is returned if the specified buffer does not exist. Otherwise zero is returned.

### See also

create_window()(pg. 65)., inq_buffer_title()(pg. 107)., set_window_title()(pg. 192).

## set_busy_state([flags], [obj_id])

### Description

This function can be used by macros in the GUI versions of CRiSP to change the mouse cursor to that of the *busy* cursor. This is used to give immediate feedback that some action that the user has invoked is going to take some time (it is best to provide feedback when you know the action will take more than say .5 of a second).

When called with no arguments no change in the cursor state is performed.

The flags argument is defined as follows:

If bit 1 is set (0x01) then the cursor will be set to the busy (stop-watch or egg-timer cursor); otherwise it is restored.

If the *obj_id* parameter is specified then only that dialog box will be affected by this function.

### Return value

-1 on an error (primitive not available). Zero for success.

## set_calling_name(name)

### Description

This macro is used to set the calling name of the current macro. The calling name can be retrieved via the *inq_called()* macro.

### Return value

Nothing.

### See also

inq_called()(pg. 107).

## set_char_timeout([timeout])

### Description

This macro is used to set the delay in reading a subsequent character from the keyboard which is part of an escape sequence. By default after reading an escape character, CRiSP will wait upto 1/2 second before deciding that an escape was pressed and not the start of a function key escape sequence.

You can change this default by calling this primitive and specifying a **timeout** (in milliseconds).

### Return value

Previous timeout value.

## set_color()

### set_color(list)

### set_color(NULL, list)

### Description

This macro is used to define the internal color names to be used for specific parts of the screen. The argument is a list of color names (strings) corresponding to each object type to be set. Refer to the table below for the meaning of each list element.

Optionally an argument list of enumerated strings can be specified, e.g. `set_color("black", "white")` will set the background color to black and the foreground color to white.

An integer argument within the list of color names means to skip the intervening colors (i.e. keep them the same) and start assigning colors from the specified index.

The second form of this function can be used to set the attributes of each color setting (i.e. bold, italic). The list parameter is a list of integer values corresponding to each attribute to be set. A value of NULL in the list means to ignore the setting for this color value.

| Code | Mnemonic | Description |
|---|---|---|
| 0 | COL_BACKGROUND | Background color of the screen. |
| 1 | COL_FOREGROUND | Foreground color for all windows. |
| 2 | COL_SELECTED_WINDOW | Color of selected window title. |
| 3 | COL_MESSAGES | Normal color of prompts and messages and Line:/Col: fields. |
| 4 | COL_ERRORS | Error message color. |
| 5 | COL_HILITE_BACKGROUND | Color of background for a hilighted area. |
| 6 | COL_HILITE_FOREGROUND | Color of foreground for a highlighted |

| | | |
|---|---|---|
| | | area. |
| 7 | COL_INSERT_CURSOR | Color associated with the insert mode cursor. |
| 8 | COL_OVERTYPE_CURSOR | Color associated with the overtype mode cursor |
| 9 | COL_BORDERS | Color associated with the window borders. |

### Return value

Returns -1 on failure to set colors; 0 if successful.

### See also

get_color()(pg. 93).

## set_crisp_flags([value])

### Description

This macro is used to set various internal CRiSP flags, mainly to do with optimisations and internal speed ups. The reason these options are not normally enabled is that they may be *unsafe*. For example the CRF_MMAP flag indicates on systems which support it to memory-map files into memory to avoid running out of swap space on large files. However on these systems, if the file is modified by an external program whilst CRiSP is editing the file, then CRiSP may core-dump if the file is truncated due to the semantics of mmap(). Occasionally, mmaping a file is the only way to allow a very large file to be edited without using all the systems resources. For small files there is little benefit in using these mechanisms.

If no argument is specified then the current value of the flags are returned. If a value is specified then the flags are set to the value.

When changing the value of the CRF_VM and CRF_MMAP flags, they will only take effect for new files which are edited -- they will not apply to files already in memory.

| Value | Meaning |
|---|---|
| 0x01 | (Default is on). This is the CRF_VM flag. It indicates that CRiSP can delay reading a file into memory until specific parts of the file are referenced. Any changes to the file on disk before CRiSP has finished reading in the file may cause CRiSP to display an informational message indicating that the file changed size etc. |
| 0x02 | CRF_MMAP. This flag indicates that on systems that support memory mapped files (using the mmap() call), that files may be mapped into memory to avoid using up swap space. This mainly applies to SVR4 compatible systems. If any files edited using this flag are truncated at any time whilst CRiSP has the file mapped may cause a SIGBUS error from CRiSP when it attempts to access pages stolen from its address space. |

### Return value

The previous value of the flags are returned.

## set_cursor_type([flags], [ins-normal], [ins-virtual], [ovr-normal], [ovr-virtual])

### Description

This primitive is used to set the cursor type. It only has effect under the GUI environments. There are four types of cursors which can independently be set. (Each CRiSP peel-off window can contain its own set of independent cursors). The cursors are: insert mode on a normal character, insert-mode but on a virtual space, overtype mode on a normal character, overtype mode on a virtual space.

There are four supported cursor styles, described in the table below.

Not all cursor types are available; for example, on a monochrome screen the cursor type is limited to a block cursor (reverse video).

| Value | Meaning |
|---|---|
| 0 | Reserved. |
| 1 | CURSOR_CARET: Cursor is a caret between characters. |
| 2 | CURSOR_BLOCK: Cursor is a block (uses the cursor color type). |
| 3 | CURSOR_BAR: Cursor is an underline type cursor. |
| 4 | CURSOR_BEAM: Cursor is an I-beam type cursor. |

### Return value

Returns the specified value as indicated by *flags*, or zero on success, or -1 on failure.

### See also

color()(pg. 57)., get_color()(pg. 93).

## set_editable_screen()

### Description

This primitive is used to reselect the last edit window. CRiSP supports windows and screens. Screens are used to implement the top level editing frame and there is usually one screen per peel off window. Some macros use screens as an alternative to list or table widget objects. In these cases, the screens are not generic editing windows. For example the compile output window is a screen (DBOX_SCREEN) widget. But it is not designed to be used for normal editing.

If you call the macro, for example, **edit_file()**, then this will change the file being viewed in the current window inside the current screen. This has the side-effect of possibly loading an editable file into a non-editable window.

**set_editable_screen** should be used to force input focus to be moved to the last selected edit window. You will see various examples of this macro in the CRiSP supplied macros  basically anywhere that a change of buffer can occur should force editing back to an edit screen and not default to the current screen.

CRiSP internally keeps track of the last editable window in order for this macro to work.

### Return value

Nothing.

### See also

inq_screen()(pg. 121)., set_screen()(pg. 185).

## set_font(int id, font_name)

### Description

This primitive is used to change the character font when running CRiSP in a windowing environment. It is a no-op in character mode environments.

There are two ways to call this function. The older way is to list the possible fonts to set in the function call. The newer version (Version 6.1.1 and above) allows you to specify a font identifier and font name. This newer mechanism allows future additions for new fonts.

Old calling convention:

```
set_font([font_name], [dlg_font_name], [fixed_font_name])
```

| | |
|---|---|
| **font_name** | if specified, sets the font for the main CRiSP editing area. |

| | |
|---|---|
| **dlg_font_name** | if specified, sets the font for the all the dialog boxes and other non-editing area specific parts of the user interface. When using this parameter to change the default dialog box font, it is platform specific as to whether existing fonts are changed, or whether only new dialog boxes are affected. |
| **fixed_font_name** | Specifies the font to use in dialog boxes where a fixed width font is needed. |

The font names used are platform specific. For example, under X11 fully qualified font names may be specified. The names which can be passed in these strings are exactly those normally returned by dialog_box("font").

Under Windows/32 some internal font names are understood (e.g. '`normal`' is the normal default font). Under Windows a specially coded font name may be specified corresponding the fields of the **`LOGFONT`** structure.

### Return value

Returns 0 on success and -1 on a failure.

### See also

dialog_box()(pg. 72)., font_ctl()(pg. 92)., inq_font()(pg. 110).

## set_idle_default([timer-value], [key-value])

### Description

This macro is used to change the value of the idle timer. The idle timer is a timer which can be triggered after so many seconds of keyboard inactivity or after every N key presses. The idle timer is used to implement the autosave feature.

The first argument, *timer-value* is a value in seconds measuring the amount of idle time before the REG_IDLE trigger fires off. If timer-value is zero then the REG_IDLE trigger will not be enabled.

The *key-value* parameter is a keystroke count and can be used to force the REG_IDLE macro to be fired after the specified number of keystrokes. This allows the autosave facility to fire off even if the user is continuously typing away.

### Return value

Previous value of idle timer-value.

### See also

register_macro()(pg. 163)., inq_idle_time()(pg. 111).

## set_kbd_name(name)

### Description

This primitive is used to define the current keyboard name. It is mainly designed for keeping track of the base set of function keys so that the setup macro can show the user which key bindings are in effect.

### See also

inq_kbd_name()(pg. 112).

## set_keyword_index([name])

### Description

This function is used to associate a keyword table for a buffer when it is displayed in a window. Keyword tables are loaded via the colorizer() primitive. The name parameter is passed to the "_load_keyword" function on a demand basis.

This allows buffers to have a keyword table associated with them and on the first reference to a keyword table, the macro mentioned above, will be called to cause the table to be loaded.

This allows different files to have a different keyword list which is highlighted (e.g. whilst editing source files in different languages).

### Return value

The keyword table name currently associated with this buffer.

### See also

colorizer()(pg. 58).

## set_line_flags([buf_id], [sline_no], [eline_no], [and_mask\, [or_mask])

### Description

This primitive is used to set the flags for one or more lines. You can either set flags or set/clear particular flag combinations.

| | |
|---|---|
| **buf_id** | Buffer to use. If not specified, use the current buffer. |
| **sline_no** | Starting line number; if omitted, current line number is used. |
| **eline_no** | Ending line number. If omitted just a single line is affected. |
| **and_mask, or_mask** | Each set of flags is logically ANDed with the **and_mask** and logically OR'ed with the **or_mask.** |

### Return value

Line number of line matching the search conditions. 0 if search fails.

### See also

find_line_flags()(pg. 90)., inq_line_flags()(pg. 113).

## set_model_buffer()

### Description

This function marks the current buffer as a model buffer. A model buffer is one whose various attributes can be inherited by a new buffer. CRiSP uses this primitive to allow files of a similar type to inherit the same attributes (e.g. colorizer, backups, etc. ).

Refer to the **packages.cr** macro file for an example of this macro.

### Return value

Nothing..

### See also

inherit_buffer_attributes()(pg. 103).

## set_msg_level([msg_level])

### Description

This macro is used to set the message level. The message level is used to control what sort of messages are to be displayed on the status line. It is needed by some macros which don't want CRiSP internal messages to be displayed.

msg_level is a number in the range 0-3 with the following effects:

| Value | Meaning |
|---|---|
| 0 | This is the default - all messages on. |
| 1 | Normal messages (those printed via (message) are not displayed). Error messages still display. |

| | |
|---|---|
| 2 | Error messages are disabled. |
| 3 | normal messages and error messages are disabled. |

Message level 3 can be used to ensure a macro is 'quiet'.

### Return value

Current message level.

### See also

inq_msg_level()(pg. 118).

## set_process_position([line], [col])

### Description

This macro sets the line and/or column associated with the input from a sub-process. line & col are optional, and if specified are integer values specifying the line and column where input should carry on from.

Processes maintain their own independent input in the buffer so that it is easier to write macros which manipulate sub-processes.

### Return value

Returns -1 if current buffer is not attached to a process; returns 0 otherwise.

### See also

inq_process_position()(pg. 120).

## set_record_size([buf], size)

### Description

This primitive is used to set a fixed line length for lines read from a file. Typically this would be used for a data file rather than a text file. If *size* is non-zero then the input file is split into *size* characters with no carriage-return or line-feed processing. If *size* is zero, then the default line mode behaviour is used.

This option will have no effect if a file has already been read in or partially read in. It should only be used for model buffers as an inherited buffer attribute, since it is only examined when a buffer is initially created.

### Return value

Nothing.

### See also

inq_record_size()(pg. 121)., set_model_buffer()(pg. 183).

## set_scrap_info([insert_newline], [mark_type], [buf_id])

### Description

This macro is used to set the newline flag at the end of the scrap buffer. This flag controls whether a newline should be inserted into the buffer after a paste operation. This field is not currently supported.

The mark_type parameter is used to set the type of scrap information. The mark_type should be a number indicating the type of region which was inserted into the scrap buffer. These numbers are the same as those for drop_anchor().

If buf_id is specified then that buffer is used as the new scrap buffer.

### Return value

Nothing.

## set_property(obj_id, property, value)

### Description

The *set_property()* primitive is a mechanism for associating one or more values with a dialog box. Properties are similar to variables (but they are always enclosed in quotes and can have arbitrary names). Properties can later on be retrieved by using the *get_property* function.

These two functions allow macros to associate values for a dialog box without recourse to creating global variables and tracking which variables belong to which dialog box. Thus they promote the ability to easily support multiple objects of a particular type.

When assigning a value to an existing property, the previous value will be lost. Note that properties do not have a '*type*'; they have a type as given by the value parameter. (Thus, properties automatically have a polymorphic property).

### Return value

-1 if the specified object does not exist; 0 on success.

### See also

create_dictionary()(pg. 63)., dialog_box()(pg. 72)., dict_exists()(pg. 73)., dict_list()(pg. 74)., get_property()(pg. 95)., remove_property()(pg. 168).

## set_ruler([buf_id], ruler_list)

### Description

Clears the current ruler settings and sets the buffer ruler markers to the list defined by **ruler_list. Ruler_list** is a list of integers for the column markers.

If **buf_id** is omitted then the current buffer is used.

### Return value

Returns 0.

### See also

inq_ruler()(pg. 121).

## set_screen([scr_id], [mask], [value])

### Description

The *set-screen()* primitive is used to change the current GUI window as if the user had clicked the mouse or started typing in to an alternate window.

Each GUI window is a separate screen. (Actually each CRiSP-like text editing window within a dialog box is a distinct screen).

The set_screen() primitive allows the calling macro to change screens and thus access the windows which are local to the particular screen.

You can either specify a single argument (scr_id) in order to specify the screen to switch to, or a mask/value pair which is used to access the DBOX_FLAGS of the parent of a dialog window. This latter option is used by the CRiSP macros to select peel-off windows and is used to avoid using arbitrary screens as sources of editing. (E.g. the color selector screen containing the sample text is not normally used as a general purpose editing window).

The mask value is and-ed with the DBOX_FLAGS field and if the value is equal to the specified value, then a screen switch will occur. This allows the calling macro to find the most recent of the potentially many peel-off windows.

NOTE: Screen #0 always refers to the initial startup screen.

### Return value

-1 if specified screen cannot be found.

### See also

create_object()(pg. 64)., dialog_box()(pg. 72)., inq_screen()(pg. 121).

## set_screen_size(lines, columns)

### Description

This primitive is available in the GUI versions of CRiSP only.

It is used to set the size of the viewing screen to the specified number of rows and columns. It will cause the window to be resized to make room for the number of characters.

The window cannot be made smaller than 3x3 to avoid problems with buggy macros.

### Return value

Zero on success; -1 if operation not supported or values out of range.

### See also

inq_screen_size()(pg. 122).

## set_term_characters([top_left], [top_right], [bot_left], [bot_right], [vertical], [horizontal], [top_join], [bot_join], [cross], [left_join], [right_join])

### Description

This macro is used to set the escape sequences needed to support the CRiSP window drawing characters. These escape sequences are non-standard in the sense that termcap does not support them (although some of these are supported by terminfo, but CRiSP does not currently support terminfo).

This macro should be called before the display is enabled (see *enable_display()*).

Each parameter is either NULL indicating that no value is being provided, or an integer or string expression. If an integer expression is specified, then CRiSP treats this as meaning that the escape sequence as such consists of a single ASCII character, whose value is given by the integer expression. If a string expression is given, then this string is printed when the specific character is needed on the display.

The following parameters have the following meanings:

| | |
|---|---|
| top_left | Printed at top left of window. |
| top_right | Printed at top right of window. |
| bot_left | Printed at bottom right of window. |
| bot_left | Printed at bottom left of window. |
| vertical | Printed at vertical line at side of window. |
| horizontal | Printed on horizontal line at top and bottom of window. |
| top_join | Character used at the top right and top left of windows where they abut each other. |
| bot_join | Character used at the bottom right and bottom left of windows where they abut cross Character used to print where four windows intersect at a common corner. |
| left_join | Character used at intersection of three windows on the left side of main window. |
| right_join | Character used at intersection of three windows on the right hand side of main window. |

### Example

For examples of how this macro is used, refer to the tty/*.cr macros, which have examples for a number of common environments.

### Return value

Nothing.

## set_term_features(index1, value1, index2, value2, ...)

### Description

This macro is used to specify certain attributes about the current display - features which are not adequately handled by termcap or terminfo. The arguments are specified in pairs. Each attribute is defined by an index value. These values are integer numbers; manifest constants can be found in the tty.h file in the macro source directory.

The value associated with an attribute may be either an integer or a string value, depending on the actual attribute. The attributes are described in the table below.

This function is used to refine the operating environment in which CRiSP will run and is used to describe terminal capabilities not adequately covered by the usual termcap/terminfo mechanism.

It is acceptable to omit many of these attributes for most terminals. These attributes are made definable to give a better performing CRiSP for those terminals which can describe these features. Some of these features are especially important for users using CRiSP over low speed modems.

The parameters describe the following features of the display:

| Arg | Description |
|---|---|
| **TC_ERASE_SPACES** | This is an escape sequence used to blank out the next N character positions. This is usually defined as "x1B[%dX" for those displays that understand the ANSI ESC [ X escape sequence. The escape sequence must have an embedded '%d' to specify where the numeric qualifier goes.<br><br>If this escape sequence is not defined, then CRiSP makes do with printing the correct amount of spaces and then moving the cursor back to the start of the cleared region. This is not as efficient as the ESC[X escape sequence when N is larger than about 8 or 9 characters. |
| **TC_EIGHT_BIT_CHAR** | This is an escape sequence used to allow access to characters with the top bit set. Some displays, especially those based on the IBM/PC require a special escape sequence to access the characters with the top bit set.<br><br>It is valid for a '%c' or '%d' to be embedded in the character definition.<br><br>Characters with the top bit set are normally displayed *naturally*, however if the user selects hex mode or some other mode then these characters may printed out as backslashed-hex sequences instead. |
| **TC_INSERT_CURSOR**<br>**TC_INSERT_VIRTUAL_CURSOR**<br>**TC_OVERWRITE_CURSOR**<br>**TC_OVERWRITE_VIRTUAL_CURSOR** | These escape sequences are used to change what the cursor looks like. These sequences represent the insert cursor, virtual-insert cursor, overtype cursor, and virtual-overtype cursor respectively.<br><br>If these escape sequences are not defined CRiSP will display the string **OV** when in overtype mode, and will not bother to distinguish between a virtual space and a real space. In windowing versions of CRiSP, CRiSP may display the cursor in a different color. |
| **TC_REPEAT_CHAR** | This defines an escape sequence for repeating the previous character printed. It is not currently used. |
| **TC_PRINT_ESCAPE** | This defines an escape sequence used for printing out the ESC character itself. This is available because some systems have a feature whereby there is a short |

| | |
|---|---|
| | cut to printing the ESC character. |
| **TC_COLOR** | If non-zero, then terminal supports color. The escape sequences for color are built into CRiSP. CRiSP assumes that if this is set, that the ANSI X3.64 escape sequences are valid. |
| **TC_FORWARD_CHAR** | Escape sequence to move 'n' columns forwards. (Needed because termcap cannot describe this entry). (Normally ESC[%dC for ANSI terminals). |
| **TC_BACKWARD_CHAR** | Escape sequence to move 'n' columns backwards. (Normally ESC[%dD for ANSI terminals). |
| **TC_UP_LINE** | Escape sequence to move 1 line upwards. (Normally ESC[A for ANSI terminals). |
| **TC_UP_LINES** | Escape sequence to move 'n' lines upwards. (Normally ESC[%dA for ANSI terminals). |
| **TC_DOWN_LINES** | Escape sequence to move 'n' lines down. (Normally ESC[%dB for ANSI terminals). |
| **TC_CLEAR_COLOR_IS_BLACK** | Sending an escape sequence to erase an area (e.g. to end of line) sets the area to black if this is specified. If not specified, or FALSE, then area gets set to the current background color. |
| **TC_AVOID_SCROLL_GLITCH** | Set to TRUE to avoid scrolling the window when attempting to optimise output. On some bitmapped displays its faster to rewrite the window. This attribute should not normally be set since CRiSP now supports scrolling regions. |
| **TC_ENTER_GRAPHICS** | Escape sequence to send when sending graphics (line drawing) characters. |
| **TC_EXIT_GRAPHICS** | Escape sequence to send when exiting graphics mode and going back to normal character set. |
| **TC_ANSI_STYLE** | Reserved. |
| **TC_GOTO_LINE** | Termcap style string used to go to column 1 of a line. E.g. ESC[%i%dH for ANSI type terminals. Needed because termcap nor terminfo can specify this attribute. |
| **TC_SCROLL_RECTANGLE** | Escape sequence which can be used to scroll a rectangular region. No *defacto* terminals or terminal emulators support this. This sequence is available in the fcterm color xterm terminal emulator only. |

### Example

Refer to the tty/*.cr files for examples of this macro.

### Return value

Nothing.

### See also

get_term_features()(pg. 97).

## set_term_keyboard(range1, key-list1, range2, key-list2, ..)

### Description

This macro is used to define the keyboard bindings. The keyboard bindings is an internal table to CRiSP describing which physical keys on the keyboard generate which internal key code.

When a key is pressed CRiSP scans the internal key code table to see if the character it has just read is part of a function key sequence. If so it waits until it receives an unambiguous string of characters either identifying the function key or a sequence typed by the user.

This primitive allows the user to define mappings between the escape sequences generated by the function keys and the internal key codes. It also allows multiple physical keys to be mapped to the same internal key code. For example, on a Sun keyboard the L1 and F1 keys can be mapped so that they are indistinguishable by the various macro primitives.

The internal function keys are given numbers out of the range of 8-bit ASCII characters.

On non-PC keyboards, e.g. SUN-3 keyboard, the function keys and LEFT/RIGHT keys can be used as a 'META' key prefix (much the way ALT is used on a PC keyboard).

The arguments to *set_term_keyboard()* consists of a sequence of ranges and key-lists. The ranges exist to make it easy to allocate the keys and read the macro afterwards. This avoids having to call this macro with upwards of 100 key definitions, and the resultant inability to decipher mistakes made on the 51st argument, for example.

The include file tty.h defines the ranges. A range is simply a number. The first function keys (<F1>..<F12> start at 128). By specifying a range of 128, this means that consecutive key definitions will start at this range, and be allocated consecutive slots from thereon. (See the tty-*.m files for an example of how this works).

(Ranges can also be specified by passing a string in the format of the *assign_to_key()* macro. The string will be converted to an internal key code automatically).

The key-list is simply a list of strings describing each consecutive key. If the keys are to be hard-wired, then the *quote_list()* function will be very useful. Otherwise the list can be constructed with the other list primitives.

Ambiguous key prefixes are allowed. This allows keys to be defined with the usual ESC-type prefixes. If the start of an ambiguous key sequence is typed, then CRiSP will wait up to .5 second for more keystrokes to resolve the ambiguity. After this time, CRiSP treats the original ambiguous characters as standalone. On some keyboards, e.g. Suns, this means that hitting the ESCape character will delay progress for 0.5 second, until CRiSP realises that no more keys are coming.

Because of this ambiguity problem, problems can arise when using CRiSP across a network, for example. When using applications like telnet, keeping your finger on the <Up> or <Down> keys, etc, can cause CRiSP to misread the keyboard, with the effect that instead of treating something like ESC[A as <Up>, it treats it as the single characters, ESC, [, A causing them to be inserted into the buffer. I do not know of any cure for this, except to increase the default .5 second timer. Increasing the .5 second timer inside CRiSP causes the side effect that hitting the ESC key at the buffer_list prompt, for example, causes an annoying delay.

### Return value

Nothing.

### Example

The following example is the first section of the *set_term_keyboard()* macro from the tty/at386.cr macro file.

```
set_term_keyboard(F1_F12,
    quote_list("\x1BOP", "\x1BOQ",
       "\x1BOR", "\x1BOS", "\x1BOT", "\x1BOU",
       "\x1BOV", "\x1BOW", "\x1BOX", "\x1BOY",
       "\x1BOZ", "\x1BO["));
```

## set_top_left([line], [col], [win_id], [csrline], [csrcol], [buf])

### Description

This macro is used to set the position of the buffer attached to a window. It sets the display position. All variables are optional integer expressions.

line and col are optional integer expressions which specify the line in the buffer which is to appear as the top line in the screen, and col specifies the column number which should appear in the top left hand position in the screen.

csrline and csrcol specify the buffer position to set in the associated buffer.

win_id specifies the window to use. buf indicates to use the window which has the specified buffer attached. win_id is used first, and if not specified, then the buf parameter is used. If neither are specified then the current window is assumed.

### Example

The following example puts the current line in the center of the window (this is the macro normally mapped to <Ctrl-C>).

```
set_center_of_window()
{
int     cur_line;

inq_position(cur_line);

set_top_left(cur_line - inq_window_size() / 2);
}
```

### Return value

Nothing.

### See also

inq_top_left()(pg. 122).

## set_window(win_id, [set_buffer])

### Description

Changes the current window to the window specified by win_id. win_id is a window identifier, as previously returned via *inq_window()*.

If **set_buffer** is specified and is non-zero, then the current buffer will be changed to the buffer of the specified window.

### Return value

Nothing.

### See also

inq_window()(pg. 123).

## set_window_cmap([cmap_id], [win_id])

### Description

This primitive is used to associate a character map with a window. A character map is a way of viewing the underlying data in a buffer. The character map is simple a map of each of the 256 possible byte values and the string to be printed. Character maps allow the user to write customised macros, for example to view a buffer in hexadecimal mode.

cmap_id is a character map ID previously created by the create_char_map() primitive, or NULL to use the internal default map.

win_id is the window to set the map for. If not specified, then the current window is used.

### See also

create_char_map()(pg. 62)., inq_char_map()(pg. 108)., set_buffer_cmap()(pg. 176).

### Return value

Nothing

## set_window_flags([win_id], value)

### Description

This is used to set certain options about how a window is displayed.  The flags are defined as follows:

| **Flag** | **Description** |
|---|---|
| 0x0001 | WF_LINE_NO. Display line numbers in the window. |
| 0x0002 | This flag is used to allow the cursor to *disappear* from view. I is primarily supported to allow the GUI scrollbar to scroll around the current buffer without forcing the cursor to move. When this flag is set it will cause the screen to reflect the res of a previous call to the set_top_left() primitive even althougł the cursor may not be visible as a result of doing this.<br><br>See the register_macro() description of event 20 for information on the REG_KEY_ACTIVITY trigger which may b used to detect keyboard input so that the cursor-ghosting car be turned off. |
| 0x0004 | WF_NO_SHADOW. Used to turn off the shadow around a popup window. |
| 0x0008 | WF_SELECTED. Pretend window is the selected window (i.є make the title appear as if the window were selected). |
| 0x0010 | WF_STICKY_TITLE. Dont change the window title when the buffer being viewed in it changes. |
| 0x0020 | WF_BUTTON. Window is acting as a button. |
| 0x0040 | WF_NO_BORDER. This flag indicates that the window is not to have any borders, irrespective of the *borders()* setting. |
| 0x0080 | WF_HILIGHT. Show currently hilighted region in this window even if this window is not the current window. |
| 0x0100 | WF_DELAY_UPDATE. Delay updating the window until the next refresh. Used to allow the current window contents to be *frozen* until a new keystroke, for example. This is used by the search highlighting macro to display the hilighted matched string, but will cause the hilight to automatically disappear on the next key press. |
| 0x0200 | WF_SYSTEM. A system window is similar to other windows but is ignored under certain circumstances, e.g. you cannot directly navigate into it using the change_window() primitive. is mainly provided for the implementation of the character mode menu bar facility. |
| 0x0400 | WF_MENU_BAR. Window is a menu bar. CRiSP will take int consideration the dimensions of this window when the main window is resized. |
| 0x0800 | WF_HEX_MODE. Reserved for future implementation. |
| 0x1000 | WF_BLOCK_NUM. Display current internal block number for each line in window. The semantics of this bit are subject to change and are used for internal debugging purposes only. Programmers should avoid using or setting this bit. The functionality underlying this bit is subject to removal at a futui date. |
| 0x2000 | WF_AUTO_WRAP. Lines too long to fit in the window will wr around to the next line. (Not currently supported). |
| 0x4000 | WF_EOF. Display an [EOF] marker at the physical end of a buffer so its clearer to see exactly where the buffer ends. Thi attribute overrides the default DC_EOF setting for the display_mode() control flags. |

### Return value

None.

### See also

display_mode()(pg. 76)., inq_buffer_flags()(pg. 105)., inq_menu_bar()(pg. 116)., set_buffer_flags()(pg. 176)., inq_window_flags()(pg. 124)., register_macro()(pg. 163).

## set_window_layer([win_id], [z-value])

### Description

This primitive is used to control the Z-axis (layering) of windows. Each window created is normally placed on top of any older created windows. Each such window is assign an x, y and z co-ordinate value. This primitive lets you retrieve the Z-axis value and modify it, e.g. so that the window appears on top of all others or below all others.

When called with no parameters, this primitive will return a new unique Z-axis value which you can then assign to a window to ensure it sits above any other created windows.

If *win_id* is specified only, but no z-value, then the current Z-axis value for that window will be returned.

If *win_id* and *z-value* is specified then the Z-axis value for the window will be set to the specified value.

### Return value

Returns next Z-axis value if no arguments are specified.

Returns -1 if the specified *win_id* does not exist.

Returns previous Z-axis value for the window.

## set_window_title([win_id], title)

### Description

This primitive can be used to change the title of a window. Normally the title of a window is based on the buffer being displayed in it. This allows the title to be overridden.

*win_id* is the window which you wish to affect or it may be NULL if you wish to change the title of the current window.

### Return value

-1 is returned if the specified window does not exist. Otherwise zero is returned.

### See also

create_window()(pg. 65)., set_buffer_title()(pg. 178).

## set_wm_name(window_name, icon_name)

### Description

This function is used to allow a macro to set the window name and/or icon name, when CRiSP is running under a windowing system. It is a no-op on other systems.

window_name and icon_name are string values. Either one may be omitted in which case the appropriate name will not be modified.

This function is designed to allow macros to be written which can set the window name to the current directory, for example.

### Return value

Nothing

## setegid(gid)

### Description

Sets the effective group ID of the current task to gid. May need system privileges.

This function is system dependent.

### Return value

0 on success, -1 on error, -2 if not supported.

### See also

getuid()(pg. 99)., geteuid()(pg. 98)., getgid()(pg. 98)., setuid()(pg. 194)., seteuid()(pg. 193)., setruid()(pg. 194)., setgid()(pg. 193)., setegid()(pg. 193)., setrgid()(pg. 193).

## seteuid(uid)

### Description

Sets the effective user ID of the current task to uid. May need system privileges.

This function is system dependent.

### Return value

0 on success, -1 on error, -2 if not supported.

### See also

getuid()(pg. 99)., geteuid()(pg. 98)., getgid()(pg. 98)., setuid()(pg. 194)., seteuid()(pg. 193)., setruid()(pg. 194)., setgid()(pg. 193)., setegid()(pg. 193)., setrgid()(pg. 193).

## setgid(gid)

### Description

Sets the group ID of the current task to gid. May need system privileges.

This function is system dependent.

### Return value

0 on success, -1 on error, -2 if not supported.

### See also

getuid()(pg. 99)., geteuid()(pg. 98)., getgid()(pg. 98)., setuid()(pg. 194)., seteuid()(pg. 193)., setruid()(pg. 194)., setgid()(pg. 193)., setegid()(pg. 193)., setrgid()(pg. 193).

## setrgid(gid)

### Description

Sets the real group ID of the current task to gid. May need system privileges.

This function is system dependent.

### Return value

0 on success, -1 on error, -2 if not supported.

### See also

getuid()(pg. 99)., geteuid()(pg. 98)., getgid()(pg. 98)., setuid()(pg. 194)., seteuid()(pg. 193)., setruid()(pg. 194)., setgid()(pg. 193)., setegid()(pg. 193)., setrgid()(pg. 193).

## setrlimit(int type, [rlim_cur], [rlim_max])

### Description

This function is machine dependent. On systems that support it can be used to set the current users resource limits. *rlim_cur* and *rlim_max* are integer variables specifying the new current and maximum limits. If either is not specified then the value will not be changed.

*type* is the resource limit to set.

Refer to your systems manual pages for more information.

### Return value

0 on success; -1 on error. -2 if call not supported.

### See also

getrlimit()(pg. 99).

## setruid(uid)

### Description

Sets the real user ID of the current task to uid. May need system privileges.

This function is system dependent.

### Return value

0 on success, -1 on error, -2 if not supported.

### See also

getuid()(pg. 99)., geteuid()(pg. 98)., getgid()(pg. 98)., setuid()(pg. 194)., seteuid()(pg. 193)., setruid()(pg. 194)., setgid()(pg. 193)., setegid()(pg. 193)., setrgid()(pg. 193).

## setuid(uid)

### Description

Sets the user ID of the current task to uid. May need system privileges.

This function is system dependent.

### Return value

0 on success, -1 on error, -2 if not supported.

### See also

getuid()(pg. 99)., geteuid()(pg. 98)., getgid()(pg. 98)., setuid()(pg. 194)., seteuid()(pg. 193)., setruid()(pg. 194)., setgid()(pg. 193)., setegid()(pg. 193)., setrgid()(pg. 193).

## shell([command], [use_shell], [completion], [nowait])

### Description

This command is used to execute a command in a shell. command is a string expression which evaluates to a command (with or without its arguments) which is exec()ed by CRiSP.

The use_shell parameter is an integer expression which controls whether the display should be left intact or not. If it is omitted or zero, then CRiSP puts the terminal back into normal *cooked* mode and executes the command string. When the sub-process exits, CRiSP repaints the screen. If **use_shell** is non-zero, then CRiSP does not repaint the screen. Setting **use_shell** to non-zero is useful for things like the make and directory manipulation macros which know the sub-process cannot destroy the screen.

The command string is passed to the shell so it can execute and interpret the command string (e.g. wildcard expansion, semicolons etc). The shell used is obtained from the SHELL environment variable. If this fails, it defaults to /bin/csh.

The return from this macro is the shell exit status.

If **completion** is specified, then it should be the name of a macro (in quotes - with optional arguments) which will be called when the sub-process terminates. When using this facility, you should set the use_shell parameter to 1 to avoid CRiSP from waiting for the sub-shell to complete.

The **completion** routine is called with the first parameter set to the return status from the underlying process. Any other positional parameters are shifted up one.

**nowait** can be used to tell CRiSP not to wait for the sub-process to terminate. Under Unix you can normally use an ampersand (&) after the command to force it to be a background process. Under Windows, you may not be able to do this. A non-zero value tells CRiSP not to wait.

WARNING: Under the GUI versions of CRiSP under Unix, this primitive is a NO-OP IF AND ONLY IF no arguments are specified at all. This is to prevent potential problems with hanging CRiSP if the user accidentally executes the *shell* macro from the Command: prompt whilst CRiSP is running as a background job.

`Example`

The following macro can be used to get a subshell to type commands into. The user terminates the sub-shell by typing the usual ^D or exit. (This is the same as what <Alt-Z> is usually mapped to).

```
shell()
```

The following example performs a *who(1)* command and saves the output in a temporary file so the calling macro can maybe display it prettily afterwards:

```
shell("who >& /tmp/who.tmp", 1);
```

The '1' tells CRiSP not to bother repainting the screen.

The following example causes a macro to be called when the sub-process terminates. The sub-process then prints a message saying that the process has terminated.

```
shell("make", 1, "completion");
...
completion(int status)
{
message("Job done, status = %d", status);
}
```

### Return value

Returns the shell exit status (under Unix, 0 means command exited successfully, non-zero means command failed for some reason).

## shell_execute(int flags, [string cmd], string filename, [string args], [string startdir], [int windowmode])

### Description

This primitive is provided for use on Win32 platforms which support the **ShellExecute** API function. This is used to *execute* a file - the actual operation being dependent on values stored in the registry for the registered file type and the **cmd** parameter.

| | |
|---|---|
| **flags** | Reserved for the future. |
| **cmd** | Operation to invoke. E.g. **open** may be used to execute a program or open a document.<br><br>**Edit** - opens file for editing.<br><br>**Open -** Opens file for editing (if it is a document), or executes the named application. If **filename** is a directory then the explorer is launched on it. |
| **filename** | Name of file to open/execute. |
| **args** | Optional command line arguments to the process. |
| **startdir** | Optional directory where to launch document or application. |
| **windowmode** | Value used to control initial window mode (minimized, maximized, normal, etc) for new application. |

### Return value

Returns -1 if the function fails or is not supported. Values less than zero indicate a failure under Win32.

Return value > 32 is a successful operation.

## simplify_filename(name, [type])

### Return value

This function simplifies a filename by removing redundant directory specifications e.g. **dir/../file** is the same as **file**. This helps to canonicalize a filename for ease in writing macros.

The **type** parameter is optional. If omitted it defaults to zero. The types of canonization which can be performed are:

| | |
|---|---|
| 0 | All tilde expansions and dollar environment variables are expanded. The resulting filename is simplified to remove redundant directory specifications. If the file is a relative filename then the current directory is prepended to the name. |
| 1 | All redundant directories are removed. No other conversion is performed. |
| 2 | All tilde and all dollar expansions are performed. If the filename is relative, then it is left that way. Type 2 is the same as type 0 except that the current directory is not added for a relative filename. |

### See also

file_glob()(pg. 86).

## sin(x)

### Return value

Returns the sine of x (in radians) as a floating point number.

### See also

cos()(pg. 61)., tan()(pg. 206)., asin()(pg. 46)., acos()(pg. 46)., atan()(pg. 46).

## sinh(x)

### Return value

Returns the hyperbolic-sine of x as a floating point number.

### See also

cosh()(pg. 61)., tanh()(pg. 206).

## sleep([sec])

### Description

This macro causes the calling macro to sleep for sec seconds (or 1 if omitted).

### Return value

Nothing.

## sort_buffer([bufnum], [flags], [col], [key_list])

### Description

This macro can be used to sort the lines in the current buffer or the buffer specified by bufnum.

Sorting can be done on a single key, or multi-column sorting can be specified. The **key_list** argument is used for multi-key sorting.

### Single key sorting

When single column sorting is to be performed, the **flags** parameter controls how the sorting is performed. The bits are described in the table below.

The optional ***col*** parameter specifies in which column the sort is to be performed. If not specified then the starting column is either column 1 or the starting column of the current region (unless SORT_WHOLE_BUFFER is set, in which case it will be column 1). Note: if you attempt to sort on a column other than the beginning of the line and the line contains tabs, especially spanning the region itself, then the results may not be what you expect. You may need to detab the region first (see the detab_text() primitive).

| Value | Meaning |
|---|---|
| 0x01 | SORT_DESCENDING. Sort the lines in a descending order. The default is ascending. |
| 0x02 | SORT_NUMERIC. Sort the lines treating the sort key as a numeric quantity. If this is not sorted then the lines are sorted purely on an alphabetic comparison. |
| 0x04 | SORT_WHOLE_BUFFER. If specified then sort the entire buffer and ignore any highlighted region. If not set and a highlighted region is specified then sort only the lines within the region; if no region is set then the whole buffer will be sorted. For a column region the sort key will be the left hand edge of the region. |
| 0x08 | SORT_BUFFERS. Sort the list of buffers into alphabetical order. This is used by the buffer list macro to show the buffers in alphabetical order. It will not affect the lines in the current buffer. |
| 0x10 | SORT_TEXT. Sort alphabetically, but keeping upper and lower case together. |
| 0x20 | SORT_VECTOR. If this option is specified then after the sort is complete a list of integers corresponding to the sort are returned. This sort vector can then be passed to the sort_list() primitive to order a list into an order corresponding to the sort of the buffer. This is useful when maintaining a buffer of data in corresponding order to a list, and you need to keep the two in step with each other. |
| 0x80 | SORT_DICTIONARY. Sort using dictionary order. All words starting with a letter come before any other character. Case is ignored. |

## Multi key sorting

Multi-key sorting allows you to sort a buffer based on multiple columns. Sorting is performed on the first specified column, and if the two values being compared are equal, then the second key will be used.

To perform multi-key sorting you need to specify the **key_list** value which is a list containing triplets of values defining the column, length and comparison flags:

```
list keys = make_list(
      1, 10, SORT_TEXT,
      20, 10, SORT_NUMERIC);
sort_buffers(NULL, NULL, NULL, keys);
```

This example creates a list specifying two columns. The first entry in the column definition shows that the columns start in position 1, and the second column starts in position 20.

The second part of the triplet specifies how long the field is; a value of -1 indicates the column spans to the end of the line.

The last part of the column specification is the flags which describe how values in this column should be compared.

## Example

The following example sorts all the lines in the current buffer into order:

sort_buffer()

### Return value

If SORT_VECTOR is specified then a list of integers corresponding to the sorting operation is returned.

### See also

sort_list()(pg. 198)., detab_text()(pg. 72).

## sort_list([flags], list, [sort_vector], [pick-list], [pick-offset], [offset])

### Description

This primitive can be used to sort a list of strings, integers or floating point atoms into order. The optional *flags* parameter can be used to control how the list is sorted.

You should only sort homogenous lists (atoms of the same type); attempting to sort a heterogeneous list will result in an arbitrary sorting order when comparing unlike atoms.

If you specify a pick-list to arrange the list, then you can also specify the optional parameter integer *pick-offset*. This is a number to be subtracted from each parameter of the list. It is may be needed because the result from a sort_buffer() operation has a one-base whereas lists have a zero base.

The *offset* parameter can be used when comparing strings to specify the offset into the string to start the sort.

| **Value** | **Meaning** |
|---|---|
| 0x01 | SORT_DESCENDING. Sort elements in a descending order. The default is ascending. |
| 0x02 | SORT_NUMERIC. Sort elements treating the sort key as a numeric quantity. If this is not sorted then the elements are sorted purely on an alphabetic comparison. |
| 0x04 | SORT_WHOLE_BUFFER. Reserved. |
| 0x08 | SORT_BUFFERS. Reserved. |
| 0x10 | SORT_TEXT. Sort alphabetically, but keeping upper and lower case together. |
| 0x20 | SORT_VECTOR. If this option is specified then after the sort is complete a list of integers corresponding to the sort are returned in the sort_vector list variable. This is useful when sorting independent lists which need to be sorted together. |
| 0x40 | SORT_PICK_LIST. If this option is specified and the pick_list variable is specified then the list will be sorted according to the atoms in the pick list. Typically pick-list is a list as returned by a prior SORT_VECTOR operation. |
| 0x80 | SORT_DICTIONARY. Sort using dictionary order. All words starting with a letter come before any other character. Case is ignored. |

### Return value

New list in sorted order. Any non-string values detected in the argument list will be ignored and removed from the returned list.

### See also

sort_buffer()(pg. 196).

## split(string, chars, [numeric], [no_quoting])

### Description

This macro can be used to parse a string and split it up into tokens, in a similar manner to the C function **strtok**, but it does the breaking up of the input string in one go and returns a list of strings. The **chars** parameter is a string containing delimiter character used to split the tokens.

If **numeric** is specified and is non-zero, then all tokens which start with a digit are converted to a number, rather than a string.

If the ***chars*** parameter contains a double quote character then it is assumed that any entries inside double quotes should have any spaces, etc preserved.  You can disable this behaviour by setting **no_quoting** to a non-zero value.

***string*** is the string to split; ***chars*** is either a string of delimiters or a single character (integer constant). If it is a string then all consecutive occurrences of the specified characters are used as a token delimiter. If ***chars*** is an integer then consecutive characters will not be merged. For example, the string "a:::b" may be parsed into 4 separate strings if *chars* is set to ':', but only two strings if *chars* is set to ":".

The **split()** primitive has been superseded by the **tokenize()** primitive which offers more functionality.

### Example

The following example causes 'lst' to be assigned a four element list containing strings where each string is one of the words of the parameter.

```
list lst;
lst = split("The quick brown fox", " ");
```

### See also

sscanf()(pg. 200)., index()(pg. 102)., substr()(pg. 203)., tokenize()(pg. 207).

### Return value

List of strings where each string is a 'token' from the string parameter.

## split_url(string, proto-var, host-var, page-var)

### Description

This macro can be used to parse a Universal Resource Locator (URL) and split it into its three main components: protocol, host, and page or name strings.

A URL is like a filename but is used on the Internet to name objects accessible by the various protocols available. A typical URL might look like:

**http://crisp.demon.co.uk/index.html**

The initial part leading up to the colon is the protocol name, and is entirely optional. The string following the ***//*** string up to the next forward slash delimits a hostname, which is also optional.

This function splits the input string and assigns value to the string variables: **proto-var, host-var,** and **page-var.**

### Example

The following example causes 'lst' to be assigned a four element list containing strings where each string is one of the words of the parameter.

```
list proto, host, page;
split_url("http://www.thing.com/index.html", proto, host, page);
```

This would assign the string "**http"** to the variable *proto*; **"www.thing.com"** to the variable *host*, and **"index.html"** to *proto*.

### See also

split()(pg. 199).

### Return value

-1 if the string does not contain a valid URL; zero on success.

## sprintf(buf, fmt, [arg1], [arg2], ..)

### Description

This macro acts like the C sprintf() function. It copies the string **'fmt'** to the string variable **'buf'** and applies the printf style formatting commands as specified in fmt, and using the argument list.

Refer to the message()(pg. 143). primitive for details on the formatting options available.

If the second argument is a list expression, then you can achieve a vsprintf() style of operation. In this case, the first element of the list is assumed to be a string format descriptor, and subsequent elements of the list are the arguments to print.

### Return value

Nothing.

### Example

```
list    lst;
string  str;

lst = make_list("%s %d %d %d", "hello", 1, 2, 3);
sprintf(str, lst);
message(str);
```

### See also

error()(pg. 83)., format()(pg. 93)., message()(pg. 143)., printf()(pg. 153).

## sqrt(x)

### Return value

Returns the square root of x, as a floating point value.

## srand([seed])

### Description

This macro is used to set a seed for the random number generator returned by rand(). If seed is not specified then the current time in seconds is used. Specifying seed allows the calling macro to have a repeatable set of random numbers.

### See also

rand()(pg. 158).

### Return value

Nothing.

## sscanf(string, format, sym1, sym2, ..)

### Return value

Number of variables actually assigned to.

### Description

This function is similar to the C language sscanf() function and is somewhat more generic than the split() function. It is used to parse a string and break it up into constituent elements assigning them to the specified variables.

THIS PRIMITIVE IS NOT YET IMPLEMENTED.

## _startup_complete(n)

### Description

This macro is a callback macro. It is called after all command line switches have been processed, after all command line files have been read in, and after the startup() macro has been called. The macro is passed an integer parameter, with the value 0 to indicate that no command line files were specified, or 1 to indicate that command line files were read in.

This macro is used by the restore macro to avoid restoring the state of buffers and files when files have been specified on the command line.

Refer to the **restore.cr** macro file for an example of this macro.

## stat(name, size, mtime, ctime, atime, mode, owner_id, owner_name, group_id, group_name, links)

### Description

The stat() function retrieves information about a file or file-system object. The file is specified with the *name* parameter and the other parameters are optional integer variables which receive the information: size of file, modification time of file, creation time of file, access time of the file, protection mode bits, user id, user name, group id and group name. The protection mode bits are operating system dependent.

The owner_id and owner_name return the user id and user name of the owner of the file or directory.

| | |
|---|---|
| **links** | Integer variable receiving the link count. On Unix systems this is a count of names associated with the underlying I-node. |

### Return value

On success, *stat()* returns a value of zero. On an error stat() returns a value of -1 and the global value errno will contain the operating system specific reason for the error.

### See also

ctime()(pg. 66)., find_file()(pg. 90)., file_pattern()(pg. 87)., is_directory()(pg. 132)., readlink()(pg. 161).

## static <type> id1, id2, ...;

### Description

The *static* keyword can be used to change the storage scope of a variable.

When applied to a variable definition outside of a function, the variable cannot be accessed by macros in other files. This provides a primitive form of object-hiding and can avoid name clashes with other macros.

When applied to a variable defined inside a function, this means the variable will maintain its value across invocations (and recursions).

### See also

extern(pg. 86)., global(pg. 99).

## strerror([errno])

### Description

This function returns a string corresponding to an error number. If error number is not specified then the internal errno value will be used representing the last error due to a system call.

### Return value

String containing text of corresponding error code.

## string var1, var2, ..;

### Description

This macro is used to define local variables which are to contain only string values. The variables defined are local variables, and are destroyed when the macro executing this declaration terminates.

String variables can contain variable length strings (up to 2^32 bytes long on machines with 32-bit int's, and 2^16 bytes on machines with 16-bit int's).

## string_count(string, char-list)

### Description

This macro is used to count the occurrences of characters in the string argument. char-list can be a string with one or more characters in it. The string argument is parsed looking at each character to see if any of the characters from char-list match it.

### Example

The following example counts how many spaces occur in the input string:

```
string  s;
int     num;

num = string_count(s, " ");
```

The following example counts how many upper case letters appear in the input string:

```
num = string_count(s, "ABCDEFGHIJKLMNOPQRSTUVWXYZ"));
```

### Return value

Returns the number of times the characters in char-list occur in the string parameter.

## string_match(str1, str2)

### Description

This function returns the length of the initial matching string for **str1** and **str2**.

### Example

The following example returns 5

```
num = string_match("abcdefg", "abcdefFGH");
```

### Return value

Returns the length of the overlap of the two strings.

## strlen(expr, [step])

### Description

This macro can be used to find out how long a string is or the length of the longest string in a list. If expr is a list expression, then step indicates whether certain entries should be skipped over, e.g. if the list contains paired tuples where the odd elements of the list are sub-lists, etc. If omitted, step defaults to 1.

When finding the lengths of lists, it always starts with the first element of the list. strlen silently ignores none string elements in the list.

### Return value

Returns the number of characters in the string expression, expr or the length of the longest string in the specified list.

## strtol(string, [index], [base])

### Description

The strtol function is used to convert an ASCII representation of a number to an integer. Leading '0x' is treated as a hexadecimal number; a leading '0' is treated as an octal number. Otherwise the number is treated as decimal.

If *base* is specified then the string is treated as a number in the specified base. If *index* is specified then the index into the string after the number is returned in the integer variable specified.

### Return value

Integer corresponding to number stored in string.

### See also

atoi()(pg. 48).

## sub(pattern, replacement, string)

### Description

The *sub()* macro is used to perform a single substitution on a string. *pattern* is the string to search for. *replacement* is the replacement text. *string* is the string to apply the translation to.

This function is designed to emulate the function of the same *awk(1)* name.

This primitive is actually a function defined in **crisp.cr**. It is implemented as a wrapper around the *re_translate* primitive.

### Example

```
sub("e", "", "fred had a banana");
```

This deletes the first letter 'e' from the string.

```
sub("[0-9]+", "(1)", "1+2+3");
```

This encloses in parenthesis the first number in the input string.

### Return value

The translated string is returned.

### See also

re_translate()(pg. 160)., re_search()(pg. 158)., gsub()(pg. 101).

## substr(string, start, [length])

### Return value

Returns the sub-string of string which starts at start, and goes on for length characters, or the end of the string if length is omitted.

### See also

index()(pg. 102)., rindex()(pg. 171).

## suspend()

### Description

This macro is equivalent to typing Ctrl-Z if you have job control enabled. It may be called from a macro to suspend crisp and allow the user to user an alternative key to Ctrl-Z if that is being used for something else.

This macro will only work on systems which support job control.

### Return value

Nothing.

## swap_anchor([dir])

### Description

This macro swaps the current cursor position with the start of the currently marked region, i.e. if the cursor is at the end of the marked region, the cursor is moved to the start of the marked block, keeping the marked block intact.

If **dir** is specified then the following will happen:

| | |
|---|---|
| 0 | Same as not specifying **dir**. Cursor position and anchor start are swapped. |
| 1 | Cursor is swapped so that cursor is at the top end of the region. |
| 2 | Cursor is swapped so that cursor is at the bottom end of the region. |

### Return value

0 if no movement happened; 1 if cursor position changed.

### See also

drop_anchor()(pg. 78)., end_anchor()(pg. 82)., raise_anchor()(pg. 157).

## switch (expr) { case-1: list-1 case-2: list-2 ..}

### Description

This macro implements the CRiSP switch statement. expr is an integer or string expression which is evaluated and then compared with each of case-1, case-2, etc. case-1, etc should be expressions of the same type as expr. If a match is found, then the statements in list-n are executed.

If CRiSP encounters a case-n value of NULL, then this is taken to be the default and the list is unconditionally executed.

If statement list associated with an expression is NULL, then the next non-NULL statement list associated with a case expression is executed. This allows multiple values to be associated with the same set of statements. (See example below).

### Example

The following example illustrates an example of integer switches. If the value of **i,** is 1, 2, or 3 then word is assigned the strings "one", "two" or "three" respectively. If **i** is not one of these three values, then word is assigned the ascii representation of **i**.

```
string  word, buf;
int     i;

switch (i) {
  case 1:       word = "one";
     break;
  case 2:       word = "two";
     break;
  case 3:       word = "three";
     break;
  default:
     sprintf(buf, "%d", i);
     break;
}
```

The following shows the reverse:

```
string  word;
int     i;


switch (word) {
  case "one":   i = 1;
    break;
  case "two":   i = 2;
    break;
  case "three":         i = 3;
    break;
  default:
    i = atoi(word);
    break;
}
```

The following shows an example of using multiple case values:

```
string word;

switch (word) {
  case "y":
```

```
                case "Y":
                  word = "YES";
                  break;
                case "n":
                case "N":
                  word = "NO";
                  break;
              }
```

### See also

break(pg. 52)., case(pg. 53).

## tabs([tab1], [tab2], .., [tabn])

### Description

This macro is used to set the tab stops for the current buffer. tab1, tab2, set the first tab stops. Tab stops are set every (tabn - tabn-1) stops after the last tab stop. The first column in the line is column 1.

[Instead of specifying an explicit set of tab settings you can specify a list of integers specifying the tab stops. This allows the return value of the tabs() function to be re-used to restore tab settings].

Upto 5 unique tab stops can be specified, thereafter all tab stops default to the difference of the last two tab stops.

In soft tab mode (see (use_tab_char)), whenever a tab is inserted into the text, enough spaces are inserted automatically to take the cursor to the next tab stop. When in hard tab mode, physical tab characters are inserted.

When in hard tab mode, the tab stops only affect the way the text buffer is displayed; if tab stops other than the default 8 are used, then the user may have problems printing files or using other utilities to examine the file due to column misalignment.

### Example

The following sets tab stops to every 4 columns:

```
tabs(1, 5);
```

The following sets tab stops to every 8 columns:

```
tabs(1, 9);
```

### Return value

Returns a list of the current tab stops.

## tag_close_db([int tid])

### Description

This function should be used to close a previously open tag database.

### Return value

Returns zero if the specified database has been successfully closed or -1 if the **tid** does not refer to an open database.

### See also

tag_open_db

## tag_list_tags([int buf_id], [lang], [list type_list], [list &ret_list])

### Return value

This function is used to parse a buffer and return a list of objects defined in the buffer.

| | |
|---|---|
| **buf_id** | specifies the buffer to parse. If this is NULL then the current buffer is parsed. |

| | |
|---|---|
| **lang** | specifies the language to use when parsing the file. This can be an integer language identifier or a string.<br><br>If it is a string, then this is a file extension (without the leading dot) to indicate the file type.<br><br>If this is an integer, then it is a language identifier code to indicate the type of file.<br><br>If this is not specified then the file extension of the buffer is used to determine the file type. If the file extension is not supported then the current colorizer scheme is used to determine the file type. |
| **type_list** | List of types to return. If not specified all object types are returned. |
| **ret_list** | List filled in on return of objects matching the tagging criteria. |

The **ret_list** list is a list of triplets. Each triple contains (object-name, line-number, type).

### See also

## tan(x)

### Return value

Returns the tangent of x (in radians) as a floating point number.

### See also

atan()(pg. 46)., atan2()(pg. 46)., tanh()(pg. 206)., atanh()(pg. 47).

## tanh(x)

### Return value

Returns the hyperbolic tangent of x as a floating point number.

## tempnam(string dir, [string pfx])

### Description

This function can be used to generate a unique temporary filename. The **dir** parameter specifies the directory where the temporary file is to be created.

If **pfx** is specified then the temporary filename returned will contain the specified prefix. **pfx** is optional and can be omitted.

### Return value

New unique temporary filename,

### See also

mktemp()(pg. 145)., tmpnam()(pg. 207).

## time([hours], [mins], [sec], [usec])

### Description

hours, mins, secs and usecs are optional integer variables which receive the current time.

hours is a number in the range 0-23; mins is a number 0-59; secs is a number 0-59. usec is the number of microseconds into the current second. (On systems which do not support microsecond accuracy, usec may only be accurate to milliseconds or may be zero).

### Example

The current time can be printed as such:

```
int     hours, mins, secs;

time(hours, mins, secs);
message("Time now: %02d:%02d:%02d", hours, mins, secs);
```

### Return value

Returns time in seconds measured from the *base* date: 00:00:00 GMT Jan 1 1970 for Unix based systems.

### See also

ctime()(pg. 66)., date()(pg. 67)., time_passed()(pg. 207).

## time_passed([msec])

### Description

This primitive can be used by macros which want to keep a check on the elapsed time since a macro started, e.g. in order to display periodic status messages.

When called with no argument, *time_elapsed()* returns 1 if more than the current number of milliseconds have passed since the last return of a 1 value from this function.

If an argument is specified then the primitive returns 1 if more than the specified number of milliseconds has passed since the last time the function returned 1.

### Example

The following function counts as fast as possible, updating the display periodically to show a tick update every second.

```
int i, i0;

for (i = 0; i < 10000000; i++) {
  if (time_passed()) {
    message("Count=%d Diff=%d", i, i - i0);
    i0 = i;
   }
}
```

## tmpnam()

### Description

This function returns a unique temporary filename.

### Return value

New unique temporary filename,

### See also

mktemp()(pg. 145)., tempnam()(pg. 206).

## tokenize(str, delims, [flags])

### Description

Tokenizes (splits into words) the input string. **str** is the input string. **delims** is a string consisting of one or more characters which indicate the delimiter characters.

**flags** is an integer containing a set of flags which indicate how the input string is to be tokenized.

TOK_COLLAPSE_MULTIPLE
: Indicates that multiple occurrences of the delimiter characters should be treated as a single word delimiter. For example, if you were to parse fred::harry, with ":" as the delimiter, then this can be treated as either two words (fred and harry) or three words (fred, blank word, and harry) depending on the mode of this switch.

TOK_PRESERVE_QUOTES
: If a word is enclosed in quotes (single or double) and the quote character is specified in **delims** then the returned words will have the surrounding quotes still in place.

TOK_NUMERIC
: If a word looks like a number then it will be converted to an integer in the return result rather than being treated as a string.

TOK_BACKSLASHES
: Allows backslashes to escape the meaning of delimiter characters.

TOK_DOUBLE_QUOTES
: If specified then double quotes are treated In pairs to enclose words rather than acting as plain delimiters.

TOK_SINGLE_QUOTES
: Same as TOK_DOUBLE_QUOTES but applies to pairs of single quotes.

### Return value

Returns a list of the words in **str.**

### See also

split()(pg. 199).

## top_of_buffer()

### Description

Moves the cursor to the start of line 1 in the current buffer.

### Return value

Returns non-zero if the cursor changed position; zero otherwise.

## top_of_window()

### Description

Moves the cursor to the start of the line displayed at the top of the window.

### Return value

Returns non-zero if the cursor changed position; zero otherwise.

## transfer(buf_id, start_line, start_col, end_line, end_col)

### Description

This function is used to copy a block of text from one buffer to another. buf_id is the buffer identifier of the source buffer. start_line, start_col, end_line, and end_col specify the bounds of the text to be copied.

The copied text is copied to the current buffer at the current cursor position.

### Return value

Returns zero or less if unsuccessful; zero if successful.

## translate([pattern], [replacement], [global], [re], [case], [block], [forward])

### Description

This macro is used to perform string translations within the current buffer. pattern is the string to translate. If re is not specified or is non-zero, then pattern is treated as a regular expression, otherwise it is treated as a normal string. If case is not specified or is non-zero, then the search is performed case insensitively. If block is specified and is non-zero, then the search is limited to the currently highlighted region.

| | |
|---|---|
| **pattern** | is the regular expression or fixed string to search. |
| **replacement** | is the replacement string for the matching pattern. |
| **global** | if specified and non-zero, then every occurrence in the current buffer from the cursor to the end of the buffer is translated. If global is zero, then only the first occurrence is translated. If global is not specified, then the user is prompted for each change. |
| **re** | indicates whether the **pattern** should be treated as a regular expression or not. |

The possible values are:

| | |
|---|---|
| -3 | maximal closure, backward |
| -2 | maximal closure, same as search direction |
| -1 | maximal closure, forward |
| 0 | forward (literal match) |
| 1 | minimal closure, forward |
| 2 | minimal closure, same as in search direction |
| 3 | minimal closure, backward |

For more details on minimal and maximal matching, consult the Programmers Guide.

**forward** if non-zero or omitted, then the translation is done in the forwards direction, otherwise it will occur in the backwards direction.

If **pattern** or **replacement** is not specified, then the user is prompted for the change.

If possible, you are advised to use the re_translate(pg. 160). primitive instead of this one. This primitive is provided for compatability with BRIEF macros, and is sensitive to the current regular expression syntax mode.

### Return value

Returns the number of translations.

### See also

search_fwd(pg. 173)., re_search(pg. 158)., re_translate(pg. 160)., re_syntax(pg. 159).

## translate_pos(x, y, [win_id], [line], [col])

### Description

This macro is designed to map an (x,y) co-ordinate pair into an actual line/column position within a window. x and y are the co-ordinates as returned from the get_mouse_pos ()(pg. 94). primitive representing the place where the mouse was hit.

win_id is filled with the ID of the window upon which the (x,y) position falls. If win_id is not specified then the co-ordinates returned in line and col are with respect to the current window. This allows the calling macro to track the mouse for motion events where the mouse may move outside the boundary of the CRISP window, and is used for example, to force scrolling.

If line and col are specified then these receive the line and column positions within the buffer specified.

This function returns a code indicating what type of object the mouse was hit on.

### Return value

The following returns are possible:

| Value | Meaning |
|---|---|
| 0 | Mouse not pressed inside or on any window (i.e. the status line). |
| 1 | Mouse pressed on left border of window. |
| 2 | Mouse pressed on right border of window. |
| 3 | Mouse pressed on top border of window. |
| 4 | Mouse pressed on bottom border of window. |
| 5 | Mouse pressed inside window. |
| 6 | Mouse pressed on window title. |

### See also

get_mouse_pos()(pg. 94)., process_mouse()(pg. 154).

## trim(string, [delim])

### Description

This function is used to remove trailing characters from the specified string. The default is to remove all tabs, spaces and newline characters. If delim is specified, then all characters in the delim string are removed from the end of string.

### Return value

Returns a copy of string with all trailing white space characters removed. (spaces, tabs and newlines).

## typeof(expr)

### Description

This macro is an alternative way to determine the current type of a polymorphic variable.

### Return value

Returns one of the string constants, "NULL", "integer", "float", "string", or "list" depending on the type of expr.

## umask([mask])

### Description

The umask() function is used to set and/or get the file creation mask. The file creation mask is used when creating files. When a file is created (via write_buffer() or write_block()) the umask is used to mask off certain protection bits. The mask bits are operating system dependent but are by and large the same from one system to another. For example, setting the umask to 022 (octal) causes newly created files to be read and writable by the owner but only readable by others and anyone in the same group.

### Return value

Returns the previous value of the mask.

### See also

write_buffer()(pg. 216)., write_block()(pg. 216).

## uname()

### Description

This macro returns a list of strings corresponding to the result of a *uname* system call. On non-POSIX systems, these strings will be blank (e.g. Windows).

The elements of the list correspond to the various fields in the *utsname* structure. A set of predefined constants are available in **<crisp.h>** which can be used to access this array. All these constants start with **UNAME_.**

### See also

version()(pg. 213).

## undo([move], [write_mark], [redo], [terminate])

### Description

This is the macro used to undo buffer modifications. When called with no arguments it undoes the last operation performed on the buffer, e.g. move the cursor, insert deleted text, unmark area, etc. If the previous operation on the buffer was a macro, or a complex operation, e.g. global translate, then all the buffer modifications performed are undone in one step.

If move is specified and is non-zero, then the undo command undoes all buffer operations back up to the last buffer modification. If the last command was a buffer modification, then setting move to non-zero is equivalent to not specifying it. If the last few operations were simply cursor movements, then move goes back to the last point in the buffer where the buffer was actually modified.

There is a separate undo stack for each buffer. Normally when a buffer is written away the undo stack is discarded so that it is not possible to undo any operations performed on the buffer before the last (write_buffer) operation. If write_mark is specified and is non-zero, then this allows the undo to go back past the last (write_buffer) mark.

It is possible to call (undo) from within a macro, but its use their is highly dubious. It is normally called by the user directly from one of the key assignments.

The redo parameter indicates whether a redo operation should be performed. If it is specified then a redo is performed, otherwise an undo is performed. A redo operation allows an undo to be undone.

If the *terminate* parameter is specified then all the other parameters are ignored. It is used to terminate an undo chain so that a sequence of operations is undo-able in chunks smaller than the entire operation. (This is used by the Search & Translate dialog boxes after each search to allow each modification to be undone rather than the entire history of modifications in one go). This parameter is rarely required since CRiSP will automatically terminate an undo chain on every keystroke. However this may be necessary where the keyboard isn't in use (i.e. commands are coming from a dialog box).

### Return value

Returns -1 on failure (i.e. nothing to undo) or >= 0 if successful.

## uniq_list(list)

### Description

This primitive acts like the Unix *uniq* utility. It scans the source list and returns a new list with all duplicate strings removed. Only consecutive strings which are duplicates will be removed.

### Return value

Returns source list with consecutive duplicate strings removed.

### See also

sort_list()(pg. 198).

## unregister_macro(n, macro, [local], [string])

### Description

This macro is used to remove a macro which has been registered for a particular trigger. (See (register_macro) for a description of the triggers).

If a particular macro has been registered more than once then (unregister_macro) should be called for each register of the macro. (The return from (unregister_macro) can be used in a loop).

The *local* parameter should be set to the same value as that when the macro was registered. This ensures that the correct macro is de-installed. (Valid values are: REGISTER_GLOBAL, REGISTER_BUFFER and REGISTER_SCREEN).

### Return value

Zero if macro was not registered; 1 if macro was registered and has now been unregistered.

### See also

register_macro()(pg. 163)., register_timer()(pg. 166)., unregister_timer()(pg. 211).

## unregister_timer(tid)

### Description

This primitive is used to cancel a timer previously registered via the *register_timer* macro.

Timers are automatically unregistered when they trigger.

### Return value

Zero if the timer is successfully unregistered or -1 if the timer does not exist (e.g. it may have already gone off).

### See also

register_timer()(pg. 166).

## up([n])

### Description

This macro moves the cursor to the previous line in the buffer, maintaining the same column position. If n is specified then the cursor is moved to the n'th previous or n'th following (if n < 0) line.

### Return value

Returns 1 if the cursor moves or 0 if already at the top of the buffer.

### See also

down()(pg. 78)., beginning_of_line()(pg. 51)., page_down()(pg. 150)., page_up()(pg. 150).

## upper(string, [capitalize])

### Return value

Returns a copy of string with all lower case letters mapped to their upper case equivalents.

If *capitalize* is specified and is non-zero then this function will capitalize the first letter of each word and lower case the rest.

### See also

lower()(pg. 140).

## use_local_keyboard([buf_kbd_id], [scr_kbd_id], [flags])

### Description

This macro is used to associate a keyboard map with the current buffer and/or the current screen. A keyboard map associated with a buffer means than when keys are pressed by the user and the associated macros for those keys are looked up, the keyboard map attached to the buffer will be searched first. If no macro definition is available then CRiSP will search the current screen keyboard (if one is defined), and if no screen keyboard is defined then the global keyboard map will be searched.

Note that if a screen keyboard is defined then the global keyboard map will not be searched at all.

*buf_kbd_id* and *scr_kbd_id* are optional keyboard identifiers as returned via the *inq_keyboard()* primitive. Specifying zero for either of these parameters removes the appropriate keyboard definition and if necessary frees the memory associated with it if there are no other references to the keyboard. A value of NULL (or omitted) is acceptable in which case the option is simply ignored.

Buffer local keyboards are usually used to effect things like language sensitive editing modes, e.g. in C mode, the braces **{** and **}** may be associated with a macro to perform C style indenting; obviously it is not a good thing to perform C style indenting on an nroff text buffer. Using local keyboards avoids macros being complicated by having to check the buffer type each time, and also make CRiSP run faster.

Screen keyboards are used in the GUI environment where multiple CRiSP screens are visible and each screen is essentially a separate session from any other CRiSP window (e.g. the help windows use a CRiSP window to display help).

The third parameter, *flags* is optional and if specified indicates the local keyboard should be attached to the buffer in *private* mode. Private mode means that CRiSP will only search the local keyboard associated with a buffer, and will not search the screen or global keyboard if no match is found.

### Return value

Returns 0 if operation successful; -1 if kbd_id is not a valid keyboard id.

### See also

keyboard_push()(pg. 135)., inq_keyboard()(pg. 112)., keyboard_pop()(pg. 134)., assign_to_key()(pg. 47).

## use_tab_char(yes)

### Description

This macro allows the user to specify whether hard or soft tabs should be used. yes is a string expression which should be set to **"y"** if tabs are to be physically inserted into the buffer; otherwise spaces are used instead to fill up to the next tab stop.

If **'yes'** is an integer expression then only the value of the tab toggle is returned.

### Return value

Returns the previous value of the use tab char toggle.

## utime(path, actime, modtime)

### Description

The *utime()* function is used to changed the access and modification times of a file. *path* is the name of the file.

### Return value

Returns zero on success, or -1 on failure. (See global variable *errno* for the reason why the value failed).

## version([maj], [min], [edit], [release], [mach-string], [compile-string])

### Description

This macro is used to get the current version level. The arguments are the names of integer variables which receive the major, minor, edit and release level. If maj is not specified then the version message is displayed on the command line.

Displays the current version on the status line.

Additionally, the *maj* parameter may specify the name of a string variable. In this case, CRiSP returns a fixed string identifying the operating system as one of: DOS, OS/2, UNIX, VMS.

If *mach-string* and/or *compile-string* are specified then these should refer to string variables and will receive strings used to identify the platform on which this version of CRiSP is running, and a unique flag used to identify how CRiSP was built. These fields may be used by macros which need to do something platform specific, and are basically provided to make it easier to identify which version of CRiSP this is in a multi-platform environment.

### Return value

Returns the current version number * 100 plus the minor level.

## wait([status])

### Description

This macro waits for the process attached to the current buffer to die. When the process dies, if status has been specified (the name of an integer variable), then the exit status of the process is placed in it.

This macro may be aborted by pressing a space.

### Return value

Returns -1 if there is no process attached to the current buffer. -2 if user aborted the wait. Returns 0 if process has died.

## wait_for([timeout], expr)

### Description

This macro is used to wait for a sequence of characters to be output by the sub-process attached to the current buffer. timeout specifies how long to wait (in seconds) and if omitted or zero, the wait occurs indefinitely.

expr is either a string expression, representing a regular expression of what is to be waited for, or a list of regular expressions. If it is a list, then a parallel match is performed as each character is read from the buffer. If any of the atoms in the list (all string constants) match the characters from the stream then (wait_for) terminates.

*expr* is in the Unix syntax.

This macro is used by the various process control macros to wait for magic strings, e.g. shell prompts.

### Return value

Returns -1 if there is no process currently attached to the current buffer, or user interrupted wait. Returns 1 if expr is a string expression and the expression matched; returns n if expr is a list and atom n matched.

## wait_for_object(obj_id)

### Description

This primitive is used in conjunction with user-defined dialog boxes, as created with the *dialog_box("list")* sub-function. Normally user defined dialog boxes operate asynchronously, calling the designated callback function when the user clicks on a button, menu item or dismisses the dialog box.

The purpose of this function is to wait until the user operates on the dialog box and return the action without the overhead of calling a callback function. Typically, this would be used for example to prompt from a callback routine for a filename. Rather than complicating the macro with lots of little callback routines, this function can be used to wait for a user action before continuing the macro.

Whilst this macro is in operation, the user will not be able to perform any other actions on the CRiSP editing area, i.e. it makes the dialog box *modal*.

### Return value

-1 is returned if the primitive is not implemented.

-2 is returned if the specified object does not exist.

-3 is returned if the specified object is not a user-defined dialog box.

-4 is returned if the specified object *disappears*. This can happen if the dialog box was created with a callback and the callback routine deletes the object (using *delete_object*).

Otherwise the return value corresponds to the index of the button pressed.

## (watch ...)

### Description

This macro currently is a no-op. It is intended to allow macros to be written which allow other macros to be debugged. The idea is that a macro can monitor any variable to see if it gets changed. The code does not exist, and the ideas have not been thoroughly thought out yet.

This may be supported in a later version of CRiSP if there is sufficient demand.

## while (expr) stmt

### Description

This macro is used to implement a while-loop. The expression expr is evaluated and if non-zero, the statements are executed. Afterwards, expr is tested again. This continues until either a (break) statement is encountered in list, or the expression evaluates to false.

## window_color([bg_color], [fg_color], [win_id])

### Description

This macro is used to set the foreground and/or background colors of the specified window (or the current window if *win_id* is not specified).

If either parameter is not specified then the default foreground or background colors will be used. Color values are those associated with the arguments to the set_color() primitive, e.g. a value of zero will return the background color.

Refer to the following table for valid color values.

| Code | Mnemonic | Description |
|---|---|---|
| 0 | COL_BACKGROUND | Background color of the screen. |
| 1 | COL_FOREGROUND | Foreground color for all windows. |
| 2 | COL_SELECTED_WINDOW | Color of selected window title. |
| 3 | COL_MESSAGES | Normal color of prompts and messages and Line:/Col: fields. |
| 4 | COL_ERRORS | Error message color. |
| 5 | COL_HILITE_BACKGROUND | Color of background for a hilighted area. |
| 6 | COL_HILITE_FOREGROUND | Color of foreground for a highlighted area. |
| 7 | COL_INSERT_CURSOR | Color associated with the insert mode cursor. |
| 8 | COL_OVERTYPE_CURSOR | Color associated with the overtype mode cursor |
| 9 | COL_BORDERS | Color associated with the window borders. |

### Return value

Returns previous color of window (background ORed with the foreground color in the top 16 bits).

### See also

inq_window_color()(pg. 123).

## write(fd, string)

### Description

This primitive can be used to write a string to an external file. The file descriptor, *fd*, is a value previously returned by open().

### Return value

Number of bytes written, or -1, and *errno* set to the reason for the error.

### See also

open()(pg. 149)., close()(pg. 57)., read_from_file()(pg. 160).

## write_block([filename], [append], [keep-region], [pause])

### Description

This macro is used to write out the currently marked region to a file. If filename is not specified, then it is prompted for.

Writing out a marked region does not affect the backup flag or the undo information flag. (See (undo), (set_backup)).

If keep-region is not specified or is zero, then the current anchor is raised. Otherwise it is left alone, e.g. so that you can delete the region afterwards.

If append is specified and is non-zero, then the marked area is appended to the file.

If the filename starts with a '|' character then the data is written to a pipe (via popen()) instead of a file. The data after the '|' is passed as an argument to the popen() call. When writing the block to a command, the command may destroy the screen by writing to stdout or stderr. If this is the case, then the pause parameter can be used to force the screen to be redrawn after execution. If not-specified or is non-zero then the user is prompted to hit <Enter> before continuing.

If the filename starts with either the '>' or '>>' characters then the region is appended to the specified file. (The leading '>' or '>>' are removed).

If filename contains the string '>&filename' then the string will be stripped from the filename and *stderr* will be redirected to filename. This feature is useful when writing to a pipeline and you do not want errors to destroy the screen. (The >& syntax is C-shell syntax and it would be difficult to write portable macros if the macro writer had to worry about what shell environment the user was using).

This primitive knows how to correctly write out a column marked region.

### Return value

Zero or less if the macro was unsuccessful. Greater than zero if the write was successful.

### See also

write_buffer()(pg. 216).

## write_buffer([filename], [and-flags], [or-flags])

### Description

This macro is used to write out the contents of the current buffer to a file. If filename is omitted then the file is written to the filename associated with the current buffer (i.e. as specified via create_buffer). Otherwise filename is used.

If this is the first time that write_buffer has been called for this buffer, then a backup copy of the file will be made. (See the section on making backups for a discussion of the algorithm used). Subsequent calls to write_buffer do not create backups.

Normally this macro will be called either with no parameters, in which case the current buffer with the current mode settings will be written out. You can override various settings by specifying either of the 2nd or 3rd arguments.

The way this works is to compute internally the flags for writing the file; next if the *and-flags* parameter is specified then this set of flags is AND-ed with the internal flags. If the *or-flags* are specified then these are OR-ed in at the last stage.

The flags are documented in the table below. This and-ing and or-ing is designed to allow the macro writer only have to program the bits you care about. For example, to write the current buffer but append to any existing file, simply specify:

```
write_buffer(filename, NULL, EDIT_APPEND_FILE);
```

| Flag | Description |
|---|---|
| 0x0100 | EDIT_APPEND_CR. Append a carriage return to the end of each line. |

| | |
|---|---|
| 0x0200 | EDIT_APPEND_NL. Append a newline to the end of each line. |
| 0x0400 | EDIT_APPEND_CTRLZ. Append a Ctrl-Z at the end of the file. |
| 0x0800 | EDIT_APPEND_FILE. Append the file to any existing file. |
| 0x1000 | EDIT_FORCE. Force file to be written even if CRiSP would ordinarily not attempt to (e.g. because no changes have been made). |
| 0x2000 | EDIT_FORCE_BACKUP. Force a backup to be made for this file. Normally CRiSP will only create a backup the first time a file is saved. |
| 0x4000 | EDIT_TERM_UNDO. Terminate the undo chain. Reserved. |
| 0x8000 | EDIT_FORCE_WRITE. Write the file even if a region is highlighted. If this flag is not specified and you attempt to write the buffer then you will be prompted for a filename to write the region to instead. |

Note: this macro can cause various triggers to go off if the input buffer gets corrupted.

## Return value

Returns zero if file saved (or no modifications were made); returns:

| | |
|---|---|
| -1 | WRERR_DISK_FULL. An error occurred because we ran out of disk space. |
| -2 | Output file could not be created. |
| -3 | The output file was created with a different temporary name but could not be renamed to the target file due to permission errors. |
| -4 | User aborted the attempt to save the file from one of the callback triggers. |
| -5 | The output buffer does not have a valid filename. |
| -6 | The originally loaded file has changed its permissions, size or status on disk. This option avoids potentially losing work when someone else has written to the file whilst we were editing it. |
| -7 | Writing files has been disabled either by the license manager or by the command line switch (-R). |

When an operating system error occurs, you can refer to the global integer variable *errno* for the operating system specific reason for the error.

## See also

write_block()(pg. 216).