
GCLISP 386
GMACS Editor Guide

Chapter 11

Overview of GMACS

GMACS is a full-featured screen display editor modeled after EMACS, the editor created by Richard M. Stallman at the MIT Artificial Intelligence Laboratory. This latest version of GMACS represents a significant upgrade in functionality from previous versions.

LISP Editing and Debugging Features

In its capacity as a specialized editor for writing and debugging LISP code, GMACS has commands that provide the following capabilities, allowing a user to:

- Move the point to the front or back of an s-expression, a list or a function definition
- Evaluate a single function definition or an entire buffer of LISP code
- Compile a single function definition or an entire buffer of LISP code
- Evaluate a LISP expression from within the editor and either place the results in the edit buffer or have them displayed in the type-out window
- Indent new LISP lines appropriately, correct the indentation of an s-expression, and indent comments
- Display the lambda list of a function definition
- Macroexpand an s-expression

- Perform search and replace operations on a group of LISP files at once using tag tables
- Set the package associated with the current buffer
- Exit to a temporary LISP listener and re-enter GMACS without disturbing the current environment

Text Editing Features

GMACS contains many commands that facilitate editing text files. These include commands that enable the user to transpose words, lines and regions; fill paragraphs, move back and forth within a file by paragraphs or pages; perform incremental forward and backward searches and query-replace operations; and insert binary and ASCII files into the edit buffer.

Special Editing and File Control Features

GMACS has several powerful features for effectively manipulating and editing files and buffers. These features include:

- A buffer-edit mode that enables the user to easily visit, compile, kill, load or save the contents of any active buffer
- A directory-edit mode for specifying a directory and easily finding, deleting, viewing, loading, or compiling any of the files within this directory
- The ability to split the screen into two windows, so that two separate buffers or different parts of the same buffer can be edited simultaneously
- Keyboard macros used to create custom commands bound to user-defined keychords
- Command completion for saving typing time in specifying names of buffers, files, commands, etc.

On-Line Help Features

The editor contains four powerful capabilities. You can:

- Display all the GMACS commands or all the LISP functions whose names contain a user-specified substring
- Determine the GMACS command bound to a specific keychord
- Display the documentation for every GMACS command and every LISP function
- Display information about the location of the point

The initial chapters of this guide are designed to give you a global sense of how GMACS works and what it is capable of doing. All the details that you will need to execute a particular command are found in the topical reference chapter later on. That chapter also includes the relevant key bindings for each command. Finally, there is an alphabetical listing of all GMACS commands, along with their key bindings, at the very end of the guide.

Terminology

Before proceeding, you may want to familiarize yourself with some of the more important concepts and terminology used in GMACS.

CURSOR/POINT

The *cursor* appears as a blinking mark (usually an underline or a rectangle) on the edit screen. The *point* is the position between the cursor and the preceding character. Deletions and insertions take place at the point.

EDIT WINDOW

A part of the terminal display screen used for displaying the contents of the edit buffer. The edit window usually occupies all but the bottom three lines of the display screen.

EDIT BUFFER

A temporary storage area created and used by GMACS in your GCLISP

workspace. Since it is possible to have more than one buffer, the *active*, or *current*, edit buffer refers to the one that appears in the edit window.

- ECHO/MESSAGE AREA** The bottom two lines of the edit screen make up an area where (1) edit commands that you type are displayed (or "echoed"), thus allowing you to verify your input, and (2) various informational messages are displayed.
- EXTENDED COMMAND** Any standard GMACS command, regardless of whether or not it is bound to a keychord, can be executed by typing **Alt-X** (the extended command prefix) followed by the command name.
- KILL HISTORY** The kill history is a kill ring, where each entry is put on the ring by a kill command. A new entry becomes the first element in the ring. There is a maximum of fifteen entries, with the oldest (the fifteenth) dropping out to accommodate the newest.
- MINIBUFFER** An area where you are prompted to enter the names of files and other information required by certain commands. The minibuffer appears in the right half of the echo area. The minibuffer accepts GMACS commands and has a kill ring associated with it.
- MODE LINE** The highlighted line of status information appearing near the bottom of every edit screen.
- MODIFIED FLAG** The condition of the buffer with respect to changes. If you have added or deleted data in the current edit buffer since last reading in a file to the buffer, or writing out the buffer to a file, an asterisk appears following the filename in the mode line. Otherwise this space is blank.
- TYPE-OUT** A display of information produced by a GMACS command. It appears in the *type-out window*, a temporary window in

the top part of the edit screen. The type-out window temporarily overlays part or all of the edit window.

Entering GMACS

You have two ways of entering the editor from your GCLISP environment:

- **Ctrl-E**, which has the same effect as the function call `(ed)`
- the `ed` function in one of these forms:

```
(ed "<pathname>")  
(ed t)
```

The pathname argument to the `ed` function may be an actual pathname, a symbol or a string.

When you first invoke GMACS with no pathname (the form `(ed)`), you are placed in an empty edit buffer called MAIN. If you specify a pathname, then the contents of that file are read from disk into a buffer named after the file. The form `(ed t)` gives you a new empty MAIN buffer (and preserves the MAIN buffer from a previous invocation, if any).

Exit and Re-Entry

To leave GMACS and return to the interpreter environment, type **Ctrl-X Ctrl-C**.

When you again invoke GMACS, via **Ctrl-E** or the `ed` function, the GMACS environment of buffers and files will be re-established. If your command is `(ed)`, without a pathname, you will be placed in the buffer where you were last editing, and at the same point in that buffer. If your command is `(ed "<pathname>")`, then GMACS will re-establish the edit environment following the rules of the `FIND-FILE` command.

Protecting the Buffer Contents

When you are editing a buffer, you should write out the buffer to the file often. There are good reasons for this. In the following circumstances, the contents of the GCLISP workspace, including the buffers, are lost:

- When you exit from GCLISP
- When the operating system or GCLISP has to be re-initialized due to some unforeseen problem
- When the power fails or is interrupted

The Edit Screen

This sample screen shows the display after you have loaded a file into GMACS.

```
(defun foo (bar)
  (print foo))

GMACS v4.47 6:03am 4-Aug-86 [LISP fill (GMACS)] C:\PROG.LSP
                                           Alt-H = HELP
```

The "mode line" appears in reverse video at the bottom of your screen just above the echo/message area that contains the minibuffer. It contains:

- The editor name and version number: **GMACS v4.47**
- The time and the date: **6:03am 4-Aug-86**
- In square brackets, the major mode and any active minor modes currently associated with the buffer; if a package is associated with the buffer, its name appears in parentheses: **[LISP fill (GMACS)]**
- If the name of the buffer is different from the filename without its extension, then the name of the buffer; in the above example, the buffername isn't specified
- The pathname of the file currently being edited in the buffer: **C:\PROG.LSP**
- An asterisk (*, the *buffer-status*), if the contents of the current edit buffer have been changed since they were last written out to, or read from, a file
- The help string: **Alt-H = HELP**

See the chapter "Customizing Your GMACS Environment" to see how to control whether or not the version number, time, date and help string are displayed.

Inputting Commands and Characters

While you are in the GMACS environment, everything you type at the keyboard is part of an *edit command*.

An edit command is invoked by typing an alphanumeric key, a *keychord*, a *key sequence*, or a special function key. A key, keychord, or key sequence that invokes a command is said to be *bound* to the command, and vice versa. Most GMACS bindings are the same as the standard EMACS bindings. A GMACS command that is bound can be invoked in one of two ways:

- Type the keychord that is bound to the command
- Type Alt-X followed by the name of the command

If a particular command isn't bound to a keychord, then the only way that it can be invoked is with the Alt-X prefix.

Keychords and Key Sequences

A keychord is represented in print by the symbols of the appropriate keys linked together with hyphens. Thus, Ctrl-F means that the Ctrl key is held down while the F key is pressed. A key sequence is represented by keychords and keys written one after the other without hyphens.

GMACS represents all characters and keystrokes using the 10-bit character codes described by the *COMMON LISP Reference Manual*. GMACS, like most editors patterned after EMACS, accepts keystrokes consisting of a basic ASCII character, with some combination of the Control and Meta bits set (note that the "Control" bit here refers to a high-order bit, not the low-order ASCII Control bit).

Setting the Control and Meta Bits

Generally, the Control bit is set by holding down the Control key, and the Meta bit is set by holding down the Alt key. However, for

some characters the Ctrl key does not set the proper bit. Similarly, it is impossible to set both the Meta and Control bits for any character by holding down the Control and Meta keys simultaneously.

In these cases, the only way to properly set the bits is to use prefix keys. Thus, you can alternatively set the Control bit by typing the prefix keychord Ctrl-^, that is, the keychord Control-Shift-Caret. In those cases where you need to set the Control and Meta bits simultaneously, you can use the prefix keychords Ctrl-C and Ctrl-Z. Note that the echo window displays M- to indicate Meta, C- to indicate Control and C-M- to indicate Control/Meta.

On the IBM keyboard, a number of edit commands bound to keychords or key sequences have also been bound to the "PF" function keys. To invoke one of these commands, you do not have to use the keychord or key sequence, but can use the function key instead.

The Minibuffer

Many GMACS commands prompt the user for the name of a buffer, file, directory, s-expression, DOS command or GMACS command. The area where the user provides this information is beneath the mode line in the right-hand side of the echo area known as the *minibuffer*. Although the minibuffer is only two lines long, it acts like a regular buffer in that it will accept nearly all of the GMACS editing commands. This is particularly useful when you need to provide an s-expression, since most of the LISP editing features can be used to enter and debug the expression.

Frequently, when you are prompted to enter information into the minibuffer, you will notice a default provided on the left-hand side of the echo window. If you type a <Enter>, GMACS will execute the default. The default specified is the top entry of the kill ring associated with the current command. GMACS maintains a history of user input for each family of commands that requires minibuffer input (for example, there is a separate kill ring for commands requiring directory names and another for commands requiring buffer names).

Saving Keystrokes: Command Completion

In many cases, the user need not completely specify minibuffer input. This is because GMACS includes a feature known as "command completion," which saves time by allowing the user to type

only enough characters to specify the required information. GMACS will be able to use the information typed to complete the command. Commands that include the command completion feature will display (**cmpl'g**) in the left-hand side of the minibuffer. Here is how command completion works.

For commands that prompt for a buffername or a pathname, the user need only specify the minimum number of specifying characters and then type a space. GMACS will automatically complete the command by choosing the name that matches the specification. If more than one match exists, GMACS completes as much of the name as possible and then displays all the possible matches in the type-out window. Typing **<Enter>** after typing a name invokes only what is exactly typed.

For example, let's say the list of current buffer names includes PROGNEW, PROGOLD and PROG.

- Typing PROGO and a space, completes PROGO to PROGOLD
- Typing PROG and **<Enter>**, selects PROG
- Typing PRO and a space, completes PRO to PROG and displays PROG, PROGOLD and PROGNEW in the type-out window
- Typing PRO and **<Enter>**, results in GMACS creating a new buffer named PRO.

Command completion also works with **Alt-X** extended commands and the **SET-VARIABLE** command. Similar to command-completion is the **DO-IT-AGAIN** command, which causes the previous command to be repeated.

Major Modes

Every GMACS buffer has a major mode and zero or more minor modes associated with it. A GMACS buffer must have one, and only one, major mode associated with it at any point in time.

The major mode in effect customizes an editing environment, establishing a number of crucial editing parameters that affect such things as paragraph filling and indentation. There are three predefined major modes: Normal, Text and Lisp. Because normal mode is very much like text mode (without auto-fill), we really need

only distinguish between text and lisp modes. The former is designed for editing English-language text, the latter for LISP code.

The major mode of a file buffer is automatically determined by the extension at the end of a filename. A buffer associated with a file with a `.l` or `.lsp` extension is set to LISP mode, and buffer associated with a file with a `.tex`, `.mss` or `.txt` extension is set to text mode.

However, if there is an attribute-list in the first line of your file, GMACS will recognize it and set the major mode accordingly. The line containing the attribute-list must be preceded and followed by this symbol: `-*-`. GMACS will recognize COMMON LISP as a mode if it is specified in the attribute-list, but functionally it is identical to LISP mode. The following is an example of an attribute-list that specifies both the mode and the package:

```
-*- Mode:LISP; Package:GMACS; -*-
```

Four attribute-list commands allow you to make changes in the mode and package settings:

- To change the mode and the package settings for the current buffer back to those specified in a file's attribute-list, use the `PARSE-ATTRIBUTE-LIST` command
- To update the attribute-list using the current mode and package settings for the buffer (the ones displayed in the mode line), use `UPDATE-ATTRIBUTE-LIST`
- To update just the package attribute use `UPDATE-PACKAGE-ATTRIBUTE`
- To update just the mode attribute in the attribute-list use `UPDATE-MODE-ATTRIBUTE`

If an attribute-list does not exist and the file extension does not have a mode associated with it, GMACS will set the mode to normal, or to whatever you set `*default-major-mode*` to be in your GMACS initialization file, `gmacsini.lsp`. You can also use the `SET-MODE` command to change the major mode associated with a buffer.

Minor Modes

In addition to the major mode, one or more minor modes may be active. The possible minor modes are listed as follows (the word in

parentheses is the one that appears on the mode line when selected by the user):

- **AUTO-FILL** (`fill`), which makes lines of text automatically wrap when they reach column 65 (the default setting)
- **AUTO-FILL-COMMENTS** (`filcom`), which automatically wraps comments in LISP code
- **LISP-INTERACTION** (`linter`), which turns on **LISP-INTERACT**, a command that allows you to evaluate an s-expression and display the result

A numeric prefix argument can be used to turn a minor mode on and off. A positive argument turns it on, a negative argument turns it off. Called with no arguments, a minor mode acts like a toggle.

In addition to these three commands, there are three others that allow you to customize some of the default parameters associated with the way input is formatted.

- **SET-FILL-COLUMN** sets the column at which wrapping takes place
- **SET-COMMENT-COLUMN** sets the column at which comments begin
- **SET-FILL-PREFIX** sets the characters that will appear on every new line to the left of the point when fill-mode is on

GMACS Help

At any time while in the GMACS environment, you can invoke on-line help about GMACS by typing Alt-H, which displays in the minibuffer a short menu of options and how to invoke them:

```
| GMACS v4.47 [LISP fill] C:\PROG.LSP
| Help (A,D,K,L,M-A,M-D,M-L,?):
```

Typing ? invokes a display of more detailed option descriptions:

```
| These kinds of GMACS on-line help are available. To invoke one
| of them, type the help key (M-H) followed by the indicated key
| (one of A, D, K, L, M-A, M-D, M-L, ?):
```

- A "Apropos" - Displays the keychords for all GMACS commands that contain a specified string. Prompts for the string.
 - D "Documentation" - Displays documentation on all GMACS commands containing a specified string. Prompts for the string.
 - K "Keychord binding" - Displays the GMACS command bound to a specified keychord. Prompts for the keychord. For compatibility with other editors, M-H C does this also.
 - L "Listback" - Displays the last 50 keystrokes and their command bindings.
 - M-A "LISP Apropos" - Like doing M-A at top level.
 - M-D "Function Documentation" - Like typing M-D at GCLISP top level.
 - M-L "Lambda List" - Like typing an M-L at GCLISP top level.
 - ? Displays this guide and reads another key.
-

The help guide appears in a type-out window. Once you have finished reading a help screen, remember to use only the spacebar to

continue, since any other input will be executed as a GMACS command.

Aborting GMACS Commands

You can abort any GMACS command by typing **Ctrl-G**, the keychord bound to **ED-BEEP**. This command will abort the current command and ring the bell.

Chapter 12

Manipulating Buffers and Files

Many GMACS commands facilitate the manipulation of buffers and files. These commands enable the user to do things like compile, load, delete, evaluate and save the contents of files and buffers. The user can invoke these commands directly or by first entering either buffer-edit mode or directory-edit mode.

At the outset, it is important to keep in mind the difference between a buffer and a file. A buffer is a space set aside in memory to hold data. A file is a space set aside on a disk to hold data. Just as it's possible to have more than one file on a disk, so is it possible to have more than one buffer in memory.

How Buffernames and Filenames are Related

When you read a file from disk into a GMACS buffer, the name of the buffer will be the name of the file without its extension. For example, executing a FIND-FILE on C:\gclisp3\prog.lsp, will result in a buffer being named **prog**. The buffername does not appear in the mode line unless there is a problem in deriving a buffername from a filename:

- If a buffer already exists with the same name, then GMACS will prompt you for a new buffername. If you choose not to specify a buffername, then a default name appears in the mode line. The default name is the old name plus a number designation. In the above example, **prog-1** would be given as the default name, or **prog-2** if **prog-1** already existed.
- If the buffername is different from the filename, then the buffername will appear in the mode line just before the filename.

- If GMACS has newly created a buffer and the buffer is empty, then there is no file associated with the buffer. In this instance, the buffername MAIN is assigned (or MAIN-1 if MAIN is already taken), and it, along with the designation "null pathname," appears in the mode line.

Reading and Writing Files

There are two main ways of reading the contents of a file into a buffer:

- The FIND-FILE command is used to read a file into some buffer other than the current buffer, or into a new edit buffer. If a buffer exists already associated with this file, it is selected as the current buffer and nothing is read into it. The point is positioned where it was last located when that buffer was last the current buffer. If, on the other hand, no buffer exists, then a new buffer is created and the file is read into the new buffer, which is named according to the conventions described above. The FIND-FILE command accepts wildcard path names; every file that matches the specified pathname will be read into a different buffer for each file.
- The READ-FILE command is used to read a file into the *current* buffer. Whatever is already in the current buffer is written over and lost. If you have made changes to the current buffer since you last wrote it to disk (via SAVE-FILE or WRITE-FILE), READ-FILE warns you and offers you the opportunity to cancel the command.

Writing a File

After you have edited a file in a buffer, or typed text into an empty buffer, you may want to transfer the buffer's contents to a disk file (unless you decide not to save the editing you have done). GMACS provides three main options:

- The SAVE-FILE command will write out the contents of the current buffer to its corresponding file on disk.

- If you do not want to replace an existing file with the contents of the buffer, use the `WRITE-FILE` command.
- The `DUMP-BINARY-FILE` command will dump a file to disk in binary format. Unlike `WRITE-FILE`, this command will not change any of the information in the mode line.

Working with Buffers

GMACS has many commands for handling buffers. These commands can be broken down into 3 groups.

1. Commands for manipulating buffers. These provide various ways for listing, selecting, deleting, compiling and evaluating buffers. They include:
 - `LIST-BUFFERS` for displaying all the current GMACS buffers and their associated filenames
 - `SELECT-BUFFER` and `SELECT-PREVIOUS-BUFFER` for switching from one buffer to another
 - `KILL-BUFFER` for deleting a buffer entirely or `UNLOAD-BUFFER` for releasing the memory associated with a buffer but preserving its name and file association
 - `COMPILE-BUFFER` and `EVAL-BUFFER` for compiling or evaluating an entire buffer of LISP code
2. Commands for altering the attributes associated with a buffer. Attributes of a buffer include its mode and package association, its read/write status, its modified flag, and its default settings.
 - **MAJOR MODE:** The major mode of a buffer can be either normal, text or lisp. `SET-MODE` is used for changing the major mode of a buffer. In addition, the major mode can be changed with three more specific commands: `NORMAL-MODE`, `TEXT-MODE`, and `LISP-MODE`.

- **MINOR MODE:** Minor modes are determined by AUTO-FILL, AUTO-FILL-COMMENTS, and LISP-INTERACTION.
- **PACKAGE ASSOCIATION:** The package associated with the current buffer can be changed with SET-BUFFER-PACKAGE, which allows you to assign the buffer to either an existing package or a new package.
- **ATTRIBUTE-LIST:** Changes in mode and package association vis-a-vis the attribute-list of a file can be made with the series of attribute-list commands detailed above.
- **READ/WRITE STATUS:** By default, most buffers can be read and written to (the exceptions are the special BUFFED and Dired buffers, which are read-only). The read/write status of a buffer is changed with two commands: BUFFER-READ-ONLY, which makes a buffer read-only and BUFFER-READ-WRITE, which gives read/write status to a buffer that is read-only.
- **MODIFIED FLAG:** Since GMACS marks buffers as either modified (with a * in the mode line) or unmodified (no *), it may sometimes be useful to change the status of this flag, especially in cases where you may have unintentionally modified a buffer or made and unmade changes and do not want to risk saving the buffer to a file. To reset the modified flag, use UNMODIFY-BUFFER
- **DEFAULT SETTINGS:** There are several ways of changing the default settings associated with GMACS commands (and stored as variables in the file `gmacsini.lsp`). Permanent changes must be made by editing the `gmacsini.lsp` file itself. Temporary changes made to a global variable are done using SET-VARIABLE.

3. Commands contained within a buffer-edit mode that facilitates buffer manipulation. The BUFFER-EDIT command places you in a special buffer that lists all the buffers and pertinent information about each one under these headings:

Stat Mod Buffer: Pathname: () => Unloaded

"Stat" gives the read/write status as either RW or RO. "Mod" indicates with an asterisk (*) whether the buffer has been modified. You can perform various operations on the buffers listed by placing the cursor on the line containing the buffername you wish to edit and then typing one of the following command letters, which will allow you to:

C	compile it
G	go to it
K	kill it
L	load it into the interpreter
S	save it
?	display a list of all possible buffer operations
E	exit the buffer-edit mode

Directory Operations

While in GMACS, you can read or write files in the working directory, or perform other file-handling operations. You may want to change the working directory, or you may want to examine the contents of this directory or of some other directory. Two commands enable you to do this: DISPLAY-DIRECTORY, which lists the names of files in a specified directory, and CHANGE-DIRECTORY, which changes the working directory.

Like the special editing mode invoked by BUFFER-EDIT, the command DIRECTORY-EDIT allows you to perform operations on files using a single-letter command. After being prompted for a pathname, you are placed in a buffer, which contains a list of files, along with other information under these headings:

<u>D</u>	<u>FileName</u>	<u>Creation-Date</u>	<u>Creation-Time</u>
----------	-----------------	----------------------	----------------------

"D" is the "delete column," which, if marked with an asterisk (*), notes that a file has been deleted. As with buffer-edit mode, you can perform a series of operations. Once a file is selected, you can:

C	compile it
D	delete it
F	find it
L	load it into the interpreter
U	undelete it
V	view it
?	display a list of all possible directory operations
E	exit the directory-edit mode

Sometimes it's desirable to have deleted files not be quite so permanently disposed of. By default, GMACS sets the variable ***dired-trashcan*** (located in the `gmacsini.lsp` file) to nil, thus causing all files to be irrecoverable. However, if the value of ***dired-trashcan*** is a string holding the name of a directory (with a trailing `\`), a delete command will do the following:

- Copy the file to the trashcan directory
- Delete the file from the directory being edited
- Mark the delete column in the listing of files in directory-edit mode

A subsequent "undelete command" will perform the inverse operation on a file that has been marked for deletion. Like all trash receptacles, the ***dired-trashcan*** directory must be periodically cleaned out by the user.

DOS-Related Commands

GMACS includes many commands that enable the user to access DOS from within the editor or exit temporarily to DOS after performing a particular operation.

The command EXECUTE-DOS-COMMAND prompts the user for a DOS command and displays the output from the command in the type-out window. Similarly, the command INSERT-DOS-OUTPUT prompts the user for a DOS command and inserts the output from the command into the buffer at the point. The PUSH-TO-DOS command pushes the user to the current DOS directory. The SAVE-FILES-PUSH-TO-DOS first saves all buffers that have been modified and then pushes to DOS. The commands EVAL-AND-EXIT and COMPILE-AND-EXIT enable the user to immediately push to DOS after evaluating or compiling a LISP definition.

Tag Tables

Tag tables enable you to manipulate a program that consists of multiple source files as a single unit. A tag table is a list of files that you wish to treat as a unit.

The first step is to create a tag table that contains the files that comprise your program. The TAGS-ADD-FILE and TAGS-ADD-FILES commands can be used to create and add to the current tag table. Similarly, the TAGS-REMOVE-FILE command will remove a file from the current tag table.

Each of the files in the tag table must have a corresponding index (.lsx) file. The TAGS-MAKE-INDEX command will create index files for each of the files in the current tag table, load them into memory and save them on disk. Note that these files must be loaded into memory when any tags search or replace operation is performed. The TAGS-INDEX-FILE command can be used to make an index file for a single file that you have added to the tag table. Both of these commands generate a new index file, regardless of whether or not the source code file has been changed since the last index file was made.

Once you have created the tag table and the associated index files, you can perform search and replace commands such as TAGS-REPLACE-STRING, TAGS-SEARCH, TAGS-FIND-DEFINITION and TAGS-QUERY-REPLACE. If you temporarily suspend one of these operations to do local editing on one of the files, you can use the TAGS-CONTINUE-MAP command to continue where you left off.

By default, a TAGS-REPLACE-STRING or a TAGS-QUERY-REPLACE will make changes only to the buffer(s) involved; typically, you need to explicitly save the buffers in order to make permanent changes to the corresponding files. However, if you want GMACS to automatically save these changes when they are made, you can do so by setting the global variable *save-on-tags-map* to T in

your `gmacsini.lsp` file. By default, GMACS will set this variable to `nil`.

If you wish to use a group of files as a unit more than once, you should use the `TAGS-SAVE-TABLE` command to save the current table on disk. Then, the next time you wish to use the table, you would load it into memory using the `TAGS-LOAD-TABLE` command. This command should be followed by the `TAGS-LOAD-INDEX` command, which will load all of the existing index files that correspond to the files in the current tag table.

You may want to use more than one tag table in a given session. If so, the `TAGS-USE-TABLE` command will allow you to switch back and forth between tag tables.

(In this release, the tag-table indices created by `TAGS-MAKE-INDEX` are not reliable.)

Chapter 13

Editing Text

Although this chapter deals mostly with commands that aid in writing and editing English text, many commands are useful in writing and editing LISP code. Thus "text" can be seen as any collection of alphanumeric characters and symbols capable of being manipulated in a buffer. In a later chapter, we will discuss commands that are used exclusively for handling LISP code.

Whether you are editing text or code, you should make sure that the major mode associated with the current buffer is set properly so that the default settings for delimiter characters and fill columns will be appropriate.

Cursor Motion Commands

In order to understand how the GMACS cursor commands work with words, lines, paragraphs, pages, etc., one needs to know how these different text items are delimited in GMACS. The following is a list of different text items together with a description of how they are defined in Text Mode:

Word	To GMACS, a <i>word</i> is any string of <i>alphanumeric</i> characters (that is, letters or digits). So the end of a word is marked by any other character: a punctuation symbol, special character, or <i>white space</i> (a space, tab, or newline character).
Sentence	The end of a sentence is delimited by a period (.), question mark (?), or exclamation mark (!).
Line	To GMACS, a <i>line</i> consists of the sequence of characters from one newline character to the next

(including the ending newline). There may be more characters in this line than can fit in a single line of the display screen. Then more than one display line will be used to display the line. In the edit buffer it is called a *wrapped line* on the display, because the line "wraps around" the end of one display line and continues on the next. GMACS informs you that a display line is wrapped by placing a backslash in the right-most display position.

Paragraph

GMACS interprets a blankspace or a tab at the beginning of a line as the start of a paragraph. Also, `\begin` and `\end` as well as `#|` and `|#` delimit the beginning and ending of paragraphs when they appear at the front of a line. Paragraphs are also delimited by blank lines.

Page

The `^L` character delimits pages when it appears at the front of a line.

GMACS has a whole series of commands for moving the point anywhere in the edit buffer:

- By character with FORWARD-CHAR and BACKWARD-CHAR.
- By word with FORWARD-WORD and BACKWARD-WORD.
- By sentence with BEGINNING-OF-SENTENCE and END-OF-SENTENCE.
- By line with NEXT-LINE and PREVIOUS-LINE, which move the point to the same column in an adjacent line. Two special commands for LISP code, BEGINNING-OF-NUMBERED-LINE and END-OF-NUMBERED-LINE, work with either the current line or a specified line number.
- By paragraph with FORWARD-PARAGRAPH and BACKWARD-PARAGRAPH.
- By page with FORWARD-PAGE and BACKWARD-PAGE, which leave the point at the top of the page centered in the edit window.
- By buffer length with BEGINNING-OF-BUFFER and END-OF-BUFFER.

Inserting Text

The simplest editing consists of inserting individual characters in an edit buffer. Insertion always occurs at the *point* between the character above the cursor and the character immediately preceding it. A non-printing character such as a space, tab, or newline is like any other character in this regard. For example, the newline character (produced by <Enter>) doesn't show in the screen display, but it is in the buffer like any other typed data.

GMACS provides four special commands for inserting text.

- QUOTED-INSERT is used to insert as text those characters which otherwise act as editing commands, such as control characters.
- INSERT-FILE inserts the entire contents of an ASCII text file at the current location of the point. INSERT-BINARY-FILE performs the same function for a binary file.
- INSERT-COMMAND-NAME prompts for a keychord and then inserts the name bound to that keychord.

Deleting Text

To delete the character at the cursor position use DELETE-CHAR by pressing **Ctrl-D**. The character at the cursor disappears, and all characters following the cursor move one character backward. To erase a character you have just typed (i.e, to the left of the cursor) use either RUBOUT or RUBOUT-HACKING-TABS. Both delete characters to the left of the cursor, but differ in how they treat tabs. When RUBOUT deletes a tab, it wipes out the full tab-width of spaces, changing this:

```
Column 1   Column 2
```

to this:

```
Column 1Column 2
```

When RUBOUT-HACKING-TABS deletes a tab, it converts the tab to the appropriate number of spaces and then deletes one of the spaces, so that, this:

```
Column 1   Column 2
```

becomes this:

Column 1 Column 2

The default keybindings for these two commands may cause some confusion because the <Rubout> key is in fact bound to RUBOUT-HACKING-TABS, not to RUBOUT, and so converts tabs to spaces before deleting just one.

Four special-purpose commands can be used to delete multiple blank spaces in the text.

- DELETE-HORIZONTAL-SPACE deletes any spaces or tabs on either side of the point.
- JUST-ONE-SPACE does the same thing but leaves "just one space."
- DELETE-BLANK-LINES collapses the number of blank lines above and below the point to one line.
- DELETE-INDENTATION has a special use in deleting indentation (see the section "Understanding Indentation" for more details).

Numeric Arguments (Repeat Counts)

You will often want to execute a GMACS command a certain number of times one after the other. For example, you may want to move the cursor forward exactly 65 characters. It would be a nuisance to repeat a cursor-motion command this often. Instead, you can invoke a single command with a *numeric argument* which specifies how often the command is to be repeated.

There are two ways to prefix a command with a numeric argument.

- Type **Ctrl-U** followed by the numeric argument and then the command: **Ctrl-U <number>**.
- Type **Alt** followed by the numeric argument and then the command: **Alt-<number>**.

In these cases, the number is called the *repeat count* for the command that follows it. For example, to advance the cursor 65 characters, type: **Ctrl-U 65 Ctrl-F**. Remember that the ordinary characters of the keyboard are self-inserting input: typing the character A means

"insert the character A." Thus, to insert a row of 65 asterisks into the buffer, type: **Ctrl-U 65 ***.

Ctrl-U alone, without a numeric argument, performs the command 4 times. To advance the cursor 4 characters, type: **Ctrl-U Ctrl-F**. Any additional **Ctrl-U** that *follows* the repeat-count argument multiplies the repeat count by 4. This input advances the cursor by 64 characters: **Ctrl-U 16 Ctrl-U Ctrl-F**.

With commands that have an obvious opposite command, a negative prefix argument will cause the opposite command to be executed the specified number of times. For example, an argument of -5 to the **SCROLL-UP** command will cause the screen to be scrolled *down* 5 lines.

In general, with commands that act like toggles, such as **AUTO-FILL**, a positive prefix argument will turn the command on and a zero or negative prefix argument will turn the command off. If a command uses a numeric prefix argument in a non-standard way, it is documented in the command summary.

Filling Paragraphs

In text mode, all paragraphs are automatically filled. That is, the default state of auto-fill is on, a situation indicated in the mode line by the word "fill" in the minor-mode position within the square brackets.

Paragraph filling is another way of saying that text lines are wrapped when they reach a predetermined point in the line. Here is how it works: when a non-white-space character is typed past the "fill column" (set by default to column 65) at the end of the line, the line is broken at the nearest word and a newline is automatically inserted at the break. If auto-fill mode is not on, lines will break and wrap as soon as they reach the end of the display. The default setting of the fill column can be changed with the **SET-FILL-COLUMN** command.

If auto-fill mode is not on, it can be turned on by calling **AUTO-FILL** with a positive numeric prefix argument. When called with 0 as the argument, **AUTO-FILL** is turned off. When called with no argument, **AUTO-FILL** acts as a toggle, switching alternately from on to off.

The auto-fill feature works only when new text is typed into the buffer. Unwrapped lines can be eliminated with the **FILL-PARAGRAPH** command. When the **FILL-PARAGRAPH** command is

invoked, lines extending past the fill column in the current paragraph are broken at the first space before the fill column and wrapped onto the next line. Any leading white space is deleted.

Inserting New Lines

There are four different ways for breaking the current line and generating a new line. The most familiar way is to press `<Enter>`, which is bound to the command `NEWLINE` and which can also be invoked by pressing `Ctrl-M`. The current line is broken at the point and the cursor moves to the beginning of the new line. Thus,

This is an ordinary sentence.

becomes

This is an ordinary
sentence

A related command is `OPEN-LINE`, invoked with `Ctrl-O`. This command inserts a newline character at the point, and leaves the point where it is, before the newline character:

This is an ordinary sentence.

now becomes

This is an ordinary _
sentence.

Two additional commands combine the newline function with the indent function, which, as with indenting by itself, works differently in Text mode and Lisp Mode. `NEWLINE-INDENT` operates like `NEWLINE` but moves the cursor to the first indented position of the new line. In text this position is determined by the first occurring blank space of the line above the new line.

This is an ordinary sentence.

becomes

This is an ordinary
sentence.

`OPEN-INDENTED-LINE` operates like `OPEN-LINE` but leaves the cursor where it is and indents the new line directly beneath the cursor.

This is an ordinary sentence.

becomes

This is an ordinary sentence.

Understanding Indentation

GMACS includes several commands that facilitate indenting text. First of all, we ought to understand the distinction between a tab and indent. A tab is a position somewhere on a line. In GMACS the `<Tab>` key moves eight spaces and is bound to `TAB-TO-TAB-STOP`. You can change the default 8 spaces to any number you like by using `SET-VARIABLE` to alter the value of `*tab-stop-width*`.

Indentation fills a portion of a line with blank spaces. It is invoked with `Ctrl-I`, which, when Text is selected as the Major Mode, is bound to `INDENT-TEXT-LINE`. `Ctrl-I` causes the current line to indent to positions just to the right of blank spaces in the previous line. The real usefulness of this feature can be seen when we want to align text in columns of varying widths:

Col 1	Col 2	Col 3	Col 4
—			

Pressing `Ctrl-I` repeatedly now aligns a row of numbers correctly:

Col 1	Col 2	Col 3	Col 4
3	45.00	198	789

In Lisp mode, `Ctrl-I` is bound to `INDENT-LISP-LINE` and uses a different algorithm for indenting that is appropriate for typing LISP code.

In addition to these two commands, there is a third that performs a special indenting function. `INDENT-RIGIDLY` indents every line in the region between the point and the mark one space (a numeric argument indents the region the specified number of spaces). A block of text like this:

This is a nice block
of text. It consists
of four lines and two
sentences.

becomes this when we indent-rigidly 5 spaces:

```

    This is a nice block
    of text. It consists
    of four lines and two
    sentences.
  
```

Finally, there are a number of associated indentation commands. **BACK-TO-INDENTATION**, useful in Lisp Mode, moves the cursor to the first character in a line that is not a space.

DELETE-INDENTATION for deleting the newline character and any indentation at the beginning of the current line. The current line is appended to the preceding line, with the addition of a single blank space.

```

    This is an ordinary sentence.
    This is another sentence.
  
```

becomes

```

    This is an ordinary sentence. This is another
    sentence.
  
```

Setting Uppercase and Lowercase

To aid you in formatting text, GMACS has commands for setting the case of alphabetic characters to uppercase (capitals) or lowercase (small letters):

- **UPPERCASE-WORD** and **LOWERCASE-WORD** put all the letters of the word to the right of the point in either upper- or lowercase.
- **BACKWARD-UPPERCASE-WORD** and **BACKWARD-LOWERCASE-WORD** search for the first word to the left of the point and put the entire word in either upper- or lowercase.
- **UPPERCASE-INITIAL** and **BACKWARD-UPPERCASE-INITIAL** capitalizes the first letter of the word to either the right or left of the point and put the rest of the letters of that word in lowercase.

- UPPERCASE-REGION and LOWERCASE-REGION puts all the letters in the region in either upper- or lowercase.

Manipulating Regions and Marks

The editing operations described so far have included insertions of characters, words, and lines. These are natural units to manipulate with the editor. Often, however, it's convenient to manipulate larger blocks of text: to move, copy, or delete paragraphs or other large units.

GMACS enables you to define and manipulate text in blocks of any size, called *regions*. Unlike a character or a word or a line, a region is not "naturally" defined: it is not delimited by blanks or newlines, for example. The limits of a region are completely up to you.

For example, if you wanted to delete a block of text from the buffer, you would do so in the following way. First, position the point in the place in the buffer that you wish to be the start of the region, and use the SET-POP-MARK command to set a mark at the point. The mark doesn't show in the edit window, but the message "Mark set" appears in the message area. Then, move the point to the place in the buffer that you wish to be the end of the region, and execute the KILL-REGION command to kill the region between the mark and the point.

The following is a list of commands that operate on regions, that is, the area between the current mark and the point:

- COPY-REGION copies the region between the point and the mark to the top of the kill history.
- KILL-REGION moves the text between the current mark and the point to the top of the kill history.

- **TRANSPOSE-REGIONS** uses the top three marks on the mark stack and the point to delineate two regions, and then transposes these regions. Here's how it works. Let's call the two regions Fred and Joan:

M1 ----- Fred ----- M2 ... M3 ----- Joan ----- P

First, the markers (M1, M2, M3) and the point (P) are sorted in the order of their location in the file. Then Fred is transposed with Joan; that is Joan is placed where Fred was, and Fred is placed where Joan was. If there are only two marks on the mark stack, like this:

M1 ----- Fred ----- M2 ----- Joan ----- P

then the markers along with the point are used to define two regions separated by the middle of the three markers (M2). The two regions are transposed around the middle marker.

- **UNTABIFY-REGION** converts all the tabs in the region between the point and the mark to the appropriate number of space characters.

In addition to the **SET-POP-MARK** command, there are several other commands that will set a mark in the buffer: **MARK-BEGINNING-OF-BUFFER**, **MARK-END-OF-BUFFER**, **MARK-PAGE**, **MARK-SEXP**, and **MARK-WHOLE-BUFFER**, which marks the end of a buffer and places the point at the top of it.

In addition to using the current mark to define a region, you can also specify a sequence of marks for immediate or later use. **GMACS** keeps a list of these, the *mark pdl*--"pdl" for "push-down list." You can add a mark to this list, throw away a mark from the list, or recover and use a mark which is currently on the list.

The **SET-POP-MARK** command can be used in conjunction with one or more **Ctrl-U** prefix arguments to manipulate the mark pdl in the following ways:

- Alone, **SET-POP-MARK** defines a mark (at the current location of the point) and puts the mark on the top of the stack. Each mark already on the stack is "pushed down": the top mark becomes the second, the second becomes the third, and so on. The top mark is also called the *current mark*.

- With **Ctrl-U**, **SET-POP-MARK** gives you the top mark: in other words, it gets the current mark and places the point at that position. The mark is taken off the stack. All the remaining marks, if any, are moved up one; the former second mark is now the current mark, etc.
- With **Ctrl-U Ctrl-U**, **SET-POP-MARK** takes the current mark off the stack without placing the point at the mark. All the remaining marks, if any, are moved up one.

Besides using a mark to delimit a region, you may want to use a mark simply as a way to mark a point in the buffer to which you will want to return at some later time for further editing.

One additional command enables you to move the point quickly to the current mark, without changing the region and without discarding the mark. **EXCHANGE-POINT-AND-MARK** exchanges the point and the current mark.

Killing and Recovering Text

As noted previously, the commands **Ctrl-D** and **<Rubout>** delete individual characters from the buffer permanently; text deleted in this way cannot be recovered.

All other commands that remove or copy text save the text so that it can be recovered and copied to another part of the buffer. Text is saved in a "ring" where the most recent entry becomes the first element in the ring and all the other entries are pushed down. There is a maximum of fifteen entries, with the oldest (the fifteenth) dropping out to accommodate the newest. The **DISPLAY-KILL-HISTORY** command will display in a type-out window all entries contained in the kill history. Only the first line of each entry in the kill history is displayed; subsequent lines are represented by ellipses (...). An arrow marks the current top entry.

Thus, if you delete three of the following four lines, one after the other:

```
This is line one.  
This is line two.  
This is line three.  
This is line four.
```

the kill history will look like this:

```

1: This is line two.
-> 2: This is line three.
3: This is line one.

```

Note that the entries are not *stacked* in the order they were killed. The reason for this is that the kill history is a ring, with the arrow indicating the top entry. Subsequent entries follow, not in a linear fashion but in a circular one. Thus the next entry after the "last" entry in the list is the "first" entry.

GMACS contains commands appropriate for performing either deleting or copying text. Commands that delete the text include KILL-WORD, BACKWARD-KILL-WORD, KILL-LINE, BACKWARD-KILL-LINE, and KILL-REGION. Commands that copy the text to the kill history without deleting it from the buffer include COPY-LINE and COPY-REGION.

Each of these commands creates a new entry in the kill ring, but there may be times when you would like to take pieces of text from various parts of the buffer, join them together and then reinsert them as one block. Such a task can be done with the commands already described but at the cost of repetitive keystrokes. The command APPEND-NEXT-KILL provides an easier alternative. Instead of creating a new top entry to the ring, this command attaches the killed text to the existing top entry, making them a single larger entry but preserving the order and number of the entries in the ring. Because the killed text is attached to the end of the existing top entry, it is said to be "appended" to it. Thus, if we take

```
This is line four.
```

and then use APPEND-NEXT-KILL before killing it, the same kill history that we used above would look like this:

```

1: This is line two.
-> 2: This is line three.This is line four.
3: This is line one.

```

Note that you must use APPEND-NEXT-KILL immediately before invoking the kill or copy command. Any intervening cursor-motion commands will nullify the effect of the APPEND-NEXT-KILL command.

Now it may happen that instead of attaching the killed line to the end of the top entry, you would want to attach it to the beginning. This is called "prepending." There is no special command to prepend a piece of killed text. To prepend text, you still use APPEND-

NEXT-KILL but in conjunction with a "backward" kill operation, which has two forms:

- Using a "backward" command like BACKWARD-KILL-WORD and BACKWARD-KILL-LINE
- Performing a kill or copy operation backward from the point to the mark

Thus, if in the previous example, we had prepended line four instead of appending it, the kill history would look like this:

```
1: This is line two.  
-> 2: This is line four.This is line three.  
3: This is line one.
```

Recovering Text from the Kill History

Now that we have looked at the different kill and copy commands that add entries to the kill history, let's take a look at how the text can be recovered from the kill history and put back in the buffer.

The YANK and YANK-POP commands recover entries from the kill history. Both of these commands *copy* a text entry from the kill history to the current point in the edit buffer. Neither command changes either the contents or the order of the entries in the kill history. Remember that the kill history is only changed by the progressive addition of new entries and the dropping out of entries once the history has reached its maximum of 15. The whole point of the kill, copy, and yank commands is to enable you to delete, move and copy any block of text by first moving it to the kill history with a kill command, and then recovering it, if wanted, to the same location or a new one with a YANK or YANK-POP command.

The idea behind the yank commands is that you use YANK to recover the top entry from the kill history and a series of YANK-POP commands to recover lower-down entries.

- If the preceding command was neither YANK nor YANK-POP, then YANK-POP has the same effect as YANK: it copies the top entry in the kill history.
- If the preceding command was YANK, then YANK-POP copies the second entry and makes it the top entry.

- If the preceding command was YANK-POP, then YANK-POP copies the next-lower entry and makes it the top entry.

In other words, YANK-POP copies successively lower entries each time it is used in succession, going *down* and around the kill ring from the most recent entry to earliest, and back again.

Transposing Commands

One needs to use kill and copy commands to move text from one place in the buffer to another. However, GMACS also includes commands that will transpose characters, words, lines and regions. These commands do not use or affect the kill ring. They include:

- TRANSPOSE-CHARACTERS to transpose the characters surrounding the point
- TRANSPOSE-WORDS to transpose the words surrounding the point
- TRANSPOSE-LINES to transpose the current line and the preceding line
- TRANSPOSE-REGIONS to transpose two regions (described in detail above)

Search Operations

You often need to locate a particular character string within a text. You may want to delete it, replace it with another string, or perform some other editing function at that location. You may want to do this at only one instance of the string, at every instance, or at selected instances. GMACS includes several search and replace commands that facilitate these kinds of operations. If you wish to perform search and replace operations on several files at once, you should look at the section on how to create and use tag tables.

The search and replace commands are not case-sensitive to the search string you specify: a search for "LISP" will also find "Lisp" and "lisp". Also, each of the commands automatically re-positions the edit window as necessary to show the located string.

FORWARD-SEARCH and REVERSE-SEARCH prompt for a character string and reposition the point at the first occurrence of the string found in the specified direction. Searches do not begin until after the <Enter> key is typed. Typing a Ctrl-S will move the point to the next occurrence of the search string. Ctrl-R moves the point to the next occurrence of the search string during a reverse search. You can switch back and forth between forward and backward incremental searching using Ctrl-S and Ctrl-R.

Typing any GMACS command that does not directly pertain to searches will terminate the search, leave the point where the search left it, and execute the specified command.

Like EMACS, GMACS includes forward and backward search commands that are *incremental*. This means that the search begins before you have finished typing the search string. Let's say you want to do an incremental search for the word "bar."

- First, you invoke the FORWARD-ISEARCH command, which prompts you for the search string with "Forward Search:" in the message line.
- As soon as you type "b," GMACS places the point after the first location of "b" between the point and end of the buffer.
- After you type the "a," GMACS places the point after the first location of "ba."
- After you type the final "r," it places the point after the first location of "bar."

Let's assume that you wish to terminate the search. You do this by typing Ctrl-G, which will place the point where it was located before the search began. Now let's assume there is no occurrence of "bar." In this case, the editor beeps and displays the message "Failing Forward Search: bar." At this point, you can either:

- Type Ctrl-G, which will shorten the string to "ba," the longest string where an occurrence was found, and place the point back to this occurrence.
- Type Ctrl-G twice, which will terminate the search and place the point where it was located before the search command was issued.

If you make a typing mistake while you are typing in the search string, you can use the <Rubout> key to erase any mistakes; note that as you delete characters from the search string, the point will move back to the first occurrence of the shortened string.

Search and Replace Commands

There are two commands for replacing one text string with another: REPLACE-STRING prompts for two strings and replaces all occurrences of the first string with the second string from the point to the end of the buffer. QUERY-REPLACE prompts for two character strings and replaces selected instances of the first character string with the second string.

When QUERY-REPLACE finds an instance of the string, it halts and prompts you with seven options. Your choices are:

- Y** replace that instance and continue searching
- N** leave that instance unchanged and continue searching
- S** stop the search, leaving the cursor at the current location
- !** replace all remaining instances to the end of the buffer without further prompting
- C** change the replacement string
- Ctrl-G** abort the command--no more searching or replacing
- Ctrl-C** enter a recursive edit to perform additional editing commands; exit the recursive edit and continue the QUERY-REPLACE operations by typing Ctrl-C
Ctrl-C

When QUERY-REPLACE has searched to the end of the buffer (whether it finds instances along the way or not), the cursor is returned to its original position. This also happens if you abort QUERY-REPLACE.

Window Commands

GMACS provides several commands that enable the user to easily scroll through the text in the buffer by lines, screens, or pages, controlling both what is displayed in the edit window and the position of the point in the window.

- **SCROLL-SCREEN-DOWN** and **SCROLL-SCREEN-UP** move the window forward and backward in the edit buffer by about one window-length. The window is positioned on the edit buffer so that the previous second-to-last line in the window becomes the new first line.
- **RECENTER-POINT** moves the point to the beginning of the first line in the edit window. If a positive numeric argument is used, it will position the point the specified number of lines from the top of the window. If a negative numeric argument is used, it will position the point the specified number of lines from the bottom of the window.
- **RECENTER-WINDOW** scrolls the buffer so that the line containing the point is at the center of the window. If a positive argument is used, it will redisplay the screen such that the point will be left the specified number of lines from the top of the page. If a negative argument is used, it will leave the point the specified number of lines from the bottom of the page. An argument of 0 or repeating the command twice (**Ctrl-L Ctrl-L**) will redisplay the screen.
- **SCROLL-DOWN** and **SCROLL-UP** scrolls the screen down or up one line.
- **WINDOW-BACKWARD-PAGE** and **WINDOW-FORWARD-PAGE** scrolls the screen one page backward or forward, positioning the start of the page at the top of the edit window.

Editing in Two Windows

You can split the edit-window area on the screen into two edit windows using the **TWO-WINDOWS** command. The upper window shows the current buffer and the lower window shows the previous

buffer. All the editing commands apply to only one window at a time.

Each window has an edit buffer associated with it. The two buffers may be the same buffer or they may be different buffers, enabling you to edit two different files. At any particular time, the cursor will be in one of the windows, called the *current window*. Any input that you type applies to the current window and the current point.

Having established two windows, you can:

- Scroll the noncurrent window without switching the point to that window with `SCROLL-OTHER-WINDOW`
- Work in the other window with `OTHER-WINDOW`, which makes the other window the current window
- Expand the current window to be the entire screen with `ONE-WINDOW`
- Use `MOVE-SCREEN-OTHER-WINDOW` to scroll the other window down by approximately one screen

GMACS maintains any needed information about the inactive window so that when you return there, you can pick up where you left off. In particular, the point is maintained. There is also a mark pdl for each buffer; thus, there are two mark pdl's unless the two windows have the same buffer. However, GMACS maintains only one kill history, which is accessible in both windows. This feature is one of the main reasons for editing in two windows: it enables you to merge text between two buffers with minimum effort.

Chapter 14

Editing LISP Code

This chapter describes the GMACS commands that are designed specifically for editing LISP code. If you are editing LISP code, you should make sure that the major mode associated with the buffer is "Lisp" or "Common Lisp." This will insure that the special features will work as described here.

Several of the commands refer to such positions in LISP code as "the end of the current list" and "the beginning of the current s-expression." For this to make sense, it's necessary to know what the "current" item means for an s-expression, a function definition or a list.

- The *current item* is the lowest-level item of that kind containing the point.
- The "next" item is the first item of that kind encountered, in one search direction or the other (the search direction is always specified).

The beginning and end of an item need to be defined also. Beginning and end are marked in LISP code by delimiting characters; for the items of interest, these are as follows:

- **For an atom:** Parentheses or white space (the space, tab, or newline character)
- **For a list:** Parentheses

If a command specifies an action on a current, a previous or a next item, and there is no such item in the edit buffer, then GMACS rings the bell and does not move the point. In other words, the command has no effect in that instance except to ring the bell.

Cursor Motion

These commands move the cursor to various parts of LISP code.

- **BACKWARD-SEXP** moves the point to the beginning of the s-expression to its left. If the preceding character is `)`, the point is moved to just left of the matching `(`. If the preceding character is white space, the point is moved to just left of the first character of the preceding s-expression. If the preceding character is `(`, the point moves to the left of it.
- **FORWARD-SEXP** moves the point to the end of the s-expression to its right. If the next character is `(`, the point is moved to just right of the matching `)`. If the next character is white space, the point is moved to just right of the last character of the next s-expression. If the next character is `)`, the point moves to the right of it.
- **BACKWARD-LIST** moves the point to the beginning of the list to its left. The command searches for an open parenthesis and positions the point just to the left of it.
- **FORWARD-LIST** moves the point to the end of the list to its right. The command searches for a close parenthesis and positions the point just after it.
- **DOWN-LIST** moves the point forward in the edit buffer until it is just to the right of the next open parenthesis. **DOWN-LIST** is a forward move. There is no "backward-down-list" command. Given this:

```
+ a (+ b (+ c d))
```

we get this after **DOWN-LIST**:

```
(+ a (+ b (+ c d)))
```

- **BACKWARD-UP-LIST** searches backward for an unmatched open parenthesis and positions the point to the left of the first one encountered.

- **FORWARD-UP-LIST** moves the point forward to the next highest level of a nested list structure. Searches forward for an unmatched close parenthesis and positions the point to the right of the first one encountered. If the point is not currently within a list, then the terminal beeps and the point is not moved. Before a **FORWARD-UP-LIST** command, we might have:

```
(+ a (+ b (+ c d)))
```

After **FORWARD-UP-LIST**, we have:

```
(+ a (+ b (+ c d)))_
```

- **BEGINNING-OF-DEFINITION** and **END-OF-DEFINITION** enable you to move the point to the beginning or to the end of the current function definition. (It's assumed that a function definition (and any other form which is not nested within another form) always begins in column 1 of a line.) The former moves the point backward to the beginning of the current LISP function; it looks backward for the first line that has an open parenthesis in its first column. The latter moves the point forward to the end of the current LISP function.

Convenience Aids to Writing in LISP

Several miscellaneous GMACS features aid you in writing LISP programs:

- **MAKE-MATCHING-():** This command inserts matching parentheses around the point.
- **FIND-UNBALANCED-PARENS:** This command searches for an unbalanced parenthesis and leaves the cursor at the bottom of the function that precedes the function containing the unbalanced parenthesis. If no unbalanced parenthesis is discovered, it displays a message to this effect. If given no argument, it scans through the entire buffer; if given an argument, it scans from the point to the end of the buffer.

- **Paren-flash feature:** Whenever the point is just to the right of a close parenthesis, the corresponding open parenthesis blinks on the screen (if it appears in the window). This feature is enabled automatically in GMACS. To disable it, give the GCLISP command (`setf *flash-mode* nil`) after starting up GMACS.

(The paren-flash feature does not function properly within comments delimited by the GCLISP comment-delimiter pair `#|` and `|#`. If your code includes such comments, and you want to check the balance of parentheses within them, the simplest way to use paren-flash to check the balance is to temporarily remove the comment markers during this operation.)

- **Paren-beep feature:** Whenever a close parenthesis is typed, your terminal will beep, and the message **No matching open parenthesis** will be printed, if there is no matching open parenthesis anywhere in the buffer. (The matching open parenthesis need not be visible in the window.) This feature is normally disabled. To enable it, give the GCLISP command (`setf *overbalance-warn* T`) after starting up GMACS. In addition, the paren-flash feature must be enabled.

Indenting LISP Expressions

These commands enable you to indent a line of LISP code to reflect the nesting level of the current form.

- **INDENT-FOR-COMMENT** indents an appropriate amount for comments. If the current line has no comment, moves the point out to the comment column (inserting spaces as necessary) and inserts a semi-colon. If the line already has a comment, the comment is indented the correct number of spaces and the point is positioned to the right of the semi-colon.
- **INDENT-LISP-LINE** indents the current LISP line to the appropriate level.
- **INDENT-SEXP** corrects the indentation of the s-expression to the right of the point.
- **NEWLINE-INDENT** and **OPEN-INDENTED-LINE** are discussed above in the chapter on text editing.

Displaying Information About LISP Code

Several commands enable you to display on-line documentation about LISP functions. The documentation comes from the text which would be displayed in response to the GMACS help command ED-DOC.

- **DISPLAY-APROPOS** prompts the user for a string and displays a short description of all the symbols whose print name contains the string as a substring.
- **DISPLAY-DOCUMENTATION** displays the full Help documentation for a specified function. It prompts the user for a function name, choosing as the default function the first element of the current s-expression.
- **DISPLAY-LAMBDA-LIST** prompts the user for a function name and displays the lambda-list of the requested function. If no function name is provided, it uses the first member of the current list as the function.
- **DISPLAY-MACROEXPANSION** displays in a type-out window the macroexpansion of the current s-expression.

Killing and Recovering LISP Code

A number of special commands enable you to kill s-expressions and comments. As described earlier, "killing" text means removing it from the edit buffer and placing it in the kill history. Like any entry in the kill history, it can then be recovered by YANK and YANK-POP commands for insertion, if desired, elsewhere in the buffer or in another buffer.

- **KILL-COMMENT** moves to the kill history any comment on the current line (that is, all of the characters from the first semi-colon through the last character before the newline).
- **KILL-SEXP** moves to the kill history the characters forward from the point through the end of the current s-expression.

Keeping Track of Changes Made to Your Program

The `ADD-CHANGE-LOG` command enables you to keep a record of changes that you make to LISP code. If a file called `change.log` exists in the current directory, it will load this file into a buffer, switch you to that buffer and place the point at a place appropriate to make a new entry; it will create a file called `change.log` if one does not already exist.

Evaluating LISP Code from within the Editor

There are two ways to evaluate LISP code from within the editor:

- Invoke one of the commands that will evaluate an s-expression, function definition or buffer of LISP code and display the results in the type-out window.
- Invoke LISP-interaction mode, which will evaluate a form within the buffer and place the results of the evaluation on the next line of the buffer.

It is important to keep in mind that the type-out method loses the results of the evaluation whereas LISP-interact mode places the results in the buffer.

Here are a list of commands that are used to evaluate expressions.

- `EVAL-AND-EXIT` evaluates the current function definition and exits to the top-level interpreter.
- `EVAL-BUFFER` incrementally evaluates the LISP forms in the current buffer and displays the results in the type-out window.
- `EVAL-DEFINITION` evaluates the current function.
- `EVAL-IN-MINIBUFFER` prompts the user for a form and evaluates the form in the type-out window. The advantage of this command is that the environment in which the user types the form is a two-line mini-buffer that will accept GMACS commands.
- `EVAL-SEXP` evaluates the s-expression to the right of the point. An attempt to do an `EVAL-SEXP` on an s-expression with an infinite loop will result in a stack-group-reset and the user will see a message to this effect.

If you type **Ctrl-Break** during the evaluation (or if the **break** function is part of the code), the evaluation behavior is the same as if you were typing the code form-by-form interactively. Evaluation and printing of results are suspended, a new level of the listener is invoked, and you can then perform debugging operations, which typically involve viewing the current values of variables, tracing the execution stack, and so forth. You continue via **Ctrl-G** (from an error) or **Ctrl-P** (from a break), as always in the listener.

You can use the **SET-BUFFER-PACKAGE** command to set the package associated with the current buffer to be a specific package. Subsequent s-expression or function definition evaluations will be added to the package specified.

LISP-Interaction Mode

LISP-interaction mode is a minor mode that enables the user to evaluate s-expressions within the editor and have the results of the evaluation placed back into the edit buffer. LISP-interaction mode is particularly useful because the results of an evaluation remain in the buffer rather than disappear after they have been displayed, which is the case with the **EVAL-SEXP** command since it displays the results in a type-out window.

LISP-interaction mode acts like the top-level interpreter environment except evaluation doesn't occur until the **LISP-INTERACT** command is executed, at which time the s-expression to the left of the point is evaluated. If you try to execute **LISP-INTERACT** when the LISP-interaction mode is turned off, **GMACS** will prompt you as to whether or not you would like it turned on. Note that when LISP-interaction is turned on, the word "linter" will appear within the square brackets in the mode line.

Compiling from within the Editor

GMACS includes several commands for compiling LISP code within the edit buffer. Note that all of these commands add the compiled definition(s) to the LISP environment as well as display the results of the compilation in the type-out window. Note that the **COMPILE-BUFFER** command does not create a .fas file. The following lists the **GMACS** commands that involve compilation:

Here is a list of commands used in compiling LISP code.

- **COMPILE-AND-EXIT** compiles the current function definition and exits to the top-level interpreter.
- **COMPILE-BUFFER** compiles the forms and definitions in the current buffer and loads them into memory. Essentially, it acts like the top-level **compile-file** function except that it loads the compiled code into memory and does not create a corresponding **.fas** file.
- **COMPILE-DEFINITION** compiles the current function definition and adds the compiled definition to your LISP environment.

Both buffer-edit mode and directory-edit mode also enable the user to evaluate and compile LISP code. See the details in the chapter "Manipulating Buffers and Files."

Chapter 15

Customizing Your GMACS Environment

There are two primary ways of customizing your GMACS environment:

- Changing the default settings for key bindings and global variables
- Using keyboard macros to define new commands that consist of a sequence of keystrokes or commands

The file **comtab.lsp** contains the default command key bindings and the file **params.lsp** contains the default global variable settings. The user can change GMACS default settings by editing these files directly.

Another way to do so is to edit the GMACS initialization file **gmaccini.lsp** in the `\gclisp3\gmacc` directory. This file is loaded after all of the other GMACS files are loaded; therefore, any global variable settings or keybindings specified in this file will override any previous default settings. The following is a description of what some of these default settings do and how to change them; you should also look carefully at the sample initialization file provided.

Change Key Bindings

Key bindings are defined in the file **comtab.lsp**. Technically, you can make changes directly to this file, but the preferred method is to add override commands to your **gmaccini.lsp** file. This is because the **comtab.lsp** file, as part of the distribution package, is likely to change in the future as new features are added to GCLISP.

- To bind commands to simple keychords, the `def-key` function should be used:

```
(def-key #\m-Z 'eval-definition)
```

- To bind a `Ctrl-X` keychord one would use the `def-x-key` function. For example, the following line of LISP code in `comtab.lsp` binds `Ctrl-X Ctrl-F` to the `FIND-FILE` command:

```
(def-x-key #\c-F 'find-file)
```

Associating Modes with Buffers

If the first line of a file is an attribute-list, the major mode associated with the file will be the one specified in the attribute-list. GMACS will recognize Common-LISP as a mode in the attribute-list; however, Common-LISP mode and LISP mode are functionally identical in GMACS. Otherwise, if the file has a ".l" or ".lsp" extension, the major mode will be LISP; if the file has a ".tex", ".mss" or ".txt" extension, the major mode will be text.

The user can add his own defaults; for example, to have GMACS associate text mode with any files with the ".doc" file extension, the user should add the following line of LISP code to his `gmacsini.lsp` file:

```
(pushmode "doc" 'text)
```

If the file does not contain an attribute-list and does not have an extension associated with a particular mode, its mode will be determined by the value of the global variable `*default-major-mode*`. The default value of this variable is "normal." The user may want to reset the value of this variable in the `gmacsini.lsp` file.

Specifying a Trashcan Pathname

The directory-edit mode includes the ability to delete files. If the variable `*dired-trashcan*` is `nil`, a file chosen for deletion will be deleted permanently. However, if `*dired-trashcan*` is a string specifying a valid pathname (including a trailing "\"), the file will be copied to this directory and deleted from the current directory. An `undelete` will perform the inverse operation. The default value of

dired-trashcan is nil, so you must reset the value of this variable in **gmacsini.lsp** if you want to use this feature.

Specifying a Default Comment and Fill Column

The values of the global variables ***fill-column*** and ***comment-column***, which are specified in the file **params.lsp**, determine the column where paragraphs will be filled and the column where LISP comments will begin, respectively. The user can change these default values by editing **params.lsp** or by resetting the values in **gmacsini.lsp**.

Eliminating the Pause in the Type-Out Window

By default, a GMACS command that displays text in the type-out window will pause after one screen and prompt the user as to whether or not to continue. In some cases, such as during the type-out window display of the results of a **COMPILE-BUFFER** command, for example, the user may want the display to be continuous. This can be accomplished by changing the value of ***type-out-pause*** to nil in **params.lsp** or resetting the value in **gmacsini.lsp**.

Changing the Modeline Display

By default, the **time**, **date** and GMACS version number will be displayed in the **mode line** at the bottom of the edit screen. The user can eliminate any of this information from the mode line by changing to nil the value of one or more of the global variables ***modeline-time***, ***modeline-date*** or ***modeline-version*** in the file **params.lsp**. Similarly, the user can eliminate the "Alt-H = HELP" string from the mode line by changing the value of the variable ***show-help-string*** to nil.

Limiting the Number of Buffers in Memory

The variable ***max-resident-buffers*** in the file **params.lsp** enables the user to limit the maximum number of buffers resident in memory at any one time. When the number of buffers is about to exceed this value, GMACS unloads the least recently used (LRU) buffer from

memory. When the value of this variable is `nil`, as it is by default, there is no limit on the number of buffers in memory.

Auto-Save

Having GMACS automatically save the buffer after every hundred characters are typed into it can be accomplished with the auto-save feature, which can be implemented by adding this line to your `gmacs.ini` file:

```
"(setf *auto-save-hook* 'auto-save-hook-fn)"
```

The buffer will be saved on disk with a filename identical to the one associated with the buffer but with a `.sav` filetype extension.

To change the number of characters that must be typed before a save is done, change the value of `*auto-save-limit*` in your `gmacs.ini` file.

Keyboard Macros

A keyboard macro is a user-defined command that consists of a sequence of keystrokes. If you need to perform a particular sequence of commands several times, you can define a single keyboard macro that will perform the entire sequence each time it is invoked.

The `DEFINE-KEYBOARD-MACRO` command can be used to start a keyboard macro definition. Any commands and keystrokes typed subsequent to this command will be recorded as part of the macro. The command `END-DEFINE-KEYBOARD-MACRO` is used to terminate the definition. If any of the keystrokes typed during the macro definition causes an editor error, the macro definition is terminated.

The most recently defined keyboard macro is the current keyboard macro; it is the macro that will be named with the `NAME-KEYBOARD-MACRO` command, and the macro that will be executed when the `EXECUTE-KEYBOARD-MACRO` command is called with no arguments. When given a positive numeric argument, the `EXECUTE-KEYBOARD-MACRO` command will execute the current keyboard macro the specified number of times.

The `NAME-KEYBOARD-MACRO` command will prompt the user for a name and assign this name to the current keyboard macro. A named macro can be executed by calling the `EXECUTE-KEYBOARD-MACRO` command with a negative numeric argument.

The user will be prompted for the name of a macro and that macro will be executed the specified number of times.

Once you have named a macro, you can bind it to a keychord. The `BIND-KEYBOARD-MACRO` command will prompt you for a named keyboard macro, and then prompt you for the keychord that you wish to bind to the macro. This keychord can be any single keychord except those that are prefixed with `Ctrl-X` or `F2`. A command can be bound to several keychords at one time. However, a keychord can be bound to only one command at one time. If you use the `BIND-KEYBOARD-MACRO` command to rebind a keychord that is currently bound to another GMACS or user-defined command, the name of the command to which the keychord was previously bound will be displayed on the mode-line.

The `UNBIND-KEYBOARD-MACRO` command can be used to unbind a keychord from its current command binding. For each keychord, GMACS maintains a complete stack of all the commands to which a keychord has been bound. When the `UNBIND-KEYBOARD-MACRO` command is used to unbind a keychord, the keychord is automatically rebound to the command to which it had been most recently bound - that is, the command at the top of the stack. An `UNBIND-KEYBOARD-MACRO` command removes a given command from a keychord's stack history. You can restore the standard GMACS keychord bindings at any time by loading the file `C:\gclisp3\gmacs\comtab.lsp` into your environment.

The `SAVE-KEYBOARD-MACROS` command can be used to save the currently defined keyboard macros in a file for future use. Note that this command will save the current keychord bindings, but it will not save the binding stack for the keychords. The `LOAD-KEYBOARD-MACROS` command will prompt you for a file name and load the macros saved in the file into your GMACS environment.

Chapter 16

Summary Key and Command Reference

This chapter lists commands by function key and by command type.

Function-Key Binding Table

F1	EXIT-EDITOR
F2	HELP
F3	SELECT-BUFFER
F4	SELECT-PREVIOUS-BUFFER
F5	Ctrl
F6	Esc
F7	FIND-FILE
F8	READ-FILE
F9	SAVE-FILE
F10	ED-BEEP

Table of Keypad Keys

Left Arrow	BACKWARD-CHAR
Right Arrow	FORWARD-CHAR
Up Arrow	PREVIOUS-LINE
Down Arrow	NEXT-LINE
Ctrl-Left Arrow	BACKWARD-WORD
Ctrl-Right Arrow	FORWARD-WORD
Pg Up	SCROLL-SCREEN-UP
Pg Dn	SCROLL-SCREEN-DOWN
Ctrl-Pg Up	BACKWARD-SEXP
Ctrl-Pg Dn	FORWARD-SEXP
Home	BEGINNING-OF-BUFFER
End	END-OF-BUFFER
Ctrl-Home	BEGINNING-OF-DEFINITION
Ctrl-End	END-OF-DEFINITION
Del	DELETE-CHAR
Ins	OPEN-LINE

Cursor Motion Commands

BACKWARD-CHAR	Ctrl-B or Left Arrow Moves the point to the left (back) one character position.
BACKWARD-WORD	Alt-B or Ctrl-Left Arrow Moves the point backward to the beginning of the current word.
BEGINNING-OF-BUFFER	Esc < or Home Positions the point at the beginning of the edit buffer.
BEGINNING-OF-NUMBERED-LINE	Ctrl-A Moves the point to the beginning of the current line. If called with a numeric prefix argument, it will position the point to the beginning of the specified line number.
END-OF-BUFFER	Esc > or End Positions the point after the last character in the edit buffer.
END-OF-NUMBERED-LINE	Ctrl-E Moves the point to the end of the current line. If a numeric prefix argument is used, it will move the point to the end of the line number specified.
FORWARD-CHAR	Ctrl-F or Right Arrow Moves the point one character position to the right (forward).
FORWARD-WORD	Alt-F or Ctrl-Right Arrow Moves the point forward to the end of the current word.
NEXT-LINE	Ctrl-N or Down Arrow Moves the point forward to the same column in the next line.

PREVIOUS-LINE **Ctrl-P** or **Up Arrow**
Moves the point backward to the same column in the preceding line.

Edit Window Commands

MOVE-SCREEN-OTHER-WINDOW **Alt-X MOVE-SCREEN-OTHER-WINDOW**
Scrolls the other window down by approximately one screen.

SCROLL-SCREEN-DOWN **Ctrl-V** or **PgDn**
Moves the window forward in the edit buffer by about one window-length (one edit screen). The window is positioned on the edit buffer so that the previous second-to-last line in the window becomes the new first line.

ONE-WINDOW **Ctrl-X 1**
Returns the editor display to one window by expanding the current window to the size of the terminal display.

OTHER-WINDOW **Ctrl-X O**
Moves the cursor to the other window, which becomes the current window.

SCROLL-SCREEN-UP **Alt-V** or **PgUp**
Moves the window backward in the edit buffer by about one window-length (one edit screen). The window is positioned on the edit buffer so that the previous first line in the window becomes the new second-to-last line.

RECENTER-POINT **Alt-R**
Moves the point to the beginning of the first line in the edit window. If a positive numeric argument is used, it will position the point the specified number of lines from the top of the window. If a negative numeric argument is used, it will position the

point the specified number of lines from the bottom of the window.

RECENTER-WINDOW

Ctrl-L

Scrolls the buffer so that the line containing the point is at the center of the window. If a positive argument is used, it will redisplay the screen such that the point will be left the specified number of lines from the top of the page. If a negative argument is used, it will leave the point the specified number of lines from the bottom of the page. An argument of 0 or a repeating the command twice (**Ctrl-L Ctrl-L**) will redisplay the screen.

SCROLL-DOWN

Alt-X SCROLL-DOWN

Scrolls the screen down one line.

SCROLL-OTHER-WINDOW

Ctrl-Z V

Scrolls the other window forward one screen.

SCROLL-UP

Alt-X SCROLL-UP

Scrolls the screen up by one line.

TWO-WINDOWS

Ctrl-X 2

Splits the edit window display area in two, with the upper window showing the current buffer and the lower window showing the previous buffer. The upper window becomes the current window.

WINDOW-BACKWARD-PAGE

Ctrl-X {

Scrolls the screen one page backward, positioning the start of the page at the top of the edit window.

WINDOW-FORWARD-PAGE

Ctrl-X }

Scrolls the screen one page forward, positioning the start of the new page at the top of the edit window.

Text Commands

AUTO-FILL	Ctrl-X M AUTO-FILL is a minor mode that can be toggled on or off. When auto-fill mode is on, paragraphs are filled. This means that when a non-white-space character is typed past the fill column at the end of the line, the line is broken at the nearest word and a newline is automatically inserted into the buffer at the break. When called with a positive numeric prefix argument, auto-fill is turned on. When called with 0 as the numeric prefix argument, AUTO-FILL is turned off. When called with no arguments, AUTO-FILL acts as a toggle.
BACKWARD-COPY-LINE	Alt-X BACKWARD-COPY-LINE Copies all characters to the left of the point on the current line to the top of the kill history.
BACKWARD-KILL-LINE	Alt-X BACKWARD-KILL-LINE Moves all characters to the left of the point on the current line to the kill history.
BACKWARD-KILL-WORD	Ctrl-Rubout or Esc-Rubout Moves the word to the left of the point to the kill history.
BACKWARD-PAGE	Ctrl-X [Moves the point backward one page. Note that a Ctrl-L at the beginning of a line acts as a page delimiter.
BACKWARD-PARAGRAPH	Esc [Moves the point to the beginning of the preceding paragraph.
BACK-TO-INDENTATION	Alt-M Moves the cursor to the first character in a line that is not a space.

BEGINNING-OF-SENTENCE**Alt-A**

Moves the point to the beginning of the current sentence.

COPY-LINE**Alt-X COPY-LINE**

Copies the characters from the point to the end of the current line to the top of the kill history and repositions the point at the end of the line.

DELETE-BLANK-LINES**Ctrl-X Ctrl-O**

Collapses the number of blank lines above and below the point to just one blank line.

DELETE-CHAR**Ctrl-D or Del**

Deletes the character to the right of the point.

DELETE-HORIZONTAL-SPACE**Ctrl-**

Deletes any spaces or tabs adjoining the point on either side.

DELETE-INDENTATION**Ctrl-Z ^**

Deletes the newline character and any indentation at the beginning of the current line. This action appends the current line to the preceding line.

FILL-PARAGRAPH**Alt-Q**

Fills the current paragraph. Lines extending past the fill column in the current paragraph are broken at the first space before the fill column and wrapped onto the next line. Any leading whitespace is deleted.

FORWARD-PAGE**Ctrl-X]**

Moves the point forward one page, leaving the point at the top of the page centered in the edit window. Pages are delimited by a Ctrl-L in the first column of a line.

FORWARD-PARAGRAPH**Esc]**

Moves the point forward one paragraph.

INDENT-RIGIDLY	Ctrl-X Tab Rigidly indents the region between the point and the mark one space. If a numeric argument is used, it will indent the region the specified number of spaces.
INDENT-TEXT-LINE	Alt-X INDENT-TEXT-LINE or, in text mode, Ctrl-I Indents the text on the current line to the next tab stop.
INSERT-COMMAND-NAME	Alt-J Prompts for a keychord and inserts the name of the command bound to the keychord into the buffer at the point.
JUST-ONE-SPACE	Ctrl-Z Space or Esc \ Collapses the number of spaces between the text to the right of the point and the text to the left of the point to one space.
KILL-LINE	Ctrl-K Moves all characters to the right of the point on the current line to the kill history, not including the terminating newline character. (If newline is the only character to the right of the point on the current line, it is moved to the kill history.)
KILL-REGION	Ctrl-W Moves the characters between the current mark and the point to the kill history.
KILL-WORD	Alt-D Moves the word to the right of the point to the kill history.
NEWLINE	Enter Inserts a newline character at the point. Any characters to the right of the point move to the new line.

OPEN-LINE	Insert or Ctrl-O Inserts a newline character after the point (unlike Enter, which inserts the newline before the point). The text is moved to the next line, and the cursor remains on the current line.
QUOTED-INSERT	Ctrl-Q Used for inserting as text those characters which otherwise act as editing commands, such as control characters. The character typed after Ctrl-Q is inserted into the buffer.
RUBOUT	Alt-X Rubout Deletes the character to the left of the point.
RUBOUT-HACKING-TABS	Ctrl-H or Rubout Deletes the character to the left of the point. If the preceding character is a tab, it converts the tab to the appropriate number of spaces and deletes one of the spaces.
SET-FILL-COLUMN	Ctrl-X F If called with no arguments, SET-FILL-COLUMN will reset the margin used to fill paragraphs to the column where the cursor is currently positioned. If a numeric argument is given, it will set the fill margin to the specified number.
SET-FILL-PREFIX	Ctrl-X Alt-F Defines a set of characters to be a fill prefix-- that is characters on the current line to the left of the point will appear at the beginning of every new line when fill mode is on.
TAB-TO-TAB-STOP	Tab Moves text to right of the point to the next tab stop.
TEXT-MODE	Alt-X TEXT-MODE Sets the major mode to text. Note that this will be reflected in the mode-line at the bottom of the edit window.

TRANSDPOSE-CHARACTERS**Ctrl-T**

Transposes the characters to the left of the point.

TRANSDPOSE-LINES**Ctrl-X Ctrl-T**

Transposes the current line and the preceding line.

TRANSDPOSE-WORDS**Alt-T**

Transposes the words surrounding the point.

UNTABIFY-REGION**Alt-X UNTABIFY-REGION**

Converts all the tabs in the region between the point and the mark to the appropriate number of space characters.

Buffer and File Commands

BUFFER-EDIT**Ctrl-X Alt-B**

Places you in a special buffer that contains a list of all the buffers. The list includes the name of the buffer, the full pathname of the file associated with the buffer, whether or not the buffer has been modified since the last time it was saved, and whether the buffer is read/write or read only. In this special buffer, you can compile (C), go to (G), kill (K), load into the interpreter (L) or save (S) any of the buffers listed.

The way to do this is to place the cursor on the line corresponding to a given buffer, and then type the letter corresponding to a buffer operation. If you try to load or compile a buffer that has not been saved, it will ask you if you would like to save it; if you don't, it will perform the operation on the corresponding file saved on disk. To obtain a list of all possible buffer operations, type "?"; to exit the buffer-edit buffer, type "E."

BUFFER-READ-ONLY**Alt-X BUFFER-READ-ONLY**

Changes the status of a buffer that is read/write to a buffer that is read-only.

BUFFER-READ-WRITE**Alt-X BUFFER-READ-WRITE**

Changes the status of a buffer that is read-only to a buffer that is read/write.

CHANGE-DIRECTORY**Ctrl-X C**

Prompts for a directory name, and changes the current default directory to the directory with that name.

DIRECTORY-EDIT**Ctrl-X D**

Prompts for a pathname and places you in a special buffer that displays a list of all the files that match in that directory. The list contains the filename, file-size in bytes, creation date and creation time. In this special buffer, you can compile (C), find (F), delete (D), undelete (U), view (V) or load into the interpreter (L) any of the files listed. Typing a "?" will list all the possible file operations and typing an "E" will exit you from the directory edit buffer.

The way to do perform an operation on one of the files is to place the cursor on the line in the buffer corresponding to a given file, and then type the letter corresponding to the desired file operation. If the GMACS variable ***dired-trashcan*** is nil, a delete command will permanently delete the specified file. However, if ***dired-trashcan*** is a string holding the name of a directory (with a trailing \), a delete command will copy the file to the trashcan directory, delete the file from the directory being edited and mark the delete column in the buffer. A subsequent "undelete command" will perform the inverse operations on a file that has been marked for deletion. Note that the ***dired-trashcan*** directory must be periodically cleaned out by the user.

DISPLAY-DIRECTORY	Ctrl-X Ctrl-D Prompts for a pathname and displays in a type-out window a list of all files that match it. You are prompted for the pathname of the directory you want. You can specify either a directory or a filename, or a set of filenames using the "*" wild-card convention, just as in the DOS dir command. The directory listing is displayed in a type-out window.
DUMP-BINARY-FILE	Alt-X DUMP-BINARY-FILE Dumps a file to disk in binary format. Prompts for a filename.
EXECUTE-DOS-COMMAND	Alt-X EXECUTE-DOS-COMMAND Prompts the user for a DOS command, executes it and displays the results in the type-out window.
EXIT-EDITOR	Ctrl-X Ctrl-C or F1 Exits the GMACS environment and returns you to the GCLISP environment from which you entered GMACS.
FIND-FILE	Ctrl-X Ctrl-F or F7 Searches the set of edit-buffer names for a specified filename. Selects the buffer with that filename if there is one. Otherwise, creates a buffer with that name and reads the file into the new buffer from disk. The command prompts you for the filename.
INSERT-BINARY-FILE	Alt-X INSERT-BINARY-FILE Prompts the user for the name of a binary file and inserts the contents of the file at the point. The file is opened in binary mode (unsigned-byte instead of string-char), and newline processing is not done. One result is that newlines are visible as Ctrl-M characters.
INSERT-DOS-OUTPUT	Alt-X INSERT-DOS-OUTPUT Prompts the user for a DOS command and inserts the output from the DOS command at the point.

INSERT-FILE	Ctrl-X I Prompts the user for a file and inserts the contents of the file at the point.
KILL-BUFFER	Ctrl-X K Prompts for the name of a buffer and removes it from the list of buffers known to the editor.
LIST-BUFFERS	Ctrl-X Ctrl-B Lists the names of all existing buffers in a type-out window, together with the name of associated files, if any. Modified buffers are marked with the buffer-status (*).
NORMAL-MODE	Alt-X NORMAL-MODE Sets the major mode associated with a buffer to normal. Normal mode is similar to text mode except that, unlike text mode, normal mode does not automatically turn on auto-fill mode.
PUSH-TO-DOS	Alt-X PUSH-TO-DOS Pushes the user directly to DOS. To return to GMACS, type "exit."
READ-FILE	Ctrl-X Ctrl-R or F8 Reads a specified file into the current buffer, overwriting the existing contents of the buffer. The command prompts for the filename.
SAVE-ALL-FILES	Ctrl-X Ctrl-M Saves the contents of all buffers that have been modified into disk storage under the current file names.
SAVE-FILE	Ctrl-X Ctrl-S or F9 Copies the contents of the current edit buffer into disk storage under the current file name. If a file with that name already exists on disk, the command overwrites the existing file.
SAVE-FILES-EXIT	Alt-X SAVE-FILES-EXIT Saves the contents of all buffers that have been modified into disk storage

under the current file names and exits to the top-level interpreter.

SAVE-FILES-PUSH-TO-DOS**Alt-X SAVE-FILES-PUSH-TO-DOS**

Saves the contents of all buffers that have been modified into disk storage under the current file names and pushes to DOS.

SELECT-BUFFER**Ctrl-X B or F3**

Selects a specified buffer and displays it in the edit window. The command prompts you for the name of the desired buffer. Pressing the ENTER key without entering a buffer name selects the previous buffer. If the buffer does not exist, a new buffer is opened having no current file.

SELECT-PREVIOUS-BUFFER**Ctrl-Z L or F4**

Selects the previous buffer.

SET-BUFFER-PACKAGE**Ctrl-Z : or Ctrl-X P**

Sets the package associated with the current buffer to a specified package. The command prompts for the package name. The command asks you if you would like to create a new package if the one you specified does not already exist.

SET-MODE**Alt-X SET-MODE**

Prompts the user for a major mode and resets the major mode to the one specified.

SET-VARIABLE**Alt-X SET-VARIABLE**

Prompts the user for a GMACS variable name and then for a value for the variable. When stating the variable, you must enclose the variable name in *'s. In addition, unless you are in the GMACS package, you must also state the GMACS package name when typing the variable. Typically, this command is

used to reset one of the global variables specified in the `params.lsp` file.

UNLOAD-BUFFER**Alt-X UNLOAD-BUFFER**

Prompts for a buffer name and unloads the contents of the specified buffer from memory. It first asks the user whether or not to save the contents of the buffer into the corresponding file. In the buffer-edit buffer, an unloaded buffer is displayed with parentheses surrounding the buffer name. If the user selects an unloaded buffer, the corresponding file will be read from disk.

The user can cause the editor to automatically unload buffers by setting the global variable `*max-resident-buffers*` to something other than `nil` in the `gmacsini.lsp` file. If a command causes the number of buffers to exceed the value of `*max-resident-buffers*`, the editor will automatically unload a buffer.

UNMODIFY-BUFFER**Esc ~**

Marks the current buffer as unmodified. Clears the buffer-status (*) in the mode line.

VIEW-FILE**Ctrl-X V**

Prompts the user for a file name and displays the contents of the file in the type-out window.

WRITE-FILE**Ctrl-X Ctrl-W**

Writes out the contents of the current buffer to the specified file. The command prompts you for the filename.

Search and Replace Commands

FORWARD-ISEARCH**Ctrl-S**

Prompts for a character string and performs an incremental forward search

for the specified string. All inserting characters are inserted into the search string. Typing a GMACS command keychord will stop the search, set a mark at the point and execute the command specified. **Esc** terminates the search. **Ctrl-S** find the next occurrence of the search string, and **Ctrl-R** does a reverse search on the current string. The **Rubout** key will delete characters from the current search string.

FORWARD-SEARCH**Alt-X FORWARD-SEARCH**

Searches forward from the point for a specified character string. The point moves to the end of the first instance found. The command prompts you for the string.

QUERY-REPLACE**Esc %**

Prompts the user for two character strings and replaces selected instances of the first string with the second string from the point to the end of the buffer. At each occurrence, you are queried as to whether or not to do the replace. Typing an "!" will replace all of the remaining occurrences.

REPLACE-STRING**F5 %**

Replaces all instances of a specified string with another string, from the point to the end of the buffer. The command prompts for both strings.

REVERSE-ISEARCH**Ctrl-R**

Prompts for a character string and performs an incremental backward search for the string.

REVERSE-SEARCH**Alt-X REVERSE-SEARCH**

Prompts for a character string and performs an incremental backward search for the specified string. All inserting characters are inserted into the search string. Typing a GMACS command keychord will stop the search, set a mark at the point and execute the command specified. **Esc** terminates the

search. **Ctrl-R** finds the next occurrence of the search string, and **Ctrl-S** does a forward search on the current string. The **Rubout** key deletes characters from the current search string.

Case-Setting Commands

BACKWARD-LOWERCASE-WORD

Alt-X BACKWARD-LOWERCASE-WORD

Searches for the first word to the left of the point and puts the entire word in lowercase.

BACKWARD-UPPERCASE-INITIAL

Alt-X BACKWARD-UPPERCASE-INITIAL

Puts the initial character of the first word to the left of the point in uppercase.

BACKWARD-UPPERCASE-WORD

Alt-X BACKWARD-UPPERCASE-WORD

Searches for the first word to the left of the point and puts the entire word in uppercase.

LOWERCASE-REGION

Ctrl-X Ctrl-L

Puts all the letters in the region between the mark and the point in lowercase.

LOWERCASE-WORD

Alt-L

Puts all the letters of a word that are located to the right of the point in lowercase.

UPPERCASE-INITIAL

Alt-C

Capitalizes the first letter of the word to the right of the point and puts the rest of the letters of that word in lowercase.

UPPERCASE-REGION	Ctrl-X Ctrl-U Puts all the letters in the region in uppercase.
UPPERCASE-WORD	Alt-U Puts the letters of the word to the right of the point in uppercase.

Commands for Editing and Debugging LISP

ADD-CHANGE-LOG-ENTRY	Alt-X ADD-CHANGE-LOG-ENTRY Switches you to a buffer containing the file <code>change.log</code> , placing the point in a location convenient for making a new entry.
AUTO-FILL-COMMENTS	Alt-X AUTO-FILL-COMMENTS This is a minor mode that can be toggled on or off. When on, comment blocks will be filled. Note that comment blocks begin with a <code># </code> in the first column of the first line and end with a <code> #</code> in the first column of the last line. If <code>AUTO-FILL-COMMENTS</code> is given a positive numeric argument, it will be turned on, a 0 argument, then off, no arguments, then it acts like a toggle.
BACKWARD-KILL-SEXP	Ctrl-Z Rubout Moves to the kill history the characters backward from the point to the beginning of the current s-expression.
BACKWARD-LIST	Ctrl-Z P Moves the point to the beginning of the list to its left. The command searches for an open parenthesis and positions the point just to the left of it.
BACKWARD-MARK-SEXP	Alt-X BACKWARD-MARK-SEXP Puts a mark at the beginning of the s-expression to the left of the point. It does not change the position of the cursor.

- BACKWARD-SEXP** **Ctrl-Z B** or **Ctrl-PgUp**
Moves the point to the beginning of the s-expression to its left.
- BACKWARD-UP-LIST** **Ctrl-Z U** or **Ctrl-Z (**
Searches backward for an unmatched open parenthesis and positions the point to the left of the first one encountered.
- BEGINNING-OF-DEFINITION** **Ctrl-Z A** or **Ctrl-Home**
Moves the point backward to the beginning of the current LISP function. Looks backward for the first line that has an open parenthesis in its first column.
- COMPILE-AND-EXIT** **Alt-X COMPILE-AND-EXIT**
Compiles the current function definition and exits to the top-level interpreter.
- COMPILE-BUFFER** **Alt-X COMPILE-BUFFER**
Compiles the forms and definitions in the current buffer and loads them into memory. Essentially, it acts like the Top-level **compile-file** function except that it loads the compiled code into memory and does not create a corresponding **.fas** file.
- COMPILE-DEFINITION** **Ctrl-X Alt-E**
Compiles the current function definition and adds the compiled definition to your LISP environment.
- DISPLAY-LAMBDA-LIST** **Ctrl-Z L**
Prompts the user for a function name and displays in the type-out window the lambda-list of the requested function. If no function name is provided, it uses the first member of the current list as the function.
- DISPLAY-MACROEXPANSION** **Esc-@** or **Ctrl-Z M**
Displays in a type-out window the macroexpansion of the current s-expression.

DOWN-LIST	Ctrl-Z D Moves the point forward in the edit buffer until it is just to the right of the next open parenthesis.
END-OF-DEFINITION	Ctrl-Z E or Ctrl-Z or Ctrl-End Moves the point forward to the end of the current LISP function.
EVAL-AND-EXIT	Alt-X EVAL-AND-EXIT Evaluates the current function definition and exits to the top-level interpreter.
EVAL-BUFFER	Alt-X EVAL-BUFFER Incrementally evaluates the LISP forms in the current buffer and displays the results in the type-out window.
EVAL-DEFINITION	Ctrl-X Ctrl-E Evaluates the current function.
EVAL-IN-MINIBUFFER	Esc Esc or F6 Esc Prompts the user for a form and evaluates the form in the type-out window. The advantage of this command is that the environment in which the user types the form is a two-line mini-buffer that will accept GMACS commands.
EVAL-SEXP	Esc ! Evaluates the s-expression to the right of the point.
FIND-UNBALANCED-PARENS	Alt-X FIND-UNBALANCED-PARENS Scans the buffer for an unbalanced parenthesis and leaves the cursor at the bottom of the function preceding the function that contains the unbalanced parenthesis. If given no argument, it scans through the entire buffer; if given an argument, it scans from the point on.
FORWARD-LIST	Ctrl-Z N Moves the point to the end of the list to its right. The command searches for a

close parenthesis and positions the point just after it.

FORWARD-SEXP

Ctrl-Z F or **Ctrl-PgDn**

Moves the point to the end of the s-expression to its right.

FORWARD-UP-LIST

Ctrl-Z)

Moves the point forward to the next highest level of a nested list structure. Searches forward for an unmatched close parenthesis and positions the point to the right of the first one encountered.

LISP-INTERACT

Esc Enter

Evaluates the s-expression to the left of the point and places the results on the next line of the edit buffer. It gives you the option of going into LISP-INTERACTION-MODE if you are not already in it.

LISP-INTERACTION

Alt-X LISP-INTERACTION

LISP-INTERACTION is a minor mode that can be toggled on or off. When you are in it, the LISP-INTERACT command can be used to evaluate the s-expression to the left of the point and have the results of the evaluation placed on the next line in the edit buffer. As with other minor modes, it is turned on when called with a positive numeric prefix argument, turned off with 0, and with no arguments, it acts like a toggle.

LISP-MODE

Alt-X LISP-MODE

Sets the major mode to LISP. Note that this will be reflected in the mode-line at the bottom of the edit buffer.

INDENT-FOR-COMMENT

Esc ;

If the current line has no comment, moves the point out to the comment column (inserting spaces as necessary) and inserts a semi-colon. If the line already has a comment, the comment is indented the correct number of spaces

	and the point is positioned to the right of the semi-colon.
INDENT-LISP-LINE	Ctrl-I Indents the current LISP line to the appropriate level.
INDENT-SEXP	Ctrl-Z I or Ctrl-Z Q Corrects the indentation of the s-expression to the right of the point.
KILL-COMMENT	Ctrl-Z ; Moves to the kill history any comment on the current line (that is, all of the characters from the first semi-colon through the last character before the newline).
KILL-SEXP	Ctrl-Z K Moves to the kill history the characters forward from the point through the end of the current s-expression.
MAKE-MATCHING-()	Esc (Inserts matching parentheses around the point.
NEWLINE-INDENT	Ctrl-J or Ctrl-Enter Inserts a newline character at the current point, moves the point to the new line, and inserts white space to correctly indent the new line. The point is placed to the right of the indentation.
NEXT-COMMENT-LINE	Alt-N If the next line has no comment, moves the point out to the comment column (inserting spaces as necessary) and inserts a semi-colon. If the next line already has a comment, the comment is indented the correct number of spaces and the point is positioned to the right of the semi-colon.
OPEN-INDENTED-LINE	Ctrl-Z O Inserts a newline character after the point (unlike ENTER, which inserts the newline before the point). The text is moved to the next line and indented,

and the cursor remains on the current line.

PARSE-ATTRIBUTE-LIST **Alt-X PARSE-ATTRIBUTE-LIST**
Changes the major mode and the package settings for the buffer back to those specified in the attribute-list; these changes will be reflected in the mode line.

PREVIOUS-COMMENT-LINE **Alt-P**
If the previous line has no comment, moves the point out to the comment column (inserting spaces as necessary) and inserts a semi-colon. If the previous line already has a comment, the comment is indented the correct number of spaces and the point is positioned to the right of the semi-colon.

RECENTER-DEFINITION **Ctrl-Z R**
Positions the beginning of the current function, or its leading comments if it has any, at the top of the screen.

REMOVE-SURROUNDING-() **Ctrl-Z \ or Ctrl-Z |**
Removes the closest level of parentheses surrounding the point.

SET-BUFFER-PACKAGE **Ctrl-Z :** or **Ctrl-X P**
Sets the package associated with the current buffer to a specified package. The command prompts for the package name. The command asks you if you would like to create a new package if the one you specified does not already exist.

SET-COMMENT-COLUMN **Ctrl-X ;**
If called with no numeric arguments, sets the comment column to the column on which the point is currently positioned. If called with a numeric prefix argument, it sets the comment column to the number specified.

TRANSPOSE-SEXPS**Ctrl-Z T**

Transposes the s-expressions to the left and right of the point, leaving the cursor at the end of the new right-most s-expression.

UPDATE-ATTRIBUTE-LIST**Alt-X UPDATE-ATTRIBUTE-LIST**

Updates the mode and package settings in the attribute-list using the current settings for the buffer, i.e., the settings specified in the mode line.

UPDATE-MODE-ATTRIBUTE**Alt-X UPDATE-MODE-ATTRIBUTE**

Updates the mode setting in the attribute-list using the current major mode setting for the buffer.

UPDATE-PACKAGE-ATTRIBUTE**Alt-X UPDATE-PACKAGE-ATTRIBUTE**

Updates the package setting in the attribute-list using the current package setting for the buffer.

Region and Kill History Commands

APPEND-NEXT-KILL**Ctrl-Z W**

Causes the next kill command to append the killed text to the entry at the top of the kill history.

COPY-REGION**Alt-W**

Copies the region between the point and the mark to the top of the kill history.

COPY-TO-REGISTER**Ctrl-X X**

Prompts the user for a register name and associates the text between the point and the mark with this name. It also copies the text to the kill history. If preceded by a Ctrl-U, it also deletes the text from the current buffer.

- DISPLAY-KILL-HISTORY** **Ctrl-Z Y**
Displays in a type-out window all entries contained in the kill history. Note that an arrow points to the top of the kill ring. The next highest entry in the kill ring is the one that precedes it.
- EXCHANGE-POINT-AND-MARK** **Ctrl-X Ctrl-X**
Exchanges the point and the current mark.
- INSERT-REGISTER** **Ctrl-X G**
Prompts the user for a register name and inserts the text associated with this name into the buffer at the point.
- KILL-REGION** **Ctrl-W**
Moves the text between the current mark and the point to the top of the kill history.
- SET-POP-MARK** **Ctrl-@ or F5 Space**
Puts a mark where the point is and puts it at the top of the mark pdl (making it the current mark). Prefixed with **Ctrl-U**, the command positions the point at the current mark and pops that mark from the pdl. Prefixed with **Ctrl-U Ctrl-U**, the command just pops the current mark from the mark pdl.
- SHOW-REGISTERS** Lists all the registers. For each register, it lists the name, and the first line of text followed by a series of dots, where each dot represents a line of text.
- TRANSDPOSE-REGIONS** **Ctrl-X T**
Uses the top three marks on the mark stack and the point as markers. Sorts these markers in the order of their location in the file and transposes the region between the first and second marker with the region between the third and fourth marker. If there are only two marks on the mark stack, it uses these markers along with the point to define two regions separated by the middle of the three markers and

transposes the two regions around the middle marker.

YANK

Ctrl-Y

Inserts the entry at the top of the kill history into the current buffer at the point.

YANK-POP

Alt-Y

If the last command was YANK or YANK-POP, the text returned to the buffer by the last command is replaced in the buffer by the next lower entry in the kill history. Otherwise the command has the same effect as YANK.

Documentation/Help Commands

(Break to listener)

Ctrl-Break

DISPLAY-APROPOS

Alt-H Alt-A or F2 Alt-A

Prompts the user for a string and displays in the type-out window a short description of all the symbols whose print name contains the string as a substring. Equivalent to the Top-level **apropos** function.

DISPLAY-DOCUMENTATION

Alt-H Alt-D or F2 Alt-D

Displays in a type-out window the full Help documentation for a specified function. It prompts the user for a function name, choosing as the default function the first element of the current s-expression. Equivalent to the Top-level **documentation** function.

DO-IT-AGAIN

Ctrl-X .

Repeats the previous GMACS command.

ED-APROPOS

Alt-H A or F2 A

Prompts you for a character string, and displays in a type-out window every GMACS command which contains the specified string in its name.

ED-BEEP	Ctrl-G or Ctrl-X Ctrl-G Aborts the current command, rings the terminal bell, and returns you to normal GMACS command entry.
ED-DOC	Alt-H D or F2 D Prompts you for a character string and displays in a type-out window the on-line documentation for every GMACS command that contains the specified string in its name.
ED-HELP	Alt-H ? or Alt-H H or F2 ? or F2 H Displays a help menu that describes how to access on-line documentation and help facilities from within GMACS.
ED-KEYCHORD	Alt-H K or F2 K or Alt-H C or F2 C or Esc ? Prompts you for a keychord, and displays in a type-out window the command bound to the specified keychord.
ED-LISTBACK	Alt-H L Displays the last 50 keystrokes and their command bindings.
EXTENDED-COMMAND	Alt-X Any GMACS command, including those GMACS commands not bound to a keychord or key sequence, can be invoked by entering Alt-X and typing the name of the command.
NUMERIC-ARG-PREFIX	Ctrl-U Used as a command prefix to establish a repeat count for the command (valid for most commands). Prefixed by Ctrl-U , a command executes 4 times (the default repeat count is 4). Prefixed by Ctrl-U <n> , a command executes <n> times. If <n> is negative and there is a meaningful "opposite" version of the command, that is executed positive- <n> times. (For example, the command to move the cursor down by -4 lines will move the cursor up by 4 lines.) Repetitions of Ctrl-U following the

numeric argument <n>, if any, multiply the repeat count by 4 each time.

SHOW-POSITION

Esc = or Ctrl-X =

Displays information on the current position of the point. Includes the current line number and the total number of lines in the file, the index of the point in the line, the column that represents the end of the line, the ASCII code for the current character and the syntactic category of the current character.

SHOW-VERSION

Alt-X SHOW-VERSION

Displays the GMACS version number with its corresponding date.

Marking Commands

EXCHANGE-POINT-AND-MARK

Ctrl-X Ctrl-X

Exchanges the point and the current mark.

MARK-BEGINNING-OF-BUFFER

F5 <

Puts a mark at the beginning of the buffer.

MARK-END-OF-BUFFER

F5 >

Puts a mark at the end of the buffer.

MARK-PAGE

Ctrl-X Ctrl-P

Puts a mark at the top of the page and positions the point to the right of the mark.

MARK-SEXP

Ctrl-Z @ or Ctrl-Z ^@

Puts a mark at the end of the s-expression to the left of the point.

MARK-WHOLE-BUFFER

Ctrl-X H

Marks the end of the buffer and places the point at the top of the buffer.

SET-POP-MARK

Ctrl-@ or **F5 Space**

Puts a mark where the point is and puts it at the top of the mark pdl (making it the current mark). Prefixed with **Ctrl-U**, the command positions the point at the current mark and pops that mark from the pdl. Prefixed with **Ctrl-U Ctrl-U**, the command just pops the current mark from the mark pdl.

Tag Table Commands

TAGS-ADD-FILE

Alt-X TAGS-ADD-FILE

Prompts the user for a file name and adds this file to the current tag table. Note that it will not use this file when doing search and replace operations until a corresponding index .lsx file is built.

TAGS-ADD-FILES

Alt-X TAGS-ADD-FILES

Prompts the user for a pathname and adds the files that match the pathname to the current tag table. It accepts "*" as a wildcard character in the pathname. It will continue to prompt the user for more pathnames until the user enters an "!".

TAGS-CONTINUE-MAP

Ctrl-Z .

After a search or replace mapping operation has been interrupted, TAGS-CONTINUE-MAP will continue the operation where it left off, placing the point at the next occurrence of the search string.

TAGS-FIND-ALL

Alt-X TAGS-FIND-ALL

Performs a FIND-FILE on all of the files listed in the current tag table.

TAGS-FIND-DEFINITION

Esc .

Tries to locate a function or variable definition by loading the appropriate source file and moving the cursor to the top of the definition. Given no

argument, TAGS-FIND-DEFINITION tries to find the calling function of the list closest to the point; this will fail if there is no surrounding list. Typing **Ctrl-U TAGS-FIND-DEFINITION** will prevent the search for the surrounding list and allow the user to type in any function name. Typing **Ctrl-U Ctrl-U TAGS-FIND-DEFINITION** will search for the current word rather than the first word of the current list. Note that the entire function must be specified if it is typed in by hand; only exact matches will be found.

TAGS-INDEX-FILE**Alt-X TAGS-INDEX-FILE**

Prompts the user for a file name and creates an index (.lsx) file for the file. The index file is both loaded into memory and stored on disk.

TAGS-LOAD-INDEX**Alt-X TAGS-LOAD-INDEX**

Loads from disk the index files associated with the current tag table.

TAGS-LOAD-TABLE**Alt-X TAGS-LOAD-TABLE**

Prompts for the name of a tag table and loads it from disk.

TAGS-MAKE-INDEX**Alt-X TAGS-MAKE-INDEX**

Creates index (.lsx) files for all of the files in the current tag table. It both loads the index files into memory and stores them on disk.

TAGS-QUERY-REPLACE**Alt-X TAGS-QUERY-REPLACE**

Performs a QUERY-REPLACE operation among all the files listed in the current tag table. If interrupted, the QUERY-REPLACE can be continued using the TAGS-CONTINUE-MAP command.

TAGS-REMOVE-FILE**Alt-X TAGS-REMOVE-FILE**

Removes a file from the current tag table. Note that a TAGS-SAVE-TABLE must be done to make this change permanent.

TAGS-REPLACE-STRING	Alt-X TAGS-REPLACE-STRING Performs a global replace among all of the files in the current tag table. All files that are changed during the mapping operation are immediately saved on disk.
TAGS-SAVE-TABLE	Alt-X TAGS-SAVE-TABLE Saves the current tag table in a disk file.
TAGS-SEARCH	Alt-X TAGS-SEARCH Prompts the user for a string and places the point at the first occurrence of the string it finds. The TAGS-CONTINUE-MAP command can be used to look for successive occurrences of the string among all the files in the tag table.
TAGS-SHOW-TABLE	Alt-X TAGS-SHOW-TABLE Displays a list of all the files in the current tag table.
TAGS-USE-TABLE	Alt-X TAGS-USE-TABLE Prompts the user for the name of a tag table and makes the specified table the current tag table. If the table is not already in memory, it attempts to load it from disk; otherwise, it creates an empty tag table of the same name and makes that the current tag table.

Keyboard Macros

BIND-KEYBOARD-MACRO	Alt-X BIND-KEYBOARD-MACRO Binds a named keyboard macro to a keychord. Prompts the user for a named keyboard macro, using the current keyboard macro as a default. Then, it prompts the user to type a keychord. This keychord can be any single keychord, except those prefixed with Ctrl-X or F2 .
----------------------------	---

DEFINE-KEYBOARD-MACRO**Ctrl-X (**

Begins the definition of a keyboard macro. Any keystrokes subsequent to this command will be recorded as part of the macro. The command **END-DEFINE-KEYBOARD-MACRO** is used to terminate the definition. If any of the keystrokes typed during the macro definition causes an editor error, the macro definition is terminated.

END-DEFINE-KEYBOARD-MACRO**Ctrl-X)**

Ends the definition of a keyboard macro.

EXECUTE-KEYBOARD-MACRO**Ctrl-X E**

Executes the current keyboard macro. When given a positive numeric argument, it executes the current keyboard macro the specified number of times. When given a negative argument, it prompts for the name of the macro and executes that macro the specified number of times.

LOAD-KEYBOARD-MACROS**Alt-X LOAD-KEYBOARD-MACROS**

Prompts the user for a file name and loads the macros saved in this file into the GMACS environment. Note that the keychord bindings for all of the macros will be the ones that were current when the macros were saved; however, the binding stack for the keychords was not saved and will not exist when they are reloaded.

NAME-KEYBOARD-MACRO**Alt-X NAME-KEYBOARD-MACRO**

Prompts the user for a name and assigns that name to the current keyboard macro. A macro must be defined before it can be named, and it must be named before it can be bound to a keychord.

SAVE-KEYBOARD-MACROS**Alt-X SAVE-KEYBOARD-MACROS**

Prompts the user for a file name and saves all of the currently defined named macros in this file. Note that it saves the current keychord bindings but does not save the binding stack for each keychord.

UNBIND-KEYBOARD-MACROS**Alt-X UNBIND-KEYBOARD-MACROS**

Prompts the user for a keychord, and if the keychord is currently bound to a command, unbinds it. Note that GMACS maintains a history of all the commands to which a keychord has been bound. When a keychord is unbound from a given command, it reverts to its most recent binding - that is, the command at the top of the stack. When a keychord is unbound from its current command, the command is removed from the stack history.

Chapter 17

Alphabetical Command Listing

A

ADD-CHANGE-LOG-ENTRY	Alt-X <i>command-name</i>
APPEND-NEXT-KILL	Ctrl-Z W
AUTO-FILL	Ctrl-X M
AUTO-FILL-COMMENTS	Alt-X <i>command-name</i>

B

BACK-TO-INDENTATION	Alt-M
BACKWARD-CHAR	Ctrl-B <Left Arrow>
BACKWARD-COPY-LINE	Alt-X <i>command-name</i>
BACKWARD-KILL-LINE	Alt-X <i>command-name</i>
BACKWARD-KILL-SEXP	Ctrl-Z <Rubout>
BACKWARD-KILL-WORD	Ctrl-<Rubout> Esc <Rubout>
BACKWARD-LIST	Ctrl-Z P
BACKWARD-LOWERCASE-WORD	Alt-X <i>command-name</i>

BACKWARD-MARK-SEXP	Alt-X <i>command-name</i>
BACKWARD-PAGE	Ctrl-X [
BACKWARD-PARAGRAPH	Esc [
BACKWARD-SEXP	Ctrl-Z B Ctrl-<Pg Up>
BACKWARD-UP-LIST	Ctrl-Z U Ctrl-Z (
BACKWARD-UPPERCASE-INITIAL	Alt-X <i>command-name</i>
BACKWARD-UPPERCASE-WORD	Alt-X <i>command-name</i>
BACKWARD-WORD	Alt-B Ctrl-<Left Arrow>
BEGINNING-OF-BUFFER	Esc < <Home>
BEGINNING-OF-DEFINITION	Ctrl-Z A Ctrl-<Home>
BEGINNING-OF-NUMBERED-LINE	Ctrl-A
BEGINNING-OF-SENTENCE	Alt-A
BIND-KEYBOARD-MACRO	Alt-X <i>command-name</i>
BUFFER-EDIT	Ctrl-X Alt-B
BUFFER-READ-ONLY	Alt-X <i>command-name</i>
BUFFER-READ-WRITE	Alt-X <i>command-name</i>
C	
CHANGE-DIRECTORY	Ctrl-X C
COMPILE-AND-EXIT	Alt-X <i>command-name</i>
COMPILE-BUFFER	Alt-X <i>command-name</i>
COMPILE-DEFINITION	Ctrl-X Alt-E

COPY-LINE	Alt-X <i>command-name</i>
COPY-REGION	Alt-W
COPY TO REGISTER	Alt-X <i>command-name</i>

D

DEFINE-KEYBOARD-MACRO	Ctrl-X (
DELETE-BLANK-LINES	Ctrl-X Ctrl-O
DELETE-CHAR	Ctrl-D
DELETE-HORIZONTAL-SPACE	Ctrl-\ Esc <Space>
DELETE-INDENTATION	Ctrl-Z ^
DIRECTORY-EDIT	Ctrl-X D
DISPLAY-APROPOS	Alt-H A <F2> A
DISPLAY-DIRECTORY	Ctrl-X Ctrl-D
DISPLAY-DOCUMENTATION	Alt-H D <F2> D
DISPLAY-KILL-HISTORY	Ctrl-Z Y
DISPLAY-LAMBDA-LIST	Ctrl-Z L
DISPLAY-MACROEXPANSION	Esc @ Ctrl-Z M
DO-IT-AGAIN	Ctrl-X .
DOWN-LIST	Ctrl-Z D
DUMP-BINARY-FILE	Alt-X <i>command-name</i>

E

ED-APROPOS	Alt-H A <F2> A
ED-BEEP	Ctrl-G Alt-G Ctrl-X G
ED-DOC	Alt-H D <F2> D
ED-HELP	Alt-H ? Alt-H H <F2> ? <F2> H
ED-KEYCHORD	Alt-H K <F2> K Alt-H C <F2> C Alt-?
ED-LISTBACK	Alt-H L <F2> L
END-DEFINE-KEYBOARD-MACRO	Ctrl-X)
END-OF-BUFFER	Esc > <End>
END-OF-DEFINITION	Ctrl-Z E Ctrl-Z j Ctrl-<End>
END-OF-NUMBERED-LINE	Ctrl-E
END-OF-SENTENCE	Alt-E
EVAL-AND-EXIT	Alt-X <i>command-name</i>
EVAL-BUFFER	Alt-X <i>command-name</i>
EVAL-DEFINITION	Ctrl-X Ctrl-E
EVAL-IN-MINIBUFFER	Esc Esc <F6> Esc
EVAL-SEXP	Esc !
EXCHANGE-POINT-AND-MARK	Ctrl-X Ctrl-X

EXECUTE-DOS-COMMAND	Alt-X <i>command-name</i>
EXECUTE-KEYBOARD-MACRO	Ctrl-X E
EXIT-EDITOR	Ctrl-X Ctrl-C <F1>
EXTENDED-COMMAND	Alt-X

F

FILL-PARAGRAPH	Alt-Q
FIND-FILE	Ctrl-X Ctrl-F <F7>
FIND-UNBALANCED-PARENS	Alt-X <i>command-name</i>
FORWARD-CHAR	Ctrl-F <Right Arrow>
FORWARD-ISEARCH	Ctrl-S
FORWARD-LIST	Ctrl-Z N
FORWARD-PAGE	Ctrl-X
FORWARD-PARAGRAPH	Esc]
FORWARD-SEARCH	Alt-X <i>command-name</i>
FORWARD-SEXP	Ctrl-Z F Ctrl-<Pg Dn>
FORWARD-UP-LIST	Ctrl-Z)
FORWARD-WORD	Alt-F Ctrl-<Right Arrow>

I

INDENT-FOR-COMMENT	Esc ;
INDENT-LISP-LINE	Ctrl-I

INDENT-RIGIDLY	Ctrl-X <Tab>
INDENT-SEXP	Ctrl-Z I Ctrl-Z Q
INDENT-TEXT-LINE	Alt-X <i>command-name</i> Ctrl-I in text mode
INSERT-BINARY-FILE	Alt-X <i>command-name</i>
INSERT-COMMAND-NAME	Alt-J
INSERT-DOS-OUTPUT	Alt-X <i>command-name</i>
INSERT-FILE	Ctrl-X I
INSERT-REGISTER	Ctrl-X G

J

JUST-ONE-SPACE	Ctrl-Z <Space> Esc \
-----------------------	-------------------------

K

KILL-BUFFER	Ctrl-X K
KILL-COMMENT	Ctrl-Z ;
KILL-LINE	Ctrl-K
KILL-REGION	Ctrl-W
KILL-SEXP	Ctrl-Z K
KILL-WORD	Alt-D

L

LISP-INTERACT	Esc <Enter>
LISP-INTERACTION	Alt-X <i>command-name</i>

LISP-MODE	Alt-X <i>command-name</i>
LIST-BUFFERS	Ctrl-X Ctrl-B
LOAD-KEYBOARD-MACROS	Alt-X <i>command-name</i>
LOWERCASE-REGION	Ctrl-X Ctrl-L
LOWERCASE-WORD	Alt-L

M

MAKE-MATCHING-()	Esc (
MARK-BEGINNING-OF-BUFFER	<F5> < Ctrl-<
MARK-END-OF-BUFFER	<F5> > Ctrl->
MARK-PAGE	Ctrl-X Ctrl-P
MARK-SEXP	Ctrl-Z @ Ctrl-Z ^ @
MARK-WHOLE-BUFFER	Ctrl-X H
MOVE-SCREEN-OTHER-WINDOW	Alt-X <i>command-name</i>

N

NAME-KEYBOARD-MACRO	Alt-X <i>command-name</i>
NEWLINE	<Enter> Ctrl-M
NEWLINE-INDENT	Ctrl-<Enter> Ctrl-J
NEXT-COMMENT-LINE	Alt-N
NEXT-LINE	Ctrl-N <Down Arrow>
NORMAL-MODE	Alt-X <i>command-name</i>

NUMERIC-ARG-PREFIX	Ctrl-U
	O
ONE-WINDOW	Ctrl-X 1
OPEN-INDENTED-LINE	Ctrl-Z O
OPEN-LINE	Ctrl-O <Ins>
OTHER-WINDOW	Ctrl-X O
	P
PARSE-ATTRIBUTE-LIST	Alt-X <i>command-name</i>
PREVIOUS-COMMENT-LINE	Alt-P
PREVIOUS-LINE	Ctrl-P <Up Arrow>
PUSH-TO-DOS	Alt-X <i>command-name</i>
	Q
QUERY-REPLACE	Esc %
QUOTED-INSERT	Ctrl-Q
	R
READ-FILE	Ctrl-X Ctrl-R <F8> Ctrl-V
RECENTER-DEFINITION	Ctrl-Z R
RECENTER-POINT	Alt-R

RECENTER-WINDOW	Ctrl-L
REMOVE-SURROUNDING-()	Ctrl-Z \ Ctrl-Z
REPLACE-STRING	<F5> % Ctrl-% Ctrl-X R
REVERSE-ISEARCH	Ctrl-R
REVERSE-SEARCH	Alt-X <i>command-name</i>
RUBOUT	Alt-X <i>command-name</i>
RUBOUT-HACKING-TABS	Ctrl-H <Rubout>
S	
SAVE-ALL-FILES	Ctrl-X Ctrl-M
SAVE-FILE	Ctrl-X Ctrl-S <F9>
SAVE-FILES-EXIT	Alt-X <i>command-name</i>
SAVE-FILES-PUSH-TO-DOS	Alt-X <i>command-name</i>
SAVE-KEYBOARD-MACROS	Alt-X <i>command-name</i>
SCROLL-DOWN	Alt-X <i>command-name</i>
SCROLL-OTHER-WINDOW	Ctrl-Z V
SCROLL-SCREEN-DOWN	Ctrl-V <Pg Dn>
SCROLL-SCREEN-UP	Alt-V <Pg Up>
SCROLL-UP	Alt-X <i>command-name</i>
SELECT-BUFFER	Ctrl-X B <F3>

SELECT-PREVIOUS-BUFFER	<F4> Ctrl-Z L
SET-BUFFER-PACKAGE	Ctrl-Z :
SET-COMMENT-COLUMN	Ctrl-X ;
SET-FILL-COLUMN	Ctrl-X F
SET-FILL-PREFIX	Ctrl-X Alt-F
SET-MODE	Alt-X <i>command-name</i>
SET-POP-MARK	Ctrl-@ <F5> <Space> Ctrl-<Space>
SET-VARIABLE	Alt-X <i>command-name</i>
SHOW-POSITION	Esc = Ctrl-X =
SHOW-REGISTERS	Alt-X <i>command-name</i>
SHOW-VERSION	Alt-X <i>command-name</i>

T

TAB-TO-TAB-STOP	<Tab>
TAGS-ADD-FILE	Alt-X <i>command-name</i>
TAGS-ADD-FILES	Alt-X <i>command-name</i>
TAGS-CONTINUE-MAP	Ctrl-Z .
TAGS-FIND-ALL	Alt-X <i>command-name</i>
TAGS-FIND-DEFINITION	Esc .
TAGS-INDEX-FILE	Alt-X <i>command-name</i>
TAGS-LOAD-INDEX	Alt-X <i>command-name</i>
TAGS-LOAD-TABLE	Alt-X <i>command-name</i>
TAGS-MAKE-INDEX	Alt-X <i>command-name</i>

TAGS-QUERY-REPLACE	<i>Alt-X command-name</i>
TAGS-REMOVE-FILE	<i>Alt-X command-name</i>
TAGS-REPLACE-STRING	<i>Alt-X command-name</i>
TAGS-SAVE-TABLE	<i>Alt-X command-name</i>
TAGS-SEARCH	<i>Alt-X command-name</i>
TAGS-SHOW-TABLE	<i>Alt-X command-name</i>
TAGS-USE-TABLE	<i>Alt-X command-name</i>
TEXT-MODE	<i>Alt-X command-name</i>
TRANPOSE-CHARACTERS	Ctrl-T
TRANPOSE-LINES	Ctrl-X Ctrl-T
TRANPOSE-REGIONS	Ctrl-X T
TRANPOSE-SEXPS	Ctrl-Z T
TRANPOSE-WORDS	Alt-T
TWO-WINDOWS	Ctrl-X 2

U

UNBIND-KEYBOARD-MACRO	<i>Alt-X command-name</i>
UNLOAD-BUFFER	<i>Alt-X command-name</i>
UNMODIFY-BUFFER	Esc ~
UNTABIFY-REGION	<i>Alt-X command-name</i>
UPDATE-ATTRIBUTE-LIST	<i>Alt-X command-name</i>
UPDATE-MODE-ATTRIBUTE	<i>Alt-X command-name</i>
UPDATE-PACKAGE-ATTRIBUTE	<i>Alt-X command-name</i>
UPPERCASE-INITIAL	Alt-C
UPPERCASE-REGION	Ctrl-X Ctrl-U

UPPERCASE-WORD

Alt-U

V**VIEW-FILE**

Ctrl-X V

W**WINDOW-BACKWARD-PAGE**

Ctrl-X {

WINDOW-FORWARD-PAGE

Ctrl-X }

WRITE-FILE

Ctrl-X Ctrl-W

Y**YANK**

Ctrl-Y

YANK-POP

Alt-Y

GCLISP 386
Low-Level Interface Guide



Chapter 18

Architecture of the GCLISP Environment

Introduction

Most users of the GCLISP 386 Developer do not need to read this Guide. The material presented here is relevant only if you intend to directly interface GCLISP programs to DOS, BIOS, or external hardware or software.

The topic of this document is the low-level interface services provided by GCLISP, which are similar to those provided by DOS, but not identical.

Hardware Level

At the hardware level, the GCLISP environment treats the computer as a dual-processor system consisting of the following two pseudo-processors:

- A protected-mode processor with up to 15 megabytes of physical memory (high memory)
- A real-mode processor with up to 1 megabyte of physical memory (low memory)

The two processors run synchronously and communicate with each other through shared memory. Mass storage and other external devices are controlled by the real-mode processor.

Operating-System Level

The GCLISP environment is an extension of BIOS/DOS. File system, information, and I/O services are essentially identical to those provided by BIOS/DOS

The GCLISP kernel (operating system level) is comprised of two parts:

- P-KERNEL runs in protected mode and services requests made by software running in protected mode. Some requests are processed in protected mode, while others--primarily file and I/O services--are relayed to the R-KERNEL.
- R-KERNEL runs in real mode as a regular DOS program. R-KERNEL receives requests from P-KERNEL and invokes appropriate BIOS/DOS services in response.

The communication between P-KERNEL and R-KERNEL is achieved via a shared-memory message passing mechanism.

Applications Level

At the application level, programs written in GCLISP run in protected mode. All interrupt requests are handled by the P-KERNEL, which either executes the requests or relays them to R-KERNEL.

Co-resident DOS programs (such as device drivers, desktop utilities, or programs invoked from within the GCLISP 386 environment by the use of the SYS:EXEC or SYS:DOS function) run in real mode and behave indistinguishably from any such program running under the normal DOS environment. However, communication between GCLISP 386 and such programs must recognize certain constraints, discussed in Chapter 20.

Glossary of Terms

Several terms are used in this *Guide* with particularly specific meanings. These terms are defined below.

BIOS Throughout this *Guide*, BIOS refers to the COMPAQ DESKPRO 386 BIOS only.

DOS Throughout this *Guide*, DOS refers to MS-DOS version 3.1.

High Memory "Extended memory", that is, RAM above the 1M mark, not normally addressable by DOS.

Low Memory "Base memory", that is, RAM below the 1M mark, generally addressable by DOS.

Protected Mode The execution mode of the 80286 which supports physical addressability of up to 16M bytes of memory, and which fully utilizes the built-in memory-management capability of the 80286.

Real Mode The execution mode of the 80286 that supports the 8086-compatible memory-addressing scheme, with the same 1M limitation as the 8086.

Chapter 19

GCLISP Environment

File System

The file system of the GCLISP environment is completely compatible with the DOS environment and is subject to the same limitations (for example, there can be no more than twenty files open for a program at any one time, including required system files).

The only difference is that file I/O may be somewhat slower because the I/O requests must be relayed to the real mode, and the data buffers copied to (or from) low memory each time an I/O operation is performed.

Memory Usage

LISP programs run in protected mode. Co-resident DOS routines run in real mode. LISP programs cannot access low memory directly; real-mode routines cannot use protected-mode memory, except for a small segment used for communication.

sys:%sysint Services

The function **%sysint**, described in the *GCLISP 386 Developer Reference Manual*, provides a way for the programmer to directly generate a software interrupt request from GCLISP. This function is used for two purposes:

- Invoking GCLISP interrupt service routines directly. Such service routines fall into three categories:

- **DOS-Compatible INT-21 Services**
- **BIOS-Compatible Interrupt Services**
- **GCLISP Extended INT-21 Services:** These are system services that are particular to the GCLISP 386 architecture. These service routines, described in Chapter 20, are implemented as an extension to the INT-21 system call mechanism and are invoked using the same interface protocol.
- **Communicating with interrupt handlers installed in real mode.** For interrupts other than 10, 11, 16, and 21, register values are passed to the real-mode kernel, and the interrupt is issued there. This topic is discussed in Chapter 20.

Note that `sys:%sysint` has limited error-checking capabilities. You must insure that all pointers passed are valid, and that all interrupt numbers and function codes are defined. A fatal error, which aborts GCLISP, may occur if invalid values are passed using `sys:%sysint`.

DOS-Compatible INT-21 Services

This subsection lists the DOS INT-21 functions supported by GCLISP. Many of these functions are not fully compatible with their DOS counterparts; limitations and restrictions are noted in the list. For a complete interface specification, consult the MS-DOS 3.1 Technical Reference Manual.

The functions described in this section should not be used unless equivalent GCLISP functions are unavailable or unsuitable. Their improper use may have undesirable effects on the integrity of the system.

Some of the functions (*e.g.*, `set vector` and `get vector`) pertain only to the protected-mode environment. They cannot be used to manipulate the equivalent real-mode environment.

<u>Code</u>	<u>Description</u>	<u>Restrictions</u>
01h	Keyboard Input	
02h	Display Output	
03h	Auxiliary Input	

04h	Auxiliary Output	
05h	Printer Output	
06h	Direct Console I/O	
07h	Direct Console Input Without Echo	
08h	Console Input Without Echo	
09h	Print String	The string must be no longer than 256 bytes.
0Ah	Buffer Keyboard Input	The string must be no longer than 256 bytes.
0Bh	Check Standard Input Status	
0Ch	Clear Keyboard Buffer and Invoke a Keyboard Function	
0Dh	Disk Reset	
0Eh	Select Disk	
19h	Current Disk	
1Ah	Set DTA	The DTA size must be at least 44 byte
25h	Set Vector	This function affects only the vectors in protected mode. It cannot be used for setting software interrupt handlers for programs running in real mode
29h	Parse Filename	The string must be no longer than 256 bytes.
2Ah	Get Date	
2Bh	Set Date	
2Ch	Get Time	
2Dh	Set Time	
2Eh	Set/Reset Verify Switch	
2Fh	Get DTA	
30h	Get DOS Version Number	
33h	Ctrl-Break Check	
35h	Get Vector	Only the protected-mode vectors are involved.
36h	Get Disk Free Space	
39h	Create Sub-Directory (MKDIR)	The ASCIIZ string must be no longer than 256 bytes.
3Ah	Remove Sub-Directory (RMDIR)	The ASCIIZ string must be no longer than 256 bytes.
3Ch	Create File (CREAT)	The ASCIIZ string must be no longer than 256 bytes.
3Dh	Open File	The ASCIIZ string must be no longer than 256 bytes.
3Eh	Close File	
3Fh	Read From File/Device	Up to 512 bytes can be read.

40h	Write To File/Device	Up to 512 bytes can be written.
41h	Delete File (UNLINK)	The ASCIIZ string must be no longer than 256 bytes.
42h	Move File Pointer (LSEEK)	
43h	Change File Mode (CHMOD)	The ASCIIZ string must be no longer than 256 bytes.
44h	I/O Control for Device (IOCTL)	Functions 2, 3, 4, 5 (for reading from or writing to device channels) are not supported.
45h	Duplicate File Handle (DUP)	
46h	Force Handle Duplicate (CDUP)	
47h	Get Current Directory	
4Dh	Get Return Code of Subprocess	
4Eh	Find First Matching File	The ASCIIZ string must be no longer than 256 bytes. Also, a Set DTA (1Ah) request must have been issued before issuing this request.
4Fh	Find Next Matching File	
54h	Get Verify Setting	
57h	Get/Set File's Date and Time	
59h	Get Extended Error	
5Ch	Lock/Unlock File Access	

BIOS-Compatible Services

All BIOS interrupt services are supported. The following notes apply:

- **INT 12h (Memory Size):** This service returns the size of the low memory; high memory is not included in the value returned.

BIOS interrupt services should not be used unless equivalent GCLISP functions are unavailable or unsuitable. Their improper use may have undesirable effects on the integrity of the system.

Chapter 20

Interface to Real Mode

Overview

GCLISP can interface to co-resident DOS programs running in real mode. Such co-resident programs are either:

- DOS programs spawned from within GCLISP 386 via the use of **SYS:DOS** or **SYS:EXEC**
- Terminate-and-stay-resident programs loaded prior to invoking GCLISP

In either case, such programs behave normally (that is, as if they were operating in the normal DOS environment). However, terminate-and-stay-resident programs that require direct communication with GCLISP programs must follow the interface procedure described later in this chapter.

Running a DOS Program

To run a DOS program from GCLISP, use **SYS:DOS** or **SYS:EXEC** as described in the *Reference Manual*. The code runs in real mode; it is not possible at present to load other protected mode programs or protected-mode interrupt handlers.

The spawned DOS environment is subject to the following restrictions:

- An INT-15 BIOS call to determine the amount of high memory (function code **88H**) will return **AX = FFFFH**, signaling the presence of GCLISP LM in high memory.

- An INT-15 BIOS call to switch into protected mode (function code 89H) simply returns AH = FFH, signaling failure in switching.
- An INT-15 BIOS call to copy memory to/from high memory (function code 87H) simply returns with CY = 1 and AH = 01H, signaling a memory parity error.

Running a Terminate-and-Stay-Resident Program

To use a terminate-and-stay-resident program, follow these steps:

1. Run the program in real mode. It should install itself as an interrupt handler.
2. Run GCLISP.
3. If the arguments to, or results from, the DOS program are too large to pass in the registers, use the functions described later in this chapter to allocate low memory and copy objects between low memory and high memory.
4. Call the real-mode routine by issuing its interrupt (using `sys:%sysint` or using the REALINT request, described below). Use the registers to pass arguments and return results. The arguments and results can include pointers to low memory, allowing large arguments and results to be passed.

Managing low memory

Allocating and Accessing low memory

The following GCLISP INT-21 services, accessed using `sys:%sysint`, allow you to access, allocate and free low memory.

Request 88h--Define Segment (DEFSEG)

Define a segment selector for accessing already allocated low memory. Note that the use of this function defies the memory

protection mechanism; but it is the only way to access buffers residing in a real-mode device driver.

All selectors defined using this function should be explicitly undefined (using UNDEFSEG) when no longer needed.

Input:	AH	= 88h
	DL:BX	= absolute byte address of segment start
	CX	= number of bytes in segment (0 means 64K)
Output:	CY	= 0
	AX	= protected-mode selector for the segment
Error:	CY	= 1
	BX	= 0 if there are no more free segment selectors available.

Request 89h--Undefine Segment (UNDEFSEG)

Undefine a segment selector that refers to memory. Note that because it does not check against inadvertent deletion of segment selectors used internally by the GCLISP system, this function should be used with extreme caution.

Input:	AH	= 89h
	BX	= selector to be released
Output:	CY	= 0
Error:	CY	= 1 if selector invalid

Request 91h--Low Memory Data Segment Alloc (LMALLOC)

Allocate a memory block from the DOS heap space in low memory.

All memory allocated using this function should be explicitly deallocated (using LMFREE) before GCLISP is exited.

Input:	AH	= 91h
	CX	= number of bytes to allocate (0 means 64K)
Output:	CY	= 0
	DX	= protected-mode selector of the allocated block
	AX	= real-mode paragraph number of the allocated block
Error:	CY	= 1
	DX	= 0 if there is insufficient memory in the DOS heap space, or if there are no more free segment selectors available.

Request 92h--LM Data Segment Free (LMFREE)

De-allocate a memory block (to the DOS heap space) in low memory.

Input:	AH	= 92h
	DX	= protected-mode selector of the allocated block
	BX	= real-mode paragraph number of the block
Output:	CY	= 0
Error:	CY	= 1 if either of the input addresses is invalid.

Reading, Writing, and Copying Objects in low memory

Reading and Writing

To read and write objects in low memory, use **%contents**, **%contents-store**, and related functions (**%contents-byte**, **%contents-store-byte**, etc.), described in the *Reference Manual*. These functions should normally be used to reference valid GCLISP objects, explicitly allocated memory blocks, or the memory regions defined by the following special segment selectors:

B0H	Monochrome Screen Refresh Buffer (0B0000H to 0B0FFFH)
B8H	Color/Graphics Screen Refresh Buffer (0B8000 to 0BBFFFH)

An invalid pointer specification for `sys:%contents` will result in a memory-protection error. However, because `sys:%contents` always references four bytes each time it is invoked, under some conditions the returned values may only be partially valid. When only part of the memory referenced is invalid, GCLISP returns `nil` for invalid values, rather than signalling a memory-protection error.

Copying

To copy objects between low memory and high memory, use `MCOPY`, a `sys:%sysint` request:

Request 90h -- Memory Copy (MCOPY)

Copy a block of memory. This function should be used only for copying objects inside GCLISP environment to the external environment, or vice versa. Both the source and destination addresses must be in protected-mode format, whether they refer to high memory or low memory.

Input:	AH	= 90h
	DS:DX	= source address
	ES:BX	= destination address
	CX	= number of bytes to copy (0 means 64K)

Output:	CY	= 0
---------	----	-----

Error:	CY	= 1 if the specified addresses are invalid, or if a segment boundary would be crossed during the copying process.
--------	----	---

Note. In the latter case, no action is taken--that is, there is no partial copying.

Communicating with Co-Resident Programs

To communicate with co-resident programs, use `sys:%sysint`, specifying the interrupt handled by the program you wish to call. This function allows you pass a number of registers to the called program.

If you need to pass the SI, DI, and BP registers, which cannot be passed with `sys:%sysint`, use `REALINT`, described below.

In either case, the registers can contain the addresses of objects in low memory (obtained from `BASEALLOC` or `DEFBASESEG`). Do not attempt to pass pointers to objects in high memory. Such pointers cannot be interpreted by routines running in real mode.

The routine should return control to `GCLISP` by executing an `iret` instruction.

REALINT Request

`REALINT` is itself a `sys:%sysint` request, number 93h. It is described below.

Request 93h--Real Mode Interrupt (REALINT)

Generate a real-mode interrupt with real-mode register values passed in the machine state block (MSB). This is the only way to generate a real-mode interrupt where one can pass meaningful register values for SI, DI, and BP directly, or return DS and ES.

Note on flags: Only the arithmetic flags are meaningful; others are ignored on a call and returned as 0.

Input:	AH	= 93h
	AL	= interrupt number
	DS:DX	= pointer to machine state block (see Figure 1 for format)
Output:	CY	= 0
	[DS:DX]	= register block contents updated with values returned from the interrupt
Error:	CY	= 1 if the pointer for the MSB is invalid.

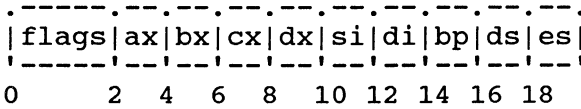


Figure 1: Format of Machine State Block (MSB)

Other Services

The following additional `sys:%sysint` services are available:

Request 84H -- Get Segment Alias (SGALIAS)

Get the segment selector of the segment which is an alias of the given selector and which has the specified segment type. If the given selector is already of the right type, it will simply be returned as the the result. If an alias of the right type does not exist, a new one will be created and returned as the result.

For definitions of the terms "segment", "segment type", and "segment alias", consult the *Intel iAPX286 Programmer's Reference Manual* and the *Intel iAPX286 Operating System Writer's Guide*.

Input:	AH	= 84h
	AL	= segment type
	DX	= segment selector
Output:	CY	= 0
	AX	= segment selector for segment alias
Error:	CY	= 1 if there are no more free segment selectors available, or if the given selector is

Request 87h -- High Mem Alloc (HMALLOC)

Allocate a block of high memory. This memory, once allocated, cannot be returned (that is, there is no corresponding de-allocate function), and thus cannot be re-allocated by GCLISP.

Input:	AH	= 87h
	CX	= number of bytes to allocate (0 means 64K)
Output:	CY	= 0
	DX	= selector of the allocated block
Error:	CY	= 1
	DX	= 0 if there is insufficient high memory available

**GCLISP 386
Compiler Guide**

Chapter 21

Using the Compiler

Loading the Compiler

The compiler is loaded automatically the first time you call it in any GCLISP 386 session. Loading is triggered by any of the following:

- a **compile** command or a **compile-file** command to the GCLISP interpreter
- a GMACS editor command requesting compilation of an buffer or a form

Loading the compiler takes about a minute. If loading is being done from within GMACS, your only signal of the loading activity will be the flashing red light on your disk drive.

If you want, you can load the compiler explicitly, without waiting until the automatic load is triggered. To do so, type:

```
(cd "c:\\gclisp3\\compiler")  
(load "loadcomp")
```

The first command sets your DOS default directory to **c:\gclisp3\compiler**. The second command performs the load (but does not execute the compiler).

Once the compiler has been loaded, it can be used in GCLISP, as described in the following sections of this *Guide*.

Running the Compiler

To run the compiler, run either the `compile-file` function or the `compile` function in the GCLISP interpreter. The syntax of these interpreter functions, and their effects, are described in the following sections. For a description of how the `compile` function can be invoked from within GMACS, see page 129.

Preparation

Before running the compiler:

- **Set the current package:** When running the compiler, the current package should be a package that will be present in the environment where the compiled function will be run. A safe tactic is always to compile from the user package, which is always present.
- **Set the atom-to-cons ratio:** Compiled code is a heavy user of atom space. When the compiler is to be run, a reasonable atom-to-cons-space ratio in the interpreter environment is 6:1. This atom-weighted space allocation can be obtained by editing the file `config.lsp` to set the variables `*initial-atom-weight*` and `*initial-cons-weight*` to 6 and 1 respectively.


The `compile-file` Function

```
compile-file input-file &key :output-file :error-file
           :lap-file :asm-file :load
```

This command takes a file of GCLISP source code as input. It produces an output file of compiled (machine-language) code, called a "fas" (for "fast-load") file, and optionally three other files, which may be useful for debugging.

The function compiles the source code in the file *input-file*; creates an output file; and writes the fas (compiled) output to the output file.

The keyword arguments have these meanings. (Note. When an output argument is a stream, the user must explicitly open and close the stream.)

- :output-file** If the **output-file** keyword argument is not supplied, or if the value of the argument is **t**, then the output file has the same pathname as the input file, but with the filetype **fas**. If a **fas** file of the same name already exists, then it will be replaced by the new file.
- If the keyword argument is a filename without a filetype, then the output file has the same name as this argument with the filetype **fas**.
- If the argument is a filename with a filetype, then the output file has this name and type.
- If the argument is **nil**, then no output file is created.
- The argument cannot be a stream.
- :error-file** If the **error-file** keyword argument is not supplied, or if its value is **t**, then the error file has the same pathname as the input file, but with the filetype **err**.
- If the keyword argument is a filename without a filetype, then the error file has the same name as this argument with the filetype **err**.
- If the argument is a filename with a filetype, then the error file has this name and type.
- If the argument is **nil**, then no error file is created.
- The argument may also be an output stream.
- In any case, all messages created during the compilation will be printed to the ***standard-output*** stream.
- :lap-file** If the **lap-file** keyword argument is not supplied, or if the argument is **nil**, then no lap file is created.
- If the keyword argument is **t**, then the lap file has the same name as the input file, but with the filetype **lap**.
- If the keyword argument is a filename without a filetype, then the lap file has the same name with the filetype **lap**.
- 

If the argument is a filename with a filetype, then the lap file has this name and type.

The argument may also be an output stream.

The lap file is an intermediate-code file which may be useful during debugging.

:asm-file

If the **asm-file** keyword argument is not supplied, or if the argument is **nil**, then no asm file is created.

If the keyword argument is **t**, then the asm file has the same name as the input file, but with filetype **asm**.

If the keyword argument is a filename without a filetype, then the asm file has this name with the filetype **asm**.

If the argument is a filename with a filetype, then the asm file has the same name and type.

The argument may also be an output stream.

The asm file is an intermediate-code file which may be useful during debugging.

:load

If the **load** keyword is specified as **t**, then the output file is loaded after compilation; otherwise, it is not.

The beginning of the output **fas** file contains certain useful information. The first two words of the file are "FASL FILE", identifying the file as a **fas** file. Following this are sets of property names and property values. Currently this information includes:

- source-file name
- interpreter version number
- compiler version number
- creation date

You can see this information by using the DOS command **type**, at DOS command level, on the **fas** file.

The compile Function

compile *function* &optional *definition* &key
 :lap-file :asm-file :error-file

This function takes as input the interpreted definition of a function. It generates fas-code output and, optionally, lap-code output or asm-code output.

compile replaces the original interpreted code (in the interpreter environment) with the compiled function. It does not save the compiled form in an output file.

Subsequently, calls on the function that was compiled are run as calls on the compiled version, with one exception: if the function was proclaimed **inline** (see page 211) before it was compiled, and before the calling function is compiled, then, when the calling function is compiled, the interpreted version of the called function is inserted in-line in the calling function prior to compilation.

Note that you must specify the optional *definition* in the command in order to use the keywords. Thus, you should specify it as **nil** if you are not supplying it.

function, definition

The argument *function* should be a symbol. If the optional *definition* is supplied, it must be a lambda-expression -- specifically, the interpreted function that the user wishes to have compiled.

If *function* is a non-nil symbol, then the compiled-function object code is installed as the function definition of the symbol, and the symbol is returned.

If the argument specified for *function* is **nil**, then the compiled-function object itself is returned.

If the optional *definition* is not supplied, then *function* must be a symbol with a definition that is a lambda-expression; that definition is compiled and the resulting compiled code is put back into the symbol as its function definition.

:error-file

If no keyword argument is supplied or the keyword argument is **t**, then the error messages will go to ***standard-output***.

If the keyword argument is a filename without a filetype, then the error file has this filename with the filetype `.err`.

If the argument is a filename with a filetype, then the error file has the same name.

The argument may also be an output stream.

:lap-file

If the `lap-file` keyword argument is not supplied, or if the argument value is `nil`, then no lap file is created.

If the keyword argument is `t`, then the lap-file output will go to `*standard-output*`.

If the keyword argument is a filename without a filetype, then the lap file has this name with the filetype `lap`.

If the argument is a filename with a filetype, then the lap file has the same name and type.

The argument may also be an output stream.

:asm-file

If the `asm-file` keyword argument is not supplied, or if the argument is `nil`, then no asm file is created.

If the keyword argument is `t`, then the asm-file output will go to `*standard-output*`.

If the keyword argument is a filename without a filetype, then the asm file has this name with the filetype `asm`.

If the argument is a filename with a filetype, then the asm file has the same name and type.

The argument may also be an output stream.

Loading Compiled Code

A `.fas` file created by the `compile-file` function is loaded into GCLISP using the `load` or `fasload` functions, described in the *Reference Manual*. (See also page 44.)

Note that if the *pathname* argument to the `load` function does not include a filetype, then `load` will look first (in the specified directory) for a file with filetype `fas` and then, if that's not found, for a file of filetype `lsp`. Regardless of file-creation date, a file of filetype `.fas` will be preferred over a file of filetype `lsp`.

Loading a `.fas` file, like loading a file of LISP source code (a `.lsp` file), includes execution of the code in the file by the interpreter. The compiled code is smaller and faster than the original source file. When a `.fas` file is loaded, the compiled functions replace the uncompiled versions in the GCLISP environment.

Controlling Compiler Operation

The following symbols, variables, and proclamations affect the running of the compiler:

`compiler::compiler-version`

This variable holds the current version number of the compiler.

`:compiling-for-gclisp`

When compiling any form, the compiler places this symbol on the `*features*` list. This can be used to conditionalize code for special purposes -- for example, compiling for GCLISP as opposed to compiling for other LISPs.

`compiler::*optimize-space*`

In order to produce code that runs fast, some functions are coded in-line with a somewhat lengthy code sequence. To minimize the size of compiled code, set this variable to `t` (it defaults to `nil`). This will cause the aforementioned functions to compile into a shorter code sequence (which will take somewhat longer to execute, in most cases). In a future release of the compiler, this flag will have a larger effect on code size than it does currently.

`compiler::*verbose*`

If this user-settable variable is non-`nil`, then more detailed information concerning the processing of forms is output to the display and to the file specified in the optional `:error-file` argument, if one is supplied. Note that this variable is initially non-`nil`.

ignore

The **ignore** variable tells the compiler that a return value will not be used and should be ignored. Typically, **ignore** is used with functions such as **multiple-value-bind** and **multiple-value-setq**, and with lambda lists.

For example, suppose that your source program includes code of this form:

```
(multiple-value-setq (nil x y) form)
```

The compiler may return this warning message:

```
NIL is an obsolete variable in the list  
of a MULTIPLE-VALUE-SETQ,  
using the symbol IGNORE.
```

This means that in the compiled output for the above, the compilation of **nil** has been replaced by the compilation of **ignore**. You might as well have written **ignore** instead of **nil** in the source program.

```
(proclaim '(inline function-name ... ))
```

```
(proclaim '(notinline function-name ... ))
```

These GCLISP proclamations take effect only in the compiler. They instruct the compiler to inline-code or not to inline-code any call on the specified function(s).

Note that the proclamation must appear before the specified function is compiled, to instruct the compiler to preserve code which can be compiled in-line into any calling function. The proclamation must also be before any calling functions are compiled; otherwise, the call will be compiled as a function call (on the compiled version of the function, if that exists; and otherwise on the interpreted version of the function).

```
(declare '(inline function-name ... ))
```

```
(declare '(notinline function-name ... ))
```

These GCLISP declarations take effect only in the compiler. They instruct the compiler to inline-code or not to inline-code any call on the specified

function(s). The declaration is in effect for the body of the special form in which it appears.

(declare '(ignore *variable-name* ...))

This GCLISP declaration takes effect only in the compiler. It instructs the compiler that the binding of the specified variable is never used. The declaration is in effect for the body of the special form in which it appears.

Note: **c** is a synonym for the package name **compiler**.

Chapter 22

Programming Notes

Interpreter Compatibility

The GCLISP compiler supports the language implemented by the GCLISP interpreter in the GCLISP 386 Developer Version 2.2, with these exceptions (supported in the interpreter, but not in the compiler):

- The "#," reader macro, which arranges that the following form is evaluated when the file is loaded
- The use of **go** and **return-from** between different functions in the same lexical environment

An additional restriction is that a compiled function can have a maximum of 255 constants. If this limit is exceeded, the compiler produces the error message "ERROR: Too many constants for compiled function".

It is not always clear how many constants will be defined by a function definition, since macro expansion can generate an unobvious number of constants.

If the error message appears, break up the desired function definition into several smaller function definitions that call one another as needed.

The Compilation and Compiled-Code Environments

Here are some important points to be aware of when using the GCLISP compiler.

When to debug There is a trade-off between safe code and fast code because, on conventional computer architectures, fast code cannot do a lot of error checking. By default, the compiler produces code which is optimized for speed of execution. For most functions, the types of the arguments of a compiled function are not checked when the compiled code is run. This is especially true of functions which are compiled in-line.

For this and other reasons, it is good software-engineering practice to debug a given function completely in the interpreter before running it compiled. (A future release of the compiler will feature a compile-time flag to increase the safety of compiled code, but at the expense of code size and speed.)

Atom-to-cons ratio

Interpreted code is a heavy user of cons space, whereas compiled code is a heavy user of atom space. You will probably want to adjust the atom-to-cons ratio in your `config.lsp` file to reflect the relative weight of interpreted code versus compiled code in your current LISP environment.

Structuring a large system

LISP is order-dependent in the way it compiles certain macros and functions. For example, macros must be defined before the functions that use them are compiled. Similarly, any declarations using `defvar`, `defconstant`, `defparameter`, and `defstruct` must be made before the symbols that they define can be referred to.

For large systems, it may be useful to collect all declarations of this type into a file that is compiled and loaded before the rest of the system's files are compiled. (If the system to be compiled fits into one file, then all definitions of this type should be placed at the beginning of the file.)

Evaluation of forms

A difference between loading a file of LISP code and compiling a file of LISP code is that typically, while a file is being compiled, the forms being processed are not evaluated. If the compilation of a function in a file depends on the evaluation of a form that occurred earlier in the file, and if the

earlier form is not a `defvar`, `defconstant`, `defmacro`, `defparameter`, `defstruct`, or package function, then the `eval-when` construct can be used to insure that the earlier form is evaluated when the compiler encounters it.

Compiled code and the autoload facility

The autoload facility currently does not work with compiled code. Functions that are autoloaded when called from the interpreter will not be autoloaded when called from within compiled code. Instead, the following message is produced:

Illegal function MACRO in internal function dispatcher

If your compiled code calls autoloaded functions, you must explicitly load them into the LISP environment (using `load`) before attempting to run the compiled code.

Incremental compilations from GMACS

When compiling incrementally from within the editor, or evaluating within the editor, make sure that you are in the correct package. To keep these processes simple, only one package should be associated with each file, rather than several packages. The GMACS command `^Z :` ("Ctrl-Z colon") can be used to set the package of the current buffer. (See the *GMACS Editor Guide*.)

While incrementally compiling the contents of a buffer, remember to periodically save (to a file) the changes you make during debugging. After you have made all your changes, the final file should be compiled if you want to save the final results in compiled form, since compiling from GMACS only replaces the in-memory version of code (that is, compiling from GMACS executes `compile`, not `compile-file`).

Debugging compiled code

Currently, the GCLISP debugger does not provide much information during debugging of compiled code. The debugger will find only anonymous forms on the stack (see Chapter 9). The debugger will, however, print any special bindings that the compiled code produces.

The `trace` function can be used to trace all the functions evaluated when a given function is called. Also, it can be useful to put print statements within a function you are compiling from within the editor.

Loading .fas and .lsp files

A `.fas` file is always preferred over a `.lsp` file, regardless of creation date.

Compatibility with earlier GCLISPs

Compiled code generated by the GCLISP compiler Version 2.2 will work only in the GCLISP interpreter Version 2.2 or greater. Code generated for the COMPAQ DESKPRO 386 will run only on that machine.

Source-Coding for Efficient Compiled Code

Defstruct

`defstruct`-defined data structures result in more efficient compiled code than lists or arrays. This is because the set functions and accessor functions for `defstruct`-defined structures are compiled into a small number of assembly instructions, as opposed to a function call which uses many more instructions.

Fixnum arithmetic

There are several fixnum-arithmetic functions which compile particularly efficiently. These are the standard arithmetic functions suffixed with a "%", for example `+%` and `logior%`. (See GCLISP Release Note 2.2 - 1 for details.)

Inline proclamations

(See page 211, regarding the `inline` proclamation.)

Characteristics of LISP Compilation

The GCLISP compiler translates interpreted LISP code into the target machine's native-machine-language code. This has the dual advantage of producing code that takes less memory, and much more efficient (faster) code, as compared with the interpreted code.

That much is true of a good compiler for any computer language. However, compiling in LISP, and LISP compiled code, have certain special features arising from particular characteristics of the LISP language, as described here.

Integration of compiled code

Although LISP is an interpreted language, LISP source code and interpreted code are easy to compile. Compiled code is completely integrated into the LISP environment. A compiled LISP program that has been loaded into the interpreter can access all of the functions already defined in the LISP system. Conversely, compiled functions can of course be called by other functions.

Manipulating programs as data

A well-known basic feature of the LISP language is its capacity for manipulating programs as data. However, this is true only for interpreted LISP programs: programs that are represented as lists. Once a program has been reduced to machine code through compilation, it can no longer be manipulated as data.

Error-checking

A LISP compiler helps verify that the code it is given is properly written. Of course, it verifies the syntactic correctness of the code. It will also flag calls on undefined functions.

The compiler as a LISP function

A LISP compiler is simply a LISP function in the LISP environment, unlike compilers for other languages. For this reason, it is easy to integrate calls on the compiler with calls on other functions, and to call the compiler from another function. It is also easy to control the environment in which compilation occurs. Macros are especially useful for this.

The macro language

A major difference between LISP and other programming languages is its macro facility. (Macros are frequently used to transform a form that is easily readable by humans into a more complex form that is easy for the compiler (or interpreter) to process.) Many other languages have a macro language, but LISP is unique in that the macro language and the LISP language itself are the

same. (This is not true of the C language, for example.)

Incremental compilation

A LISP compiler can compile incrementally, one form at a time. Incremental compilation is not practical (or not possible) in most other languages. It simplifies the debugging process, particularly when the incremental compiles are done within the editing environment.

Chapter 23

Error and Warning Messages

The GCLISP compiler produces these kinds of messages:

1. **Informational** messages
2. **Warnings**, which indicate that some unusual circumstance was encountered but that the user's code will probably compile correctly
3. **Error messages**, which indicate that the compiled code will run with unpredictable results.

The error and warning messages are listed below.

Warning Messages

Assuming *symbol* is a special variable.

The variable *symbol* was used as a free variable inside the indicated function. This message indicates that the compiler assumes the variable is **special**. However, this warning may mean that you have misspelled a lexical variable and need to correct it. Otherwise, make sure that you have proclaimed or declared the variable as **special**.

NIL is obsolete in the variable list of a MULTIPLE-VALUE-SETQ, using the symbol IGNORE instead.

To indicate that you do not want to use a return value in a **multiple-value-setq**, use the variable **ignore**.

Previously called with wrong number of args.

The compiler remembered that you previously called the function currently being compiled with a different number of arguments than it is being defined with.

Some declarations have been optimized away, since no variables were used in a MULTIPLE-VALUE-BIND.

A multiple-value-bind form did not use any of its return values, so it was eliminated during optimization. Some declarations were also lost, since the compiler does not yet support the locally special form.

Error Messages

function called with *count* args, wanted at least *min*.

function was called with too few arguments.

function called with *count* args, wanted at most *max*.

function was called with too many arguments.

Function must be a symbol or lambda form: *function*

The object in a functional position of a form was syntactically incorrect.

GO to unknown tag: *tag*

The *tag* must be in the current lexical environment.

Ill-formed EVAL-WHEN situation list: *list* Ignoring its contents.

The only valid entries in a situation list of eval-when are eval, compile and load.

Illegal form *form* (as variable)

form is illegal where a variable is expected.

Illegal Lambda list: *lambda-list*

The lambda list *lambda-list* is incorrect. Either the order of &-keywords is wrong, or an unsupported &-keyword was specified.

Illegal literal object in function, should satisfy type

A quoted object *object* to the function *function* was of the wrong type. It should have been type *type*.

Illegal use of constant *symbol* as variable

symbol has previously been declared a constant, so it cannot be used as a variable.

***object* not a legal macro name.**

A macro name must be a symbol.

***object* not a legal function name.**

A function name must be a symbol.

Too few forms: *form*

The special form *form* was given too few sub-forms.

Too many forms: *form*, the last *count* will be ignored

The special form *form* was given too many sub-forms.

GCLISP 386
Foreign Language Interface Guide

Chapter 24

Introduction to the Interface

Overview

The GCLISP Foreign Language Interface package is used to bring other programming languages into the GCLISP 386 Developer environment.

It is important that users of GCLISP, and developers using the GCLRUN environment, may be able to use, as part of the systems they develop, programs written in other languages such as Lattice C or Microsoft C. In particular, users may want to call interface functions to database or spreadsheet systems from GCLISP with arguments, and use the return values in GCLISP programs.

This is possible in the basic GCLISP environment, but only through interrupt-handling systems in which a stay-resident interface function may be "called" from a GCLISP function (via a `sys:%sysint`-generated 80386 interrupt) and returned from via an interrupt-return (IRET) instruction. The Foreign Language Interface software implements an interface mechanism which is more accessible, more versatile, more portable, and smoother in operation.

The interface enables both *foreign calls* from LISP to compiled non-LISP functions, and *foreign entries* to LISP functions from compiled non-LISP functions.

The Foreign Language Interface is particularly useful in conjunction with GCLRUN, the full runtime loader and linker that can load and link `.fas` files generated by the GCLISP compiler running in the GCLISP 386 Developer. Programs written in a foreign language such as Lattice C or Microsoft C can be separately compiled in the foreign language, then debugged in the LISP environment of the 386 Developer, and then brought together using GCLRUN into your application runtime.

The foreign languages currently supported by the Foreign Language Interface are Lattice C and Microsoft C; and assembly languages which follow the stack-format and memory-addressing conventions of either of these languages.

Requirements and Assumptions

The Foreign Language Interface software consists of a number of GCLISP functions, included with the 386 Developer. In addition to this software, you also need a compiler for the foreign language you want to interface to.

You may also want to know the structure of the Intel object file produced by the foreign language compiler. This is documented in Intel's "OMF manual", formally titled *8086 Relocatable Object Module Formats: An Intel Technical Specification* (Santa Clara, California: Intel Corporation, 1981).

Detailed Requirements on the Foreign Code

The foreign language interface should not be used to access foreign-language functions compiled to .exe files. These contain only a single entry point and can already be called from GCLISP using the `sys:exec` command. The interface should be used only to access foreign-language functions which have been compiled to .obj object files or .lib library files, which can contain several entry points.

Call-Argument and Return-Value Data Types

The Foreign Language Interface currently supports Lattice C and Microsoft C call arguments of the following C parameter types:

char
int
long
float
double
pointer

This means:

- When a LISP function calls a C function, the call arguments can be of any LISP types convertible to these C data types.
- When a C function calls a LISP function, the call arguments can be of any of these C data types. They will be converted to the appropriate LISP types. Also, when a called LISP function returns to the calling C function, the LISP function can return a value of any LISP type convertible to one of these C data types.

As described in later sections of this guide dealing with **define-foreign-call** and **define-foreign-entry**, each C data-type name in the list above can appear in foreign-call and foreign-entry function definitions as the *type* in an ordered pair of the form (*name type*) (parentheses included).

An exception to this convention is the **pointer** type. The form of the ordered-pair specification for an argument of this type is not (*name type*), but rather ((*name-1 name-2 type*)) (all parentheses included). Correspondingly, the call argument when the foreign routine is invoked is two values, not one. The first value is the segment, and the second value is the offset, of the address which the C formal parameter will be set to point to.

These C-language return-value data types are currently supported:

char
int
long
float
double
pointer
void

This means that when a C function called from LISP returns to the calling LISP function, a return value of any of the above types will be converted to the appropriate LISP type.

void causes the calling LISP function to return no value. All others but **pointer** cause the calling LISP function to return a single value. **pointer** causes the calling LISP function to return two values. The first is the segment, and the second is the offset, of the data object pointed to.

Far Calls

All code written in the foreign language must be compiled to use `far` calls to enable Foreign Language Interface functions to stamp in long calls to LISP (unresolved or resolved) entry handlers. (A `far` call is a call which addresses the code of the target function by specifying both segment and offset, rather than specifying only the offset within the current segment.)

The Foreign Language Interface will not function if no compilation mode is available for assembling `far` calls to external entries. Using "large-model" compilation, if it is available, will accomplish this. If it is not, the original code must be written to ensure that only `far` calls will be generated during compilation.

Lattice C Requirements

To use the Foreign Language Interface with Lattice C, you need a Lattice C compiler, version 2.15 or later. You also need to set certain Lattice C compiler switches to make the object code generated by the compiler compatible with the requirements of the GCLISP environment, as follows (note that this may necessitate running separately the two passes of the LC compiler, LC1 and LC2):

- When invoking LC1, set a compiler switch (in version 2.15, the `-ml` switch) to specify code for the large-memory model. Set a switch also (in version 2.15, the `-s` switch) so that Lattice C will *not* do pointer manipulations with pointers in canonical form.
- When invoking LC2, set a compiler switch (in version 2.15, the `-v` switch) so that the C program will *not* do stack checking.

Do not use the Lattice C library memory management routines, and do not attempt any console input or output.

Microsoft C Requirements

To use the Foreign Language Interface with Microsoft C, you need a Microsoft C compiler of version 3.00 or later. You must also set certain Microsoft C compiler switches to make the object code generated by the compiler compatible with the requirements of the GCLISP environment:

- When invoking CL, set the **-ML** (or **/AL**) switch to specify code for the large-memory model.
- Be sure to set the compiler switch so that no stack checking occurs.

Do not use the Microsoft C library memory management routines, and do not attempt any console input or output.

Terminology

These terms are basic to your understanding and use of this guide and the software itself:

- | | |
|-------------------------|--|
| Foreign language | Any callable non-LISP computer language supported by the Foreign Language Interface (currently Lattice C and Microsoft C). |
| Foreign call | A function call from GCLISP to a foreign-language function. |
| Foreign entry | A function call from a foreign-language function to a GCLISP function. |
| Namespace | A LISP object consisting of a set of one or more related functions invoked by foreign calls or foreign entries. (These will be foreign-language functions, and LISP functions called from those foreign-language functions.) The functions in the namespace are structured into a single packet of code, and share a single static data segment. |
| Object file | A compiled foreign-language file in Intel (standard MS-DOS) .obj or .lib file format. |

Chapter 25

Basic Usage

The plan of the rest of this guide is as follows:

- We first describe, in the section "Interface Operations", the logical structure of the interface operations, tying particular operations to the names of particular interface functions and macros.
- The section "Operating Suggestions and Restrictions" details miscellaneous facts that will steer you around corners of the interface and potential problems in using it.
- Descriptions of the basic interface macros occupy the next two sections, under "The Function-Definition Macros." The section "An Example" illustrates these macros (**define-foreign-call** and **define-foreign-entry**). Other useful illustrations of **define-foreign-call** are in the chapter "Examples and Error Messages."
- Some other functions, and the global variables used by the interface, are listed in the last sections in this chapter.
- The complete Foreign Language Interface consists of about a dozen GCLISP functions, macros, and global variables. The sub-collection of these described in this chapter enables the programmer without significant GCLISP experience to do useful work with foreign-language programs in the GCLISP environment. The following chapter, "Programming with the Interface," describes the remaining functions.
- The last chapter, "Examples and Error Messages," also explains the error messages output from the interface.

Overview of Operations

Interface Operations

This is how the foreign language interface operates.

Suppose that you have written a function in a supported foreign language, which we'll take to be (Lattice or Microsoft) C, and you want to execute this function in the GCLISP environment.

To do so, you must define a LISP function to invoke the C function. Once this is done, you direct the interpreter to invoke the LISP function in the usual way; and that call to the LISP function invokes the C function.

To define the LISP function, execute the LISP macrodefine-foreign-call (described in a subsequent section). This serves to:

- Define a named LISP function
- Specify by name the C function it is to call
- Specify, by name and data type, the arguments which the LISP function will take and will pass through to the called C function at execution time (and similarly for the return values)

Besides these basic operations, **define-foreign-call** does the following:

- Finds the C function--its entry point and its code and data segments--in a DOS file with file extension **obj** or **lib**, using a DOS pathname you specify. This is done automatically by **define-foreign-call**, using lower-level calls on **load-object-file**, **find-object-entry**, and **find-data-entry**. (See the descriptions of these other functions in the preceding chapter.)
- Associates the C code segment and data segments with a *namespace*, the LISP object which serves to wrap collections of foreign code and data. (You specify which namespace in the call to **define-foreign-call**.)

- Establishes, if possible, a link within the LISP workspace from each external function reference within the C program to an entry point external to that C program. This is so that the C function can execute. The linking is done by lower-level calls on the function **resolve-unresolved-references**. Each additional entry point is of one of two kinds:
 - It may be an entry point to another C function. At run time, such a function must also be available in the workspace in order for it to execute: that is, it must have been loaded, into the same namespace. If it has not already been loaded when **resolve-unresolved-references** attempts to resolve a reference with that name, the reference will be flagged by **resolve-unresolved-references** as unresolved. You can then load the needed function by an explicit call to **load-object-file**. Or the function may be loaded by another call to **define-foreign-call**, if it so happens that you also want to call this C entry point explicitly from LISP.
 - It may be an entry point to some LISP function. This function must be defined in LISP by **defun**, like any other LISP function. It must also be "declared available" to be called from the C function. You make this declaration by executing **define-foreign-entry**. This call has the effect of declaring the LISP function to be callable from any C function with arguments with specified names and types, and a return value of a specified type.

In summary:

- **define-foreign-call** sets up a LISP function to call a specified C function.
- **define-foreign-entry** declares that a specified LISP function shall be callable by a C function (no particular one, but a class of functions with a particular structure of arguments and return values).

The process just described is illustrated in the section "An Example", below.

Other functions in the foreign language interface perform some auxiliary operations:

- **Namespace-saving:** `dump-namespace` saves (to a DOS file in the GCLISP .fas format) the exact contents of any namespace. This image can later be re-loaded into the GCLISP environment of the same session or any later session, saving you the time and trouble of running again the individual commands to define functions, load files, and resolve references.

Importantly, the .fas file can also be input to GCLRUN, to incorporate the namespace-packaged C and LISP code into a runtime application.

- **Cleanups:** `flush-namespace-info` serves to release the part of the GCLISP workspace occupied by a namespace, after you are done working in it. `fli-makunbound` frees up the space occupied by the interface software itself (the functions, macros, and variables described in this guide).
- **Namespace maintenance:** `do-namespace-fixups` enables the established links between all code and data in a namespace to be preserved if the namespace must be moved in the GCLISP workspace.

Operating Suggestions and Restrictions

Initializing and Accessing the Software

- **Initializing the interface:** In any GCLISP session, the first call to `define-foreign-call` or `define-foreign-entry` automatically loads all of the interface software. However, if you want to invoke any of the other interface functions first, then begin by loading the file `setup.lsp`, to make the interface available. `setup.lsp` is found in the directory `\gclisp3\lisplib\fli`, where all of the interface software resides.

- **Loading the compiler:** The first call to `define-foreign-call` automatically loads the GCLISP compiler (since the defined LISP function is compiled before `define-foreign-call` returns).
- **The `fli` package:** The home package of the interface software is the pre-defined package `fli`. Since all of the user-accessible software has been exported from the package, it can be accessed without using the `fli`: package qualifier.

Namespaces and Name Resolution

- **One language per namespace:** A given namespace should be used for foreign language functions from only one source language. If you have some functions written in Lattice C and some written in Microsoft C, don't load from both languages into the same namespace.
- **Scope of the name-resolution process:** When `resolve-unresolved-references` is called by `define-foreign-call`, the only namespace searched for name-resolution is the namespace specified in the `define-foreign-call` function call (where the foreign-language program is loaded). In general, if a foreign-language program **A** is to call another foreign-language program **B**, then **A** and **B** must be in the same namespace.

Miscellany: Calling Macros; Call Arguments; Debugging

- **Calling LISP macros from C:** `define-foreign-entry` can be used to declare either a function or a macro (defined by `defun` or `defmacro` respectively).
- **Number of arguments in function calls:** The interface software currently supports only calls to foreign language functions with a fixed number of arguments, not a variable number.

- **Argument type checking:** At program execution time, argument type checking can be performed (or disabled) for arguments passed to a C function from a LISP function (and return values from the C function to the LISP function). Argument type checking can't be done for calls from C to LISP.
- **Stack frames and debugging:** Execution of a foreign-language function creates a stack frame for the call. These frames are not visited by the debugger. (See the *Operating Guide* chapter "Debugging in GCLISP" for a description of the debugger.)

The Function-Definition Macros

Define-foreign-call

```
define-foreign-call lisp-function-name namespace object-file-pathnam  
entry-name &optional args-types-list return-value-type  
(compiler *default-foreign-compiler*) => t
```

The **define-foreign-call** macro generates and compiles a GCLISP function which, when executed, calls a program which was written and compiled in (Lattice or Microsoft) C.

The C program must have been compiled with certain compiler switches set (see the section "Detailed Requirements on the Foreign Code" in the first chapter in this guide).

The arguments to **define-foreign-call** have these meanings:

lisp-function-name This symbol names the GCLISP function which will be created to call a C function.

The call to **define-foreign-call** defines *lisp-function-name* in the current GCLISP environment. *lisp-function-name* can then be called like any other GCLISP function.

namespace The code of the C function which you wish to call will be loaded into the namespace named by the symbol *namespace*. This namespace need not exist at the time of this call to **define-foreign-call**.

- object-file-pathname* This string specifies the DOS pathname which names the .obj or .lib file which you want to load.
- entry-name* This string names the C function which you want to call. This function must be an entry point inside the .obj or .lib file specified in *object-file-pathname*.
- args-types-list* This optional parameter gives information about the formal parameters which the C function accepts.
- args-types-list* must be a list of ordered pairs. There is one pair for each of the C function's formal parameters. This list is enumerated in the same order as the formal parameters in the C function header.
- Each ordered pair in the list has the form (*name type*) (parentheses included).
- The first element of each ordered pair, *name*, specifies a GCLISP formal parameter of the GCLISP function called *lisp-function-name*. (An exception to this is the GCLISP formal parameter type **pointer**.) When the function *lisp-function-name* is called, the corresponding call argument of the call will be translated into the C parameter to which it corresponds.
- The second element of each ordered pair, *type*, specifies the C data type into which the GCLISP call argument will be translated.
- return-value-type* This optional parameter tells GCLISP the type of the entity which the C function returns. It must be the same as the C type which is returned by the C function *entry-name* in *object-file-pathname*.
- compiler* This optional parameter specifies the language which the foreign language program was compiled in.
- It must be either the word **:lattice**, for the Lattice C language, or **:ms-c**, for Microsoft C.

The notation (*compiler* ***default-foreign-compiler***) means that if the *compiler* parameter is not specified in the command, then the current value of the variable ***default-foreign-compiler*** is used. This variable is initialized to `:lattice` in the Developer initialization file `config.lsp`.

Note that if any optional parameter is to be specified in the command, then any optional parameters preceding it in the command-syntax displayed above must be specified (at least as `nil`).

See the section "Call-Argument and Return-Value Data Types" in the preceding chapter for a list of the parameter types currently supported. (Note there also how the specification of an argument or return value of type `pointer` differs from the description above.)

Examples of the use of `define-foreign-call` are in the chapter "Examples and Error Messages", and also in the section "An Example" below.

Define-foreign-entry

```
define-foreign-entry namespace lisp-target-fname internal-link-fname
  &optional args-types-list return-value-type
  (compiler *default-foreign-compiler*) => t
```

define-foreign-entry enables a foreign language routine which has been loaded into the LISP world to call a LISP function. Thus it is the inverse of **define-foreign-call**.

The **define-foreign-entry** call creates a link to the LISP function named *lisp-target-fname*. When this name is referenced during execution of a foreign language routine, the LISP function currently associated with the name will be called.

An example of the use of **define-foreign-entry** is in the next section.

The arguments to **define-foreign-entry** have these meanings:

namespace This symbol names the namespace where the *lisp-target-fname* is to be loaded. The effect of this argument is to enable any foreign-language routine in this namespace to call *lisp-target-fname*.

The namespace need not exist at the time of this call to *define-foreign-entry*; and it will not be created at this time if it does not already exist.

lisp-target-fname This symbol names the LISP function to be called.

The name must be defined as an external reference in the foreign language routine. For example, a C module from which the LISP function is to be called would include the declaration:

```
extern lisp-target-fname;
```

This function need not exist at the time of this *define-foreign-entry* call. The function must exist by the time it is called. (The result will otherwise be an "Undefined function" LISP error.)

internal-link-fname When *lisp-target-fname* is invoked during execution of a C routine, the function *internal-link-fname* (which will have been automatically compiled) will actually be called. (This is an internal implementation mechanism which concerns the user only during debugging, when this name may be printed out during examination of the execution stack or the bindings stack.)

The user need only supply a name (a symbol) here; since it will never be used directly in a user-written GCLISP statement, the exact name matters very little (so long as it is not already bound in the GCLISP environment to a needed value or a function definition).

args-types-list This has a role similar to *args-types-list* in **define-foreign-call**. (See the preceding description of **define-foreign-call**.) Here, however, the symbol *name* in the ordered pair (*name type*) refers to an actual argument which will be passed by the foreign language routine to *lisp-target-fname*, rather than to a *formal* parameter.

Note that if the *return-value-type* parameter is to be specified in the command, then the *args-types-list* parameter must be specified (as *nil*, the empty list) if the C function has no parameters.

return-value-type This is the data type which the foreign language routine expects to receive back from *lisp-target-fname*. It thus has a role similar to *return-value-type* in **define-foreign-call**.

compiler This optional parameter specifies the language which the foreign language program was compiled in.

It must be either the word **:lattice**, for the Lattice C language, or **:ms-c**, for Microsoft C.

The notation (*compiler* ***default-foreign-compiler***) means that if the *compiler* parameter is not specified in the command, then the current value of the variable ***default-foreign-compiler*** is used. This variable is

initialized to `:lattice` in the Developer initialization file `config.lsp`.

An Example

Assume that you have coded this Lattice C language routine, which echoes an input character, converting it to upper case if it is a letter:

```
int echo()
{ char in;
  in = '\0';
  while( in != '?' )
  {
    in = fgetc();
    if ( ('a' <= in) && (in <= 'z') ) in -= 'a' - 'A';
    fputc( in );
  }
}
```

This C routine, `echo`, calls two subroutines, `fgetc()` and `fputc()`, for input and output respectively. In C programming, these routines are normally found in a library of C subroutines. However, when `echo` is embedded inside a LISP environment, it must use LISP facilities to communicate with devices external to that LISP environment. So `fgetc()` and `fputc()` must be written in LISP, and `echo` must access them (and via them, the I/O devices) through links defined by `define-foreign-entry`. We show now how this is done, in several steps.

Step 1. These two declarations must be added to `echo`:

```
extern fputc();
extern char fgetc();
```

This tells the C compiler that the names `fputc` and `fgetc` are not defined in the C module containing `echo`. (We assume for simplicity that this function definition of `echo` comprises a complete C module also called `echo`.)

That is, occurrences of these names in that module are *unresolved references*. The C compiler can now compile the module `echo` into the object file `echo.obj`, without trying to resolve the occurrences of `fputc` and `fgetc`.

However, when `echo` is loaded into the LISP world, these references will have to be resolved for `echo` to run in that world.

Step 2. The rest of the work--steps 2 through 4 here--must be done in the LISP world.

fgetc and **fputc** are names which are *used* inside a C routine, but which *refer to* functions in the LISP world. So the LISP world must make the names available to C routines which want to call them, such as **echo**.

These references are constructed via **define-foreign-entry**. A call to **define-foreign-entry** defines a name which a foreign routine can mention.

These two calls to **define-foreign-entry** do the trick:

```
(define-foreign-entry sra fputc fputc-internal ((ch :char)) :char)
(define-foreign-entry sra fgetc fgetc-internal nil :char)
```

Step 3. Now **echo.obj** can be loaded into the LISP world by **define-foreign-call**:

```
(define-foreign-call echo sra "echo.obj" "echo")
```

During loading, the loader will register the two external declarations which appear in **echo.obj**. It will try to resolve these names: that is, it will try to link each of them to an external object or function with the same name. In general, such an external entity could be in another C routine; or it could be in the LISP world. In the present instance, we already know that the intent is to define LISP functions named **fgetc** and **fputc** to be linked to the names declared in the C function **echo**.

If the **define-foreign-call** call above which loads **echo.obj** were executed before the two calls to **define-foreign-entry** (in Step 2 above) which create references for **fputc** and **fgetc**, then **fgetc** and **fputc** would be flagged as unresolved references. They could then be resolved by executing the calls to **define-foreign-entry** and then re-running **define-foreign-call**. (Note that it is not enough to run **resolve-unresolved-references** after executing the calls to **define-foreign-entry**. The original call to **define-foreign-call** must be repeated.)

Thus, this Step 3 and the preceding Step 2 can be done in either order (with an extra call to **define-foreign-call** if Step 3 were done first).

Step 4. The final step is to define the LISP functions which the C routine is going to call.

Here are LISP definitions of **fputc** and **fgetc**:

```
(defun fputc( ch )
  (format t "~c" ch)
  (coerce ch 'string-char))
```

```
(defun fgetc()
  (setq ch (read-char))
  ch)
```

These LISP `defun` function definitions must happen before any C routine--such as `echo`--is executed which is going to call the functions thus defined. The reason is that if a name does not refer to anything in the LISP world, then it will not refer to anything when it is mentioned inside a C program running in the LISP world; and the result will be the equivalent of a LISP "Undefined variable" error. Since this is a LISP-world error, however (it is the LISP-world function which is undefined), the C routine will not be able to recover from it, and the LISP system will crash.

However, these definitions of `fgetc` and `fputc` may be modified interpretively in the LISP world, as often as the user wishes. All that must be preserved is the names of the arguments passed and the type of the value returned, since these attributes have been established by the `define-foreign-entry` calls.

Note that `fgetc` and `fputc` do *not* need to be defined before the executions of `define-foreign-entry` which declare them as names available to a foreign language routine. This is because `define-foreign-entry` simply makes a *name* available to a foreign language routine.

Thus, Step 2, Step 3, and this Step 4 may be done in any order, as long as these two rules are obeyed:

- Before the C function is executed (called), any external name used in the C function must be resolved (by a call to `define-foreign-entry`, or by loading a C function with that name).
- Any function referred to must exist before it is invoked by name: that is, before the C function which uses the function's name is executed. If the name names another C function, that function must have been loaded; if it names a LISP function, that function must have been defined. If a referenced function doesn't exist at execution time, an "Undefined function" error will result.

At this point, all the necessary steps have been completed. In the LISP world, a call to `echo` will invoke the LISP function of that

name, which will transfer control to the C routine of the same name, which, while executing, will call the two LISP functions `fgetc` and `fputc`. When either of these is exited, the C `echo` regains control. When this returns, the LISP `echo` regains control. And the LISP `echo` returns like any other LISP function.

Other Functions and Variables

Flush-namespace-info

`flush-namespace-info &rest namespaces => nil`

This function unbinds all objects defined in the specified namespaces. These objects are no longer needed after all references have been resolved and the namespaces have been dumped (if wanted). The effect is to free up space in your GCLISP workspace.

If *namespaces* is `nil`, all existing namespaces are flushed.

Fli-makunbound

`fli-makunbound => nil`

This function unbinds all of the internal functions and variables which comprise the Foreign Language Interface software.

It should be used only *after* all desired foreign-language object files have been loaded.

Global Variables Used by the Interface

These are the global variables used by the interface.

default-foreign-compiler

This specifies the language which a foreign language program was compiled in. Its value may be either `:lattice` or `:ms-c`, for Lattice C and Microsoft C respectively. It is initialized to `:lattice` in `config.lsp`. The value of the variable is used (and may be specified) in a call to the macro `define-foreign-call` or the macro `define-foreign-entry`.

foreign-type-checking-enabled

This specifies whether or not type checking should be performed when a function written in a foreign language is actually called and arguments are passed to it, and when a LISP function called from a foreign-language function passes its return value to the foreign-language function upon exiting. It is a boolean variable: its value should be either `t` or `nil`. It is initialized to `t` in `config.lsp`, meaning that type checking is enabled (should be done).

While foreign language interface functions are being debugged, checking should be enabled. After debugging, for faster execution at run time, type checking can be disabled by setting the variable to `nil`.

Note that whenever you change the value of this variable, the `define-foreign-call` call which created the foreign-language-calling code must be re-run to re-compile the code minus the type-checking code.

Note also when the foreign language is C, type checking can be performed only on arguments (or return values) passed from a LISP function to a C function, and not from C to LISP.

object-file-namespaces

When an `.obj` or `.lib` file is first loaded and linked into a namespace, the namespace is stored in this list.

object-data-segments

An object module may have multiple data segments, but only one code segment. This is a list of valid data segments.

objload-debug

During the loading of an `.obj` file or a `.lib` file, debugging messages are output to this stream if it is non-nil.

Chapter 26

Programming with the Interface

The preceding chapter, "Basic Usage", documented a basic useful subset of foreign-language-interface functions and variables.

By contrast, the sections of the current chapter document individually those functions included in the Foreign Language Interface which should be needed (and used) only by knowledgeable GCLISP programmers.

The Interface Functions

Load-object-file

```
load-object-file namespace object-file-pathname  
  &optional entry-list  
  &rest [[object-file-pathname [entry-list]] ...]  
  => nil
```

This function can be called by the user to load any object or library files that must be loaded to resolve external references. (**load-object-file** is already used (called) automatically by the **define-foreign-call** macro to load into the GCLISP world the object file or library file specified in a user-written call to that macro.)

The arguments to **load-object-file** have these meanings:

namespace This required argument is a quoted symbol, specifying the namespace into which the object file

is to be loaded. If a namespace of this name does not exist at the time of the call, it is created.

object-file-pathname

This required argument is a string specifying the DOS pathname of the object file to be loaded.

entry-list

This optional argument is a list of strings, naming the library entries which should be loaded when *object-file-pathname* names a library (.lib) file. *entry-list* is meaningless when *object-file-pathname* names an object (.obj) file, which is loaded as a single module.

object-file-pathname [*entry-list*]

A **&rest** argument specifies any desired additional file, and optionally the particular entries in it to be loaded. For all except the last-specified file, the *entry-list* is not optional; it must be specified (at least as nil).

The function returns t if at least one file is successfully loaded; or nil if all of the specified files were previously loaded.

Resolve-unresolved-references

resolve-unresolved-references &rest *namespace-list*
=> *nil/list*

This function is called when the user wants to cause all external references within the loaded modules to be resolved.

The optional list *namespace-list* specifies the set of namespaces in which to complete the resolution process. If *namespace-list* is unspecified or is nil, all namespaces are resolved.

The function returns nil if all references are resolved. Otherwise, information is printed out regarding each remaining unresolved reference, and the function returns a list of these unresolved references.

Do-namespace-fixups

do-namespace-fixups &rest *namespace-list* => *nil*

This function is called automatically by the function **resolve-unresolved-references** when there are no unresolved references. All fixups are kept with the namespace.

If a namespace is moved in memory, this function must be called to re-fixup the code and data segments.

The optional argument *namespace-list* is a list of namespaces in which to complete the fixup process. If *namespace-list* is unspecified or is *nil*, the fixups are performed on all namespaces.

Find-object-entry and Find-data-entry

find-object-entry *namespace object-file-pathname entry-name*
=> *segment-offset-address segment-base-address*

find-data-entry *namespace entry-name*
=> *segment-offset-address segment-base-address*

These functions are used primarily by the **define-foreign-call** macro to locate the address of the foreign-language code to execute. These functions may be used by the GCLISP programmer to access the code or data object directly.

namespace is a symbol naming the namespace where the object is to be sought. *object-file-pathname* and *entry-name* are strings.

The logical address of the entry is returned.

Dump-namespace

dump-namespace *namespace output-filename* => *nil*

This function may be called when the user is finished defining foreign calls and foreign entries, loading object files, and resolving references.

The purpose of this function is to preserve the current user-defined foreign-language-interface functions and their associated states.

The namespace itself--that is, the parsed object files and their associated states--is "fasdumped" to an output "fasl" file. The name of this output file will be the *output-filename* argument, with a "\$" prepended to the front of the filename and with the same file extension (or the file extension *fas*, if none is user-supplied.)

All of the linkage functions which were defined in this namespace by **define-foreign-call** or **define-foreign-entry** calls are written out to a second file. The name of this file is the *output-filename* argument with the file extension *lsp*.

This second file is then compiled and the compilation output written to a third file, *output-filename.fas*. This compiled file may be re-loaded, via a call to the load function, into a fresh GCLISP environment at any later time. The new environment may be another session of the GCLISP 386 Developer, or a session in the GCLISP 286 Developer or in GCLISP 1.1, or the GCLRUN environment.

This re-load will automatically load also the first file, *\$output-filename.ext*, which contains the namespace. The net result is that in the new environment, all of the needed linkages to foreign entries and foreign calls will be defined without re-executing the calls to **define-foreign-call** and **define-foreign-entry** which originally created them.

Note that the loading of this dumped namespace will invoke the loading of two of the Foreign Language Interface files, *objlink.fas* and *objfixup.fas*.

Chapter 27

Examples and Error Messages

Examples of the Define-foreign-call Macro

Suppose that we want to call three C functions which reside in the module pointed to by the pathname `C:\mycprogs\c1.obj`. These three functions are named `foo`, `bar`, and `utility18`.

For each of these three C functions, a LISP function will be created (by a `define-foreign-call` macro call) to establish a link to the C function. The namespace into which `c1.obj` is to be loaded will be called `c-region`.

Suppose that the three C functions `foo`, `bar`, and `utility18` are defined as follows:

```

/* foo accepts an int and returns another */
int foo( y )
int y;
{ return( y + 1 ); }

/* bar accepts an int and a float and returns a float */
float bar( i, f )
int i;
float f;
{ return( f * i ); }

/* utility18 calls a function at a supplied address */
typedef int FUNCT_RETURN_TYPE;

FUNCT_RETURN_TYPE utility18( plf )
FUNCT_RETURN_TYPE (*plf)();
{
    int x;
    x = (*plf)();
    return( x );
}

```

The `define-foreign-call` macro call which establishes a link for a call to `foo` can be written as follows:

```

(define-foreign-call
  my-lisp-foo          ; the <lisp-function-name> argument
  c-region            ; <namespace>
  "C:\mycprogs\cl.obj" ; <object-file-pathname>
  "foo"               ; <entry-name>
  ((a :int))          ; <args-types-list>
  :int                ; <return-value-type>
)

```

Here *args-types-list* is the list `((a :int))`, containing one ordered pair.

Now `my-lisp-foo` can be called, as for example in:

```
(setq x (my-lisp-foo 5))
```

The `define-foreign-call` macro call which establishes a link for calls to `bar` can be coded as follows:

```
(define-foreign-call
  jump-to-bar          ; the <lisp-function-name> argument
  c-region            ; <namespace>
  "C:\mycprogs\c1.obj" ; <object-file-pathname>
  "bar"              ; <entry-name>
  ((x :int) (y :float)) ; <args-types-list>
  :float             ; <return-value-type>
)
```

Here *args-types-list* is a list containing two ordered pairs.

Now `jump-to-bar` can be called, as for example in:

```
(equal (jump-to-bar 2 3.14159) (* 2 3.14159))
```

The `define-foreign-call` macro call which establishes a link for calls to `utility18` is coded as follows:

```
(define-foreign-call
  new-tool            ; <lisp-function-name>
  c-region           ; <namespace>
  "C:\mycprogs\c1.obj" ; <object-file-pathname>
  "utility18"        ; <entry-name>
  (((base off) :pointer)) ; <args-types-list>
  :int               ; <return-value-type>
)
```

Here *args-types-list* is a list containing one ordered pair. But the first element of this ordered pair is another ordered pair, `(base off)`, the address which `utility18` requests. The two parts of the address, namely `base` and `off`, are two LISP fixnums which will be concatenated to form a 4-byte C pointer.

Now `new-tool` can be called, as for example in:

```
(new-tool funct-seg funct-off)
```

Note that `new-tool` takes two call arguments. The first is the segment, and the second is the offset, of the desired address.

Error Messages

These error messages may be issued during running of the foreign language interface macros and functions.

The messages here are specific to the foreign language interface operations. Other error messages besides these may appear when you invoke these functions--for example, messages about syntax errors in your command input.

The messages here are grouped by the kind of operation or function execution which produces them, for example "From **define-foreign-call**" or "While resolving references." This does not match exactly the set of interface functions. For example, a call to **define-foreign-call** results in loading one or more object files, linking the loaded code in the GCLISP workspace, and using defined names to resolve references. So when **define-foreign-call** is invoked, error messages may appear from any of the categories "From **define-foreign-call**", "During object-file loading", "During linking", or "While resolving references." You may want to refer to the first section, "Interface Operations", in the chapter "Basic Usage" for a description of the logical sequence of operations performed by the interface functions.

From **define-foreign-call**:

Unsupported C compiler.

The value of the **compiler** parameter or of ***default-foreign-compiler*** did not name a recognized foreign-language compiler.

Unsupported C parameter type.

The specified argument type is not supported in this implementation.

Unsupported C return-value type.

The specified return-value type is not supported in this implementation.

From:**Unsupported C compiler.**

The value of the **compiler** parameter or of ***default-foreign-compiler*** did not name a recognized foreign-language compiler.

Unsupported C parameter type.

The specified argument type is not supported in this implementation.

LISP cannot return this type to C.

The specified return-value type is either unsupported or invalid.

During object-file loading:**Bad group component descriptor type: *type*.**

The object-file format is corrupted or non-standard, or this feature is unsupported.

Unknown segdef attribute: *value*.

The object-file format is corrupted or non-standard, or this feature is unsupported.

During linking:**t5 not yet implemented.**

This object-file-format feature is not supported.

t7 not yet implemented.

This object-file-format feature is not supported.

Unrecognized fixup target.

The object file is corrupted, or the object-file format is not supported.

***path/name* not found for fixup.**

The object file is corrupted.

Unrecognized fixup mode *n*.

Unknown type of fixup. This object-file format is not supported.

Unrecognized fixup loc *nnnn*.

The object file is corrupted, or the object-file format is not supported.

While resolving references:*namespace not a valid namespace.*

Use a valid namespace name.

entry-point unresolved reference.

This is issued from a call to **resolve-unresolved-references**. The call may have been issued directly to the GCLISP interpreter. Or it may have originated in a direct call to **define-foreign-call**.

The message means that for some entry point referred to (as a function call) in a C function which has been loaded into the namespace or namespaces where you are attempting to resolve references, there is no defined function name to resolve the reference to. Either a GCLISP function name or a C function name must be supplied.

To fix the error:

- If the entry point was intended to refer to a C function: use **load-object-file** to load into the namespace the C-language **.obj** or **.lib** file including the needed function definition.
- If the entry point was intended to refer to a GCLISP function: use **define-foreign-entry** to specify the name of the needed GCLISP function.

In both cases:

- If the message was issued while **define-foreign-call** was running, then you must also re-run the same **define-foreign-call** command after fixing the error.
- Otherwise the message was issued during the running of a direct call to **resolve-unresolved-references**. Then you must re-run the same **resolve-unresolved-references** command after fixing the error.

path not found in namespace namespace.

find-object-entry was called with an incorrect pathname for an object file in the namespace *namespace*.

entry not found in namespace name/object-file path.

The entry being sought by **find-object-entry** was not found in the specified object file in the specified namespace.

This message may be issued during the running of **load-object-file** or **define-foreign-call**. If the source was a direct call to **load-object-file**, the cause may have been an invalid setting of the global variable ***default-foreign-compiler***.

At runtime:

Bignum too large to fit in C long.

The integer argument being passed to a C function was longer than the representation of a long in C.

C type and type of LISP entity are incompatible.

This error message issues from the type checker, when a LISP object is being passed as an argument to a C function. If type checking is disabled, the error will not be detected and the message will not appear. However, the results of this condition without the type checking are unpredictable and could be fatal.

C cannot accept the type returned by LISP.

This error message issues from the type checker, when a LISP function called from a C function returns and passes its return value to the C function. If type checking is disabled, the error will not be detected and the message will not appear. However, the results of this condition without the type checking are unpredictable and could be fatal.

GCLISP 386
Index

Index

Symbols

" 48
%contents 198
%contents-store 198
%sysint 189
' 48
(48
(break) 40
(Break to listener) 162
(clean-up-error) 39
(configure-gclisp) 13
(continue) 40
(exit) 13
(sys:dos) 14
) 48
command-line 11
default-foreign-compiler 239,
242
MONITOR-IS-COLOR 9
.exe file 228
.fas file 227, 236, 252
.lib file 228, 250
.obj file 228, 250
84H -- Get Segment Alias 201
88h--Define Segment 196
89h--Undefine Segment 197
90h -- Memory Copy (MCPY)
199
91h--Low Memory Data Segment
Alloc 197
92h--LM Data Segment Free 198
93h--Real Mode Interrupt 200
: 48
:: 48
:asm-file 208

:error-file 207
:lap-file 207
:load 208
:output-file 207
; 48
\ 48
\gclisp3 11
| 48

A

ADD-CHANGE-LOG-ENTRY
154, 171
Address, Gold Hill 17
Allocating and Accessing low
memory 196
alphanumeric 105
Alt-E 12
Alt-H 12
Alternate key 23
APPEND-NEXT-KILL 160, 171
apropos - function 54, 55, 58
apropos - help option 54
arglist 55, 57
attribute-list 93
auto-fill 92, 100, 109, 142, 171
AUTO-FILL-COMMENTS 100,
154, 171
autoexec.bat 9

B

BACK-TO-INDENTATION 142,
171
backslash 45, 48

BACKWARD-CHAR 139, 171
 BACKWARD-COPY-LINE 142, 171
 BACKWARD-KILL-LINE 142, 171
 BACKWARD-KILL-SEXP 154, 171
 BACKWARD-KILL-WORD 142, 171
 BACKWARD-LIST 124, 154, 171
 BACKWARD-LOWERCASE-WORD 153, 171
 BACKWARD-MARK-SEXP 154, 172
 BACKWARD-PAGE 142, 172
 BACKWARD-PARAGRAPH 142, 172
 BACKWARD-SEXP 124, 155, 172
 BACKWARD-UP-LIST 124, 155, 172
 BACKWARD-UPPERCASE-INITIAL 153, 172
 BACKWARD-UPPERCASE-WORD 153, 172
 BACKWARD-WORD 139, 172
 Base memory 187
 Batch file 11
 BEGINNING-OF-BUFFER 139, 172
 BEGINNING-OF-DEFINITION 125, 155, 172
 BEGINNING-OF-NUMBERED-LINE 139, 172
 BEGINNING-OF-SENTENCE 142, 172
 BIND-KEYBOARD-MACRO 167, 172
 binding keys 131
 BIOS 187
 BIOS-Compatible Services 192
 BIOS interrupt services 192
 bound 33
 bound (to key) 90
 break 61, 129
 (break) 40
 break - function 61
 Break key 24
 break level 64
 break message 63

BUFFED 100
 BUFFER-EDIT 100, 146, 172
 buffer-edit mode 97
 BUFFER-READ-ONLY 100, 147, 172
 BUFFER-READ-WRITE 100, 147, 172
 buffer-status 89
 buffername 97

C

call argument 228, 242, 247, 259
 case - upper
 lower 112
 CHANGE-DIRECTORY 101, 147, 172
 (clean-up-error) 39
 clean-up-error - function 62
 Co-resident DOS programs 186
 code segment 248
 colon 48
 Color monitor 9
 command-line 11
 command completion 91
 Command line options 11
 COMPAQ monochrome monitor 9
 COMPILE-AND-EXIT 103, 155, 172
 COMPILE-BUFFER 99, 155, 172
 COMPILE-DEFINITION 155, 172
 config.hb 9
 config.lsp 7, 9, 13, 240, 243, 247
 (configure-gclisp) 13
 %contents 198
 %contents-store 198
 (continue) 40
 continue - function 63, 64, 65, 71
 Control bit 91
 Control key 22
 COPY-LINE 143, 173
 COPY-REGION 160, 173
 COPY-TO-REGISTER 160
 COPY TO REGISTER 173
 Ctrl-Break 40
 Ctrl-C 41
 Ctrl-D 15

Ctrl-E 12
Ctrl-End 138
Ctrl-G 39
Ctrl-Home 138
Ctrl-Left Arrow 138
Ctrl-P 40
Ctrl-Pg Dn 138
Ctrl-Pg Up 138
Ctrl-Right Arrow 138
current item 123
current window 122
Cursor 21, 85
cursor-control keys 24

D

damaged components 18
data segment 231, 248
data type 229
debugging 61, 238
default settings 99, 100
define-foreign-call 229, 234, 238,
244, 253, 256
define-foreign-entry 229, 235,
241, 244, 257
DEFINE-KEYBOARD-MACRO
167, 173
DEFSEG 196
defun 35
Del 138
DELETE-BLANK-LINES 143,
173
DELETE-CHAR 143, 173
DELETE-HORIZONTAL-SPACE
143, 173
DELETE-INDENTATION 143,
173
DIRECTORY-EDIT 101, 147,
173
directory-edit mode 97
DIRED 100
DISPLAY-APROPOS 127, 162,
173
DISPLAY-DIRECTORY 101,
148, 173
DISPLAY-DOCUMENTATION
127, 162, 173

DISPLAY-KILL-HISTORY 160,
173
DISPLAY-LAMBDA-LIST 127,
155, 173
DISPLAY-MACROEXPANSION
127, 155, 173
display device 9
DO-IT-AGAIN 92, 162, 173
do-namespace-fixups 236, 250
doc - function 54, 58
doc - help option 54
DOS 102, 187
(sys:dos) 14
DOS-Compatible INT-21 Services
190
DOS - operating system 13
double colon 48
double quote 48
DOWN-LIST 124, 156, 173
Down Arrow 138
DUMP-BINARY-FILE 99, 148,
173
dump-namespace 236, 251

E

echo window 91
ed 87
ED-APROPOS 162, 174
ED-BEEP 96, 163, 174
ED-DOC 163, 174
ED-HELP 163, 174
ED-KEYCHORD 163, 174
ED-LISTBACK 163, 174
edit buffer 86, 87
edit command 90
editor 44
edit screen 89
edit window 85
edit window - commands 140
edit window - current 122
End 138
END-DEFINE-KEYBOARD-
MACRO 168, 174
END-OF-BUFFER 139, 174
END-OF-DEFINITION 125, 156,
174

END-OF-NUMBERED-LINE
 139, 174
END-OF-SENTENCE 174
Enter 21
Enter key 22
entry point 239, 250, 258
error 39, 41
error level 62, 64
error message 43
error messages 41
Escape key 23
EVAL-AND-EXIT 103, 156, 174
EVAL-BUFFER 99, 156, 174
EVAL-DEFINITION 156, 174
eval - function 31
EVAL-IN-MINIBUFFER 156,
 174
EVAL-SEXP 156, 174
evaluation 32
evaluator 31
**EXCHANGE-POINT-AND-
 MARK** 161, 164, 174
EXECUTE-DOS-COMMAND
 103, 148, 175
**EXECUTE-KEYBOARD-
 MACRO** 168, 175
(exit) 13
EXIT-EDITOR 148, 175
EXTENDED-COMMAND 163,
 175
extended command 86
Extended memory 187
external reference 234, 241, 243,
 249

F

F1 138
far call 230
filename 97
file system 189
FILL-PARAGRAPH 143, 175
filling paragraphs 109
find-data-entry 251
FIND-FILE 87, 97, 98, 148, 175
find-object-entry 251
FIND-UNBALANCED-PARENS
 125, 156, 175

fli-makunbound 236, 246
fli directory 236
fli package 237
flush-namespace-info 236, 246
foreign call 227, 231
foreign entry 227, 231
format directive 63
FORWARD-CHAR 139, 175
FORWARD-ISEARCH 119, 151,
 175
FORWARD-LIST 124, 156, 175
FORWARD-PAGE 143, 175
FORWARD-PARAGRAPH 143,
 175
FORWARD-SEARCH 119, 152,
 175
FORWARD-SEXP 124, 157, 175
FORWARD-UP-LIST 124, 157,
 175
FORWARD-WORD 139, 175
function call 42
function keys 24, 91

G

garbage collection 12
GC 12
GCLISP.BAT 11
GCLISP compiler 227, 236
gclisplm 11, 15
GCLRUN 227, 252
GMACS 44
GMACS editor 12
graphics functions 79

H

harddisk.bat 7
harddisk1.bat 7
help - on-line 29, 53
High Memory 187
Home 138

I

incremental search 119

INDENT-FOR-COMMENT 126,
157, 175

INDENT-LISP-LINE 126, 158,
175

INDENT-RIGIDLY 144, 176

INDENT-SEXP 126, 158, 176

INDENT-TEXT-LINE 144, 176

indenting LISP 126

Initialization 21

Ins 138

INSERT-BINARY-FILE 148,
176

INSERT-COMMAND-NAME 144,
176

INSERT-DOS-OUTPUT 103, 148,
176

INSERT-FILE 149, 176

INSERT-REGISTER 161, 176

INT-21 190

INT 12h (Memory Size) 192

interpreter 27

interrupt request 189

J

JUST-ONE-SPACE 144, 176

K

keyboard 22

keyboard macro 134

keychord 12, 15, 90

keys, special 22

key sequence 90

KILL-BUFFER 99, 149, 176

KILL-COMMENT 158, 176

KILL-LINE 144, 176

KILL-REGION 144, 161, 176

KILL-SEXP 158, 176

KILL-WORD 144, 176

kill history 86, 115, 122, 127

killing text 115

kill ring 91

L

lambda-list 59

lambda-list - function 54, 59

lambda list - help option 54

language conventions 48

large-model compilation 230

Lattice C compiler 230

Lattice C requirements 230

Left Arrow 138

LISP 27

LISP-INTERACT 157, 176

LISP-INTERACTION 100, 157,
176

LISP-interaction mode 129

LISP-MODE 99, 157, 177

LISP object 31

LIST-BUFFERS 99, 149, 177

listener 31

listener level 39, 62

list processing 27

LMALLOC 197

LMFREE 198

LOAD-KEYBOARD-MACROS
168, 177

load-object-file 249

loading files 45

loading GMACS 87

LOWERCASE-REGION 153, 177

LOWERCASE-WORD 153, 177

Low Memory 187, 198

M

macros 237

MAIN 87

major mode 92, 99

MAKE-MATCHING-() 125, 158,
177

mark 113

MARK-BEGINNING-OF-
BUFFER 164, 177

mark - current 114
 MARK-END-OF-BUFFER 164, 177
 MARK-PAGE 164, 177
 MARK-SEXP 164, 177
 MARK-WHOLE-BUFFER 164, 177
 mark pdl 114, 122
 MCOPY 199
 Meta bit 91
 Microsoft C compiler 230
 Microsoft C requirements 230
 Microsoft mouse 9
 minibuffer 86, 91
 minor modes 93, 100
 missing components 18
 mode line 89
 modes 92
 modified flag 99, 100
 MONITOR 9
 MONITOR-IS-COLOR 9
 mouse support 9
 Mouse Systems mouse 9
 MOVE-SCREEN-OTHER-WINDOW 122, 140, 177

N

NAME-KEYBOARD-MACRO 168, 177
 namespace 231, 234, 241
 NEWLINE 144, 177
 NEWLINE-INDENT 126, 158, 177
 NEXT-COMMENT-LINE 158, 177
 NEXT-LINE 139, 177
 NORMAL-MODE 99, 149, 177
 null pathname 98
 NUMERIC-ARG-PREFIX 163, 178
 numeric argument 108
 Numeric Lock key 24

O

object file 231

ONE-WINDOW 122, 140, 178
 OPEN-INDENTED-LINE 126, 158, 178
 OPEN-LINE 144, 178
 OTHER-WINDOW 122, 140, 178

P

P-KERNEL 186
 package 100
 package contents 4
 parameter type 228, 239, 242
 Paren-beep feature 126
 Paren-flash feature 125
 parentheses 48
 Parentheses keys 23
 PARSE-ATTRIBUTE-LIST 93, 159, 178
 Path command 12
 pathname 45, 89, 148
 Pg Dn 138
 Pg Up 138
 Phone number ((617) 492-2071) 17
 point 85, 107
 pointer type 229, 255
 pprint - function 72
 pprint - property 73
 prepend 116
 PREVIOUS-COMMENT-LINE 159, 178
 PREVIOUS-LINE 139, 178
 print - function 31
 print-name 56
 Print-Screen Key 23
 problems 17
 Prompt 21
 Protected Mode 187
 push-down list 114
 PUSH-TO-DOS 103, 149, 178

Q

QUERY-REPLACE 120, 152, 178
 QUOTED-INSERT 145, 178
 quote marks 48

R

R-KERNEL 186
 read-eval-print 31, 45
READ-FILE 98, 149, 178
 read - function 31
 read/write status 99, 100, 101
 reader 31
REALINT 200
REALINT Request 200
Real Mode 187
RECENTER-DEFINITION 159, 178
RECENTER-POINT 121, 140, 178
RECENTER-WINDOW 121, 141, 179
 recursive edit 120
 regions 113
REMOVE-SURROUNDING-() 159, 179
 repeat count 108
REPLACE-STRING 120, 152, 179
 Replacement Order Card 18
resolve-unresolved-references 235, 237, 250, 258
REVERSE-ISEARCH 152, 179
REVERSE-SEARCH 119, 152, 179
Right Arrow 138
RUBOUT 145, 179
RUBOUT-HACKING-TABS 145, 179
 Rubout key 22

S

San Marco LISP Explorer 12
SAVE-ALL-FILES 149, 179
SAVE-FILE 98, 149, 179
SAVE-FILES-EXIT 149, 179
SAVE-FILES-PUSH-TO-DOS 103, 150, 179
SAVE-KEYBOARD-MACROS 169, 179

SCROLL-DOWN 121, 141, 179
 Scroll-Lock/Break key 24
SCROLL-OTHER-WINDOW 122, 141, 179
SCROLL-SCREEN-DOWN 121, 140, 179
SCROLL-SCREEN-UP 121, 140, 179
SCROLL-UP 121, 141, 179
SELECT-BUFFER 99, 150, 179
SELECT-PREVIOUS-BUFFER 99, 150, 179
 self-evaluating form 32
 self-inserting input 108
 semi-colon 48
SET-BUFFER-PACKAGE 100, 150, 159, 180
SET-COMMENT-COLUMN 159, 180
SET-FILL-COLUMN 145, 180
SET-FILL-PREFIX 145, 180
SET-MODE 93, 99, 150, 180
SET-POP-MARK 161, 164, 180
SET-VARIABLE 92, 100, 150, 180
setf 33
setup.lsp 236
SGALIAS 201
SHOW-POSITION 164, 180
SHOW-REGISTERS 161, 180
SHOW-VERSION 164, 180
 single quote 48
 starting the Developer 11
 step - function 66
 step - options 67, 70
 symbol 33
sys:%sysint 227
sys:backtrace 64
sys:command-line 11
(sys:dos) 14
sys:exec 228
%sysint 189
 system requirements 3

T

TAB-TO-TAB-STOP 145, 180
TAGS-ADD-FILE 165, 180

TAGS-ADD-FILES 165, 180
 TAGS-CONTINUE-MAP 165, 180
 TAGS-FIND-ALL 165, 180
 TAGS-FIND-DEFINITION 165,
 180
 TAGS-INDEX-FILE 166, 180
 TAGS-LOAD-INDEX 166, 180
 TAGS-LOAD-TABLE 166, 180
 TAGS-MAKE-INDEX 166, 180
 TAGS-QUERY-REPLACE 166,
 181
 TAGS-REMOVE-FILE 166, 181
 TAGS-REPLACE-STRING 167,
 181
 TAGS-SAVE-TABLE 167, 181
 TAGS-SEARCH 167, 181
 TAGS-SHOW-TABLE 167, 181
 TAGS-USE-TABLE 167, 181
 tag table 103
 Technical support service 17
 Telephone number ((617)
 492-2071) 17
 template 75
 Terminate-and-stay-resident
 programs 12
 Terminology 21
 TEXT-MODE 99, 145, 181
 Top-Level 39
 trace - function 65
 TRANSPOSE-CHARACTERS
 118, 146, 181
 TRANSPOSE-LINES 118, 146,
 181
 TRANSPOSE-REGIONS 118,
 161, 181
 TRANSPOSE-SEXPS 159, 181
 TRANSPOSE-WORDS 118, 146,
 181
 trashcan directory 102
 troubleshooting 18
 TWO-WINDOWS 121, 141, 181
 type 55
 type-out window 95
 type checking 237, 247, 259

U

UNBIND-KEYBOARD-MACRO
 181
 UNBIND-KEYBOARD-MACROS
 169
 UNDEFSEG 197
 undelete 102
 uninst.bat 7
 UNLOAD-BUFFER 99, 151, 181
 UNMODIFY-BUFFER 100, 151,
 181
 unresolved reference 243, 250,
 258
 UNTABIFY-REGION 146, 181
 untrace - function 66
 Up Arrow 138
 UPDATE-ATTRIBUTE-LIST 93,
 160, 181
 UPDATE-MODE-ATTRIBUTE
 93, 160, 181
 UPDATE-PACKAGE-
 ATTRIBUTE 93, 160, 181
 UPPERCASE-INITIAL 153, 181
 UPPERCASE-REGION 153, 181
 UPPERCASE-WORD 154, 182
 userinit.lsp 13

V

variable 33
 vertical bars 48
 VIEW-FILE 151, 182

W

white space 33, 105
 WINDOW-BACKWARD-PAGE
 121, 141, 182

WINDOW-FORWARD-PAGE 121, 141, 182
YANK-POP 162, 182
working directory 101
wrapped line 106
WRITE-FILE 151, 182

Y

YANK 162, 182