LIGHTSPEEDC™ USER'S MANUAL

THINK
TECHNOLOGIES, INC.

THINK
Technologies, Inc.

420 Bedford Street Suite 350
Lexington, MA 02173

MATH LIBRARY ERRATA

If you use the LightspeedC library "math", you may encounter a problem with the functions exp(), pow(), tan(), sinh(), or cosh().

The problem is that the Standard Apple Numerics Environment (SANE) does not reset the error flags when it is called. If the SANE overflow error flag happens to be set for any reason, **calls to any of exp(), pow(), and sinh() will refer to this flag and also return an error when there is no error.** A call such as log(0.0) is one which sets the SANE overflow error flag and creates this situation. A similar condition is possible for the tangent function tan(), which uses the SANE invalid operand flag. cosh() is affected because it uses the exp() function.

CORRECTION

The following describes changes to be made to the math.c file, the source module for the library called "math". Using LightspeedC, create a new project that contains only the file "math.c", which is included on the library disk "LS2.Libraries".

Edit the C functions exp(), pow(), and sinh() to add the following statement as the **first statement** of each of these functions:

```
SANEglobalenv.overflow = 0;
```

A similar situation exists for the tan() function. Add the following statement as the **first statement** of the tan() function:

```
SANEglobalenv.invalid = 0;
```

The source file math.c is now corrected. Save it, compile it, and then select **Build Library** from the Project menu. Name the library "math" (or something else if you prefer such as "new math"). You should now use this library in place of the original math library.

LightspeedC ™
Release 2.01 Supplement

THINK

Contents

LIGHTSPEEDC™ RELEASE 2.01 SUPPLEMENT

1 Introduction 1
- What to Read First 1
- A Note on Converting Old Projects 2
- A Note About Customer Support 2

2 The *LightspeedC* Environment 3
- Handling of HFS 3
- "Use Disk" Enhancements 4
- Using a Separate Resource File 4
- Setting the Application Creator 5
- More "Check Link" Information 5
- Typing in the Project Window 5
- "Remove Objects" Command 6
- Options Dialog 6
- "More Memory" Option 6
- Opening a Source File From the Finder 6
- RelConv Enhancements 6

3 The Editor 7
- Windows Menu 7
- Find Definition of Symbol 8
- Pop-up Include-File Menu 8
- Undo 8
- "Balance" Command 9
- Keyboard Usage 9
- Default Font/Tab Settings 9
- Search and Replace 10

4 The Compiler 14
- Code Generation 14
- Function Prototypes 15
- "Require Prototypes" Option 16
- "Check Pointer Types" Option 16
- The Inline Assembler 17
- Syntax for Absolute Variables 20
- Redeclaring Built-In Functions 21

5 Running with Switcher 22

6 Drivers and Code Resources 24
Running a Desk Accessory 24
Global Data Area in Drivers 24
Global Data Area in Code Resources 25
"Opening" an Open Driver 26
Open/Close Result Codes in Drivers 27
Runtime Routines in Code Resources 27
Additional Header Information for Drivers and Code Resources 27

7 Libraries 28
128K ROM and HFS Support 28
Other Macintosh Library Changes 28
Enhancements to Standard C Libraries 29
New Standard C Library Functions 30

8 Tips from Technical Support 45
Recommended Disk Layouts 46
"hello, world" in LightspeedC 47
Some Common Problems 48
Some Useful Tips 52
Some Grep Examples 57
Further Reading 58

APPENDIX

A Corrections to the *User's Manual* 59

1
Introduction

Welcome to the second release of LightspeedC, version 2.01. Many new features have been added since the original release, so please read this document carefully to familiarize yourself with them. This release also fixes all known bugs.

LightspeedC has grown as a result of the enhancements in this release. Approximately 28K more disk space is used. There is approximately 11K less available memory in the LightspeedC development environment. (There is, however, an option to obtain more memory at the expense of longer compile times. Read about the **More Memory** option on page 16 of this supplement.)

This supplement documents features which are new to this LightspeedC release. The original release of LightspeedC, version 1.02, was accompanied by a short supplement documenting features of the product that, for one reason or another, were not adequately described in the *User's Manual*. That material has been incorporated into the present supplement. To allow you to identify the older material, it is indicated by the marker **v1.02** immediately prior to the section to which it applies. Information so marked is not new to this release, but it does not appear in the *User's Manual*.

Note: We are no longer shipping a System Folder with LightspeedC. Use the System, Finder, and Font/DA Mover supplied with your Macintosh.

What to Read First

This is a large document, so we'd like to call your attention to certain portions that we especially hope you won't miss.

If you have been using the original release of LightspeedC, be sure to read "A Note on Converting Old Projects", below. If you've ever dreamed of an even faster LightspeedC, read the chapter "Running with Switcher".

If you are new to LightspeedC, the section "'hello, world' in LightspeedC", on page 47, can get you started in a hurry. You may also find the section "Recommended Disk Layouts", on page 46, helpful.

If you are using the Hierarchical File System (HFS), be sure to read the section "Handling of HFS" on page 3.

Everyone should be aware of the section "Using a Separate Resource File", on page 4. This section replaces Chapter 8 of the *User's Manual*. An appendix to this supplement contains other errata to the manual.

A Note on Converting Old Projects

This release (version 2.01) of LightspeedC uses a slightly different project document format than the previous release (version 1.02). When you open an old project, LightspeedC will automatically convert it to the new format, after asking you for confirmation.

If you bring up the **Make... (⌘M)** dialog after the conversion, you will see that all the source files are checked, indicating they need to be recompiled. Actually, the code generated the last time each source file was compiled is still intact. However, include-file information stored in the project for use by the Auto-Make facility is lost as a result of the conversion process. To regenerate this information, all source files must be recompiled. If the project is one which you obtained from a third party, you might not have all the sources! In this case, simply use **Make... (⌘M)** to uncheck the files that can't be recompiled.

If your project uses libraries, such as *MacTraps* or *stdio*, that have changed with this release, they are not automatically reloaded. Reload them manually or bring up the **Make... (⌘M)** dialog and select **Use Disk**, then **OK**.

Projects created by LightspeedC version 1.02 can be used as libraries (i.e. using **Add...**) without requiring conversion. However, LightspeedC version 1.02 cannot open a project created (or converted) by LightspeedC version 2.01; the message "`unknown error, ID=-192`" is generated.

A Note About Customer Support

A large portion of our technical support calls regard a few common problems. On page 48 of this supplement is a section called "Some Common Problems" describing these problems and their solutions. Before calling our customer support number, please check this section for possible solutions to your problem.

2
The *LightspeedC* Environment

Handling of HFS

LightspeedC treats the entire subtree rooted at the project directory as though it were one large "flat" folder. The project directory is defined as the directory containing the project document, but all files in subdirectories, sub-subdirectories, etc., of the project directory are treated as if their files were in the project directory itself. Similarly, files are treated as if they are in the LightspeedC directory if they are in the directory containing the LightspeedC application, or in any subdirectory, sub-subdirectory, etc., of that directory.

This method of handling folders allows you to organize your files into subdirectories as you see fit, without having to use absolute (or even relative) pathnames to refer to them. LightspeedC automatically searches the project and LightspeedC trees as necessary to find a file. However, there are some consequences of this scheme that you need to be aware of:

- Just as you cannot have two files with the same name in different "flat" folders on an MFS volume, you should not have duplicate file names in different subdirectories of the project (or LightspeedC) tree. If you do, LightspeedC will get confused about which of several files with the same name it should use. This will not lead to any explicit errors, but the "wrong" file may, in fact, used. (Be aware that **Save As...** copies files, rather than merely moving them, and so if you use **Save As...** to move a file from one folder to another without changing its name, be sure to delete the original one.)

  If you want to "shield" a directory from the tree search, enclose its name in parentheses. For example, you might have a subdirectory of the project directory named *(Backups)*. The files in this directory (and in any subdirectories, sub-subdirectories, etc.) will not be treated as part of the project tree, and may have names which are the same as files elsewhere in the tree.

  It's OK to have the same file name in both the project and LightspeedC trees: the conflict is resolved deterministically by search order.

- It is unwise to place a project directory within the LightspeedC tree, since it will be searched whenever the LightspeedC tree is searched, even when you are working on a different project. Some people like to organize all their LightspeedC work together, so they make each project directory a subdirectory of the LightspeedC directory. This is not a good idea under HFS. If you want to organize your work this way, put the LightspeedC directory and your project directories together in the same master directory.

- Searching the trees can be time-consuming, but LightspeedC remembers where it last found each file and looks there first the next time. So, when you are initially building a project, or if you've moved files around within the tree, compilations might be slower than usual at first. Don't worry. Once LightspeedC has learned where all your files are, it will speed up again.

Note that the tree search performed to achieve this emulation of "flat" folders is different from, and independent of, the search which is performed by the preprocessor to find include-files. As always, the search order for include-files (if not specified by absolute pathname) is: (1) referencing directory (that is, the directory containing the file in which the `#include` statement appears), (2) project directory, (3) LightspeedC directory. For simple filenames (i.e. *foo.h*, not *:dir:foo.h*), the entire project and LightspeedC trees are potentially searched; if the referencing directory falls within the LightspeedC tree, the LightspeedC tree is searched before the project tree.

A file may be moved freely within a tree: the tree search will automatically find it. If you want to move a file from the project tree to the LightspeedC tree, or vice versa, select **Use Disk** (in the **Make...** (⌘M) dialog) to get LightspeedC to notice the change.

Your project may contain files which are outside either tree; they are known by absolute pathname. Use **Get Info** (⌘I) to find out whether a file lives in the project tree, the LightspeedC tree, or outside. If you want to move a file known by absolute pathname into one of the trees, you will have to either **Remove** it and **Add** it back to get LightspeedC to notice the change, or open the file and **Save As...** a file in the desired tree.

The new scheme is designed to allow projects to be easily moved from one machine to another. When you move a project to a new machine, it is strongly recommended that you select **Use Disk** to let LightspeedC find all the files. Files need not be in the "same" places on the two machines as long as they are within the project or LightspeedC trees. Files outside either tree must have the same absolute pathname on the two machines.

The separate resource file associated with each project (*projectName.rsrc*) must appear in the actual project directory, not in any subdirectory: it is not searched for. Similarly, the DAShell application must reside in the actual LightspeedC directory.

"Use Disk" Enhancements

Use Disk (in the **Make... (⌘M)** dialog) now displays its progress. You can abort it by typing ⌘. (command-period), although the next **Use Disk** will start again at the beginning.

If a source file or library is not found by **Use Disk**, its make status (i.e. whether it is checked in the **Make...** box) is unchanged. As before, however, if a source file is found but one of the files it includes is not found, the source file is marked as needing to be made (i.e. it is checked).

v1.02

Using a Separate Resource File

Chapter 8 and page 3-13 (paragraph 3) of the *LightspeedC User's Manual* describe how to access resources from a separate resource file during development, then later merge the resources into the file built by **Build Application...**. Since the *User's Manual* went to press, LightspeedC has been enhanced and this process is now largely automatic.

You still need to place any resources your program needs in a resource file; you can use RMaker, ResEdit, or any other appropriate method to create this file. But, provided you obey a simple naming convention, your resource file will automatically be opened when you run your application under

LightspeedC, and your resources will automatically be copied into the stand-alone file you eventually build.

You should give your resource file a name consisting of the name of the project with the extension *.rsrc* appended. If your project document is titled *MyProg*, your resource file should be titled *MyProg.rsrc*. The *.rsrc* extension does not replace any existing extension: if your project document is named *MyProg.proj*, your resource file should be named *MyProg.proj.rsrc*. Furthermore, the resource file must be kept on the same volume—and under the Hierarchical File System (HFS), in the same folder—as the project document.

When you issue the **Run** command from LightspeedC's **Project** menu, your resource file will be opened automatically. You do not need to call `OpenResFile` from your program as described in Chapter 8.

When you issue one of the **Build Application...**, **Build Desk Accessory...**, **Build Device Driver...**, or **Build Code Resource...** commands, resources from your resource file will automatically be incorporated into the application, driver, or code resource file being built. You do not need to use an RMaker script or any other method to merge resource files as described in Chapter 8.

Setting the Application Creator

Set the creator by filling in the **Creator** field in the **Set Project Type...** dialog. If the field is set to `????` (the default), and the application being built will overwrite an existing application, then the creator is left unchanged.

More "Check Link" Information

The **Check Link** command indicates, in the Link Errors window, the files in which each undefined symbol is referenced and in which each multiply defined symbol is defined.

When the Link Errors window comes up as the result of an attempt to **Run (⌘R)** or **Build...**, this additional information is not displayed. Some disk activity is required to compute the information, so it is only displayed when you specifically request **Check Link**.

Typing in the Project Window

You can select a file in the project window by typing the first portion of its name, much as you can in Standard File. Since the files in the project window are not all in alphabetical order, the selected file is simply the first file in the project which matches the characters typed so far. If no file matches, any selected file is deselected (this is different from Standard File). There is a timeout feature just as in Standard File; if sufficient time elapses between keystrokes, a new sequence is begun. The tab key can be used to cycle among multiple files that match the characters typed so far.

The up-arrow and down-arrow cursor keys may also be used to move around in the project window. The return and enter keys may be used to open the selected file, just like double-clicking.

"Remove Objects" Command

This command, found in the **Project** menu, reverses the effects of all previous compilations and loading of libraries. The project document is "dehydrated"; it can be "reconstituted" by recompiling all source files and reloading all libraries.

Use this command when you need to make the project document as small as possible, e.g. for archiving or for transmitting to someone else.

Options Dialog

The **Options** menu is gone; in its place is a dialog displayed by selecting **Options...** from the **Edit** menu. There are several new options—**Check Pointer Types**, **Require Prototypes**, and **More Memory**—which are described in the relevant sections in this document.

"More Memory" Option

This option is initially unchecked. When checked, LightspeedC tries to find more memory during compilation by discarding certain data structures. This incurs some additional disk activity, since (1) the data structures will need to be read back in when compilation is complete, and (2) if they are "dirty", the data structures must be written out prior to compilation in case the compiler runs out of memory.

Leave this option unchecked unless you are developing a large project on a small machine which keeps running out of memory.

Opening a Source File from the Finder

If you double-click on a LightspeedC source file document in the Finder, LightspeedC will, as usual, launch and prompt you for a project document. Once you have selected a project, the source file will be opened up automatically.

RelConv Enhancements

RelConv accepts a script file containing a list of *.Rel* files, one per line. The script file must be a text file with an extension of *.RCV*. If relative pathnames appear in the script file, they are interpreted relative to the directory containing the script file.

RelConv now has an **Options** menu. There is only one option, requesting RelConv to delete each *.Rel* file after converting it.

RelConv now has a **File** menu with **Transfer...** and **Quit** selections.

RelConv can convert *.Rel* files generated by MDS 1.0, MDS 2.0, and Consulair's Mac C compiler.

3
The Editor

Windows Menu

The **Windows** menu has three sections, separated by dotted lines. The first section has five commands: **Clean Up, Zoom (⌘/), Full Titles, Close All,** and **Save All**. The second section has an entry for the project window and one for each Untitled window. The third section has an entry for each file open in an edit window, in alphabetical order.

Clean Up
The edit windows are neatly re-stacked, as though they were freshly opened. The rearmost window is assigned the first slot, as though it was the first window opened, the next-rearmost window is assigned the second slot, etc. The front-to-back order of the windows is not changed.

Zoom (⌘/)
The front window is zoomed to full screen; if it is already at full screen, it is restored to its previous position and size. This is equivalent to clicking the window's zoom box in the upper right corner of the window. Note that this feature does not require 128K ROMs.

Full Titles
This is a checkable item; it is initially unchecked. When checked, the title of each edit window indicates the volume name and directory name as well as the file name.

Close All
All edit windows are closed. If the **Confirm Saves** option is checked, you are asked whether you want to save each modified window, otherwise windows are automatically saved.

Save All
All modified edit windows are saved. No confirmation is requested.

Project window (⌘0) (Command-zero, not Command-oh)
The project window is brought to the front. (The menu item will be the name of the project, not the literal words "Project window".)

Titled and Untitled edit windows (⌘1-9)
The selected window is brought to the front. The number reflects the "slot number", i.e. the initial position of the window. (The first created window occupies slot #1, and slots #6-10 occupy the same screen positions as slots #1-5, etc. Slots vacated by closed windows are reused at the next opportunity.) The number of windows is limited only by available memory, but only windows in the first nine slots have a ⌘-key equivalent. A diamond (◇) appears next to windows which have been modified.

Find Definition of Symbol

If the Option (or ⌘) key is depressed while double-clicking in an edit window to select a word, LightspeedC opens the file in which the selected symbol is defined and searches for its first occurrence in that file. This will often be the definition of the symbol, but not always. The search string is set to the symbol, so **Find Again** (⌘A) can be used.

This is not a pure text operation but rather is based on information maintained by the compiler. The file defining the symbol must have been compiled for the editor to be able to find it. Furthermore, only global (non-`static`) functions and variables can be found in this way, since no information is retained about other names after compilation.

If the symbol is multiply defined, one of the files defining it is selected arbitrarily.

Pop-up Include-File Menu

Option- (or ⌘-) clicking in the title bar of an edit window brings up a menu of include-file names. The window must contain a *.c* file; the menu will display the names of files included by the *.c* file the last time it was compiled in the currently open project. Choosing a file from the menu opens that file or brings its window forward.

This is not a text operation; it is based on information known as of the last compilation. The editor doesn't know about `#include` statements. To open an include-file added since the last compilation, select its name and use **Open Selection** (⌘D).

If a source file has a lot of include-files, the menu could be too long to fit on the screen; it will be truncated. If you are running System 3.2, the menu will automatically scroll when you position the mouse at the bottom of the menu. (If you are running System 2.0 and prefer not to upgrade to 3.2, you can use ResEdit to copy the `MDEF 0` resource from the 3.2 System file into your System file; this gives you scrolling menus, even on 64K ROMs!)

Undo

The last edit operation may be reversed by selecting **Undo** (⌘Z) from the **Edit** menu. The name of the command in the menu changes appropriately (e.g. **Undo Paste**). Once undone, the command changes to **Redo** (**Redo Paste**, etc.). The change may be restored by selecting the command again. If there is nothing to undo, the command is dimmed. If the currently available undo operation applies to a window which is not frontmost, the name of the command in the menu indicates that there is something to undo, but the command is dimmed.

Replace All and **Revert** cannot be undone. **Set Font...** and **Set Tabs...** also cannot be undone; however, since they do not modify the text, they are not considered edit operations and therefore they do not affect the current undo status.

"Balance" Command

Balance (⌘B) extends the current selection in both directions until it encloses the smallest surrounding balanced text enclosed in parentheses `()`, brackets `[]`, or braces `{}`. Successive invocations select larger sequences of text.

Try this: Start at the beginning of a file and search for the first left brace `{`. Then use **Balance** (⌘B) and **Find Again** (⌘A) repeatedly until you get to the end of the file. This is a quick way to check whether all your function definitions are properly balanced.

Balance is a purely textual operation. It does not know about comments or strings.

Keyboard Usage

Cursor Key Support

In the editor, arrows move the insertion point up, down, left, right. Option-arrows move the insertion point to the beginning of file, end of file, beginning of line, end of line. Shift-arrows and shift-option-arrows extend the current selection.

Note: Unfortunately, it is not possible to distinguish between shift-arrows and the +, *, /, and = keys on the numeric keypad of the Macintosh Plus keyboard. For example, when you type a + on the keypad, it will be treated as a shift-left-arrow. This is unavoidable because of the hardware design.

"Enter" Key

As before, the "Enter" key brings the start of the selection onto the screen. If the start of the selection is already visible, "Enter" now makes the end of the selection visible. You can thus toggle between the two ends of the current selection. This is particularly useful after using the **Balance** (⌘B) command.

"Clear" Key

The "Clear" key on the keypad or Macintosh Plus keyboard is equivalent to the **Clear** command in the **Edit** menu.

Default Font/Tab Settings

By default, an Untitled window created by the **New** (⌘N) command appears in Monaco-9 with tabs set every 4 spaces. You can change this by using ResEdit to modify the `CNFG 0` resource in the LightspeedC application. This resource consists of four two-byte words. The first word stores option settings, and should be changed only by the **Options...** command. The second, third, and fourth words specify the font number, font size, and tab width, respectively, used for newly created Untitled windows.

Search and Replace

"Enter Selection" Command

The **Enter Selection** (⌘**E**) command, in the **Search** menu, sets the search string to the current selection, clearing **Grep** and **Multi-File Search**. You can then use **Find Again** (⌘**A**) to begin searching, or **Find...** (⌘**F**) to set search options.

Disabling Multi-File Search

Entering a new search string cancels a pending multi-file search.

To enter a new search string without cancelling a pending multi-file search, bring up the **Find...** (⌘**F**) dialog, click **Multi-File Search**, click **OK** in the multi-file selection box, and then enter the desired search string.

To cancel a pending multi-file search without entering a new search string, option- (or ⌘-) click on the **Multi-File Search** checkbox.

v1.02

⌘Tab and ⌘Return

The tab and return characters may be entered directly in the **Find...** dialog box by typing them with the Command (⌘) key held down.

v1.02

Grep

Warning: What follows in the section describing Grep mode is not for the faint-hearted. If you are not already familiar with Grep, it is strongly recommended that you experiment with this feature before attempting to use it on your files. It is both powerful and tricky, and is frequently useful, but can lead to unexpected and unfortunate results if not used correctly.

The *grep* mode of operation for the LightspeedC **Search** command is a powerful tool. At the expense of being somewhat slower than the normal *string-search* mode, the grep mode allows you to search for one of a set of many strings instead of a particular string. As a simple example, you can search for any occurrence of an identifier beginning with the letter `P`, such as `Ptr` or `Point`, instead of one specific string. This is described later in this section.

Grep is a standard utility on Unix™ systems that is used to search through one or more files for occurrences of strings that match a given *pattern* and print the lines containing those strings. Most Unix-based editors also have this *pattern-search* capability, most notably *ed* and *ex* derivatives such as *vi*.

A pattern is a string of characters that, in turn, describes a set of strings of characters. We say that a string is *matched* by a pattern if it is a member of the set described by the pattern. When you invoke one of the **Search** menu selections in grep mode, LightspeedC looks for and selects the next occurrence of a string that matches that pattern.

The kinds of patterns we describe below are sometimes referred to as *regular expressions*, but they really are not. True regular expressions are capable of describing sets of strings that these patterns cannot. By convention, the expressive power of these patterns is limited somewhat so that pattern searching can be implemented more efficiently.

We begin by describing the simplest kinds of patterns, those that match a single character. Afterwards, we will describe how to construct more complex patterns. Keep in mind that pattern matching is done on a line-by-line basis. Therefore, a pattern will never match a string that spans line boundaries because the character that separates lines (carriage return, the ASCII CR) can never be matched.

- Any character, with the exceptions noted below, is a pattern that matches itself. So, for example, the pattern `2` will match an occurrence of a character `2` in the text being searched. Note that if you have checked **Ignore Case** in the **Find...** dialog box any letter will match both its upper- and lower-case equivalent. Thus, either `a` or `A` will match both `a` and `A`.

- The character `.` is a pattern that will match any character.

- The character `\` followed by any character is a pattern that matches that character, except for `( ) < >` or one of the digits `1-9`. Thus a `.` can be matched with `\.` and a `\` can be matched with `\\`.

- A string of characters *s* surrounded by a `[` and a `]` is a pattern `[`*s*`]` that matches any one of the characters in the string *s*. The pattern `[^`*s*`]` matches any character that is not in the string *s*. If a string of three characters in the form *a*`-`*b* occurs in *s*, this represents all of the characters from *a* to *b* inclusive. All other characters in *s* are taken literally, and the only way to include the character `]` in *s* is to make it the very first character. Likewise, the only way to include the character `-` in *s* is to have it either at the very beginning or the very end of *s*. Note that the **Ignore Case** option has no effect between brackets. As an example, the pattern `[A-Za-z0-9]` matches any alphanumeric character. The pattern `[^!-~]` matches any character that is not one of the standard, printable ASCII characters.

Since we want to be able to match strings, not just individual characters, we need to have patterns that are able to match consecutive sequences of characters. One way of doing this is to append a `*` to the end of one of the above patterns.

- A pattern *x* followed by a `*` is a pattern *x*`*` that matches zero or more consecutive occurrences of characters matched by *x*. For example, the pattern `@*` will match a string containing any number of at-signs. If a string to be matched does not begin with an at-sign, or contains no at-signs at all, then we say that the pattern matches the *empty string* at the beginning of the string to be matched. The utility of this will be demonstrated further on.

Patterns may be *concatenated* to form more complex patterns:

- A pattern *x* followed by a pattern *y* forms a pattern *xy* that matches any string *ab*, where *a* can be matched by *x* and *b* can be matched by *y*.

  Of course, the compound pattern *xy* can be concatenated with another pattern *z*, forming the pattern *xyz*.

As an example of a pattern that uses the capabilities described so far, consider the pattern `(.*)`. This pattern will match any string that is enclosed in parentheses. This includes the string `()`, since the sub-pattern `.*` will match the empty string between the `(` and the `)`. But what about the string `(())`? Since the sub-pattern `.*` will match any number of occurrences of all characters, won't the pattern match just the `(()` and not the very last `)` ? The answer is any sub-pattern of the form *x** in a pattern *x*y* will match the largest number of occurrences of whatever *x* matches that still allows a match to *y*. Therefore, in matching `(())` against the pattern `(.*)`, only the inner pair of parentheses will be matched by the sub-pattern `.*`.

We now have the ability to form patterns that are composed of sub-patterns, and will find it useful to *remember* sub-strings matched by sub-patterns and to be able to match against those sub-strings.

- A pattern surrounded by `\(` and `\)` is a pattern that matches whatever the sub-pattern matches.

- A `\` followed by *n*, where *n* is one of the digits `1-9`, is a pattern that matches whatever was matched by the sub-pattern beginning with the *n*th occurrence of `\(`. A pattern `\`*n* may be followed by an `*`, and forms a pattern `\`*n*`*` that matches zero or more occurrences of whatever `\`*n* matches.

  Note that, in the pattern `\(a.b\)\1`, the sub-pattern `\1` does not imply a re-application of the sub-pattern `a.b`. If `\(a.b\)` was matched with the string `axb`, then the sub-pattern `\1` would try to match the literal string `axb` against the remainder of the search string. Therefore, the pattern `\(a.b\)\1` will match `axbaxb`, but will not match `axbazb`.

We finally will find it useful to be able to *constrain* patterns to match only if certain conditions in the context outside the string matched are met.

- A pattern surrounded by `\<` and `\>` is a pattern that matches whatever is matched by the sub-pattern, provided that the first and last characters of the matched string can be matched by `[A-Za-z0-9_]` and that the characters immediately surrounding the matched string cannot be matched by `[A-Za-z0-9_]`.

  This is used to match any string that matches the sub-pattern only if the matched string begins and ends on a word boundary. Note that if you have checked **Match Words** in the **Find...** dialog box, then the entire pattern you enter will be treated as though it were surrounded by `\<` and `\>`.

- A pattern *x* that is preceded by a `^` forms a pattern `^`*x*. If the pattern `^`*x* is not preceded by any other pattern, it matches whatever *x* matches as long as the first character matched by *x* occurs at the beginning of a line. If the pattern `^`*x* is preceded by another pattern, then the `^` is taken literally.

- A pattern *x* that is followed by a `$` forms a pattern *x*`$`. If the pattern *x*`$` is not followed by any other pattern, it matches whatever *x* matches as long as the last character matched by *x* occurs at the end of a line. If the pattern *x*`$` is followed by another pattern, then the `$` is taken literally.

These last two items constrain pattern matches to begin or end at line boundaries, and can be combined to constrain a pattern to match an entire line only.

We mentioned at the beginning of this section that you could use grep mode to search for any identifier beginning with the letter `P` . You can do this with the pattern `\<[Pp][A-Za-z0-9_]*\>` . Note that, if you had checked **Ignore Case** in the **Find...** dialog box, then the patterns `\<P[A-Za-z0-9_]*\>` and `\<p[A-Za-z0-9_]*\>` would match the same strings. Also, if you had checked **Word Match** in the **Find...** dialog box, then any of these patterns with the `\<` and `\>` removed would match the same strings.

In grep mode, not only is searching different from the normal mode of operation, but replacement is also. In a replacement string (as specified in the **Replace with:** portion of the **Find...** dialog box), the following substitutions are made before any text replacement occurs:

- Each occurrence of the character `&` is replaced with whatever was matched by the entire pattern.

- Each occurrence of a string of the form `\`*n*, where *n* is one of the digits `1-9`, is replaced by whatever was matched by the sub-pattern beginning with the *n*th occurrence of `\(`.

- Each occurrence of a string of the form `\`*x*, where *x* is a character other than one of the digits `1-9`, is replaced by *x*.

This allows you to not only be able to search for a string satisfying a complex set of conditions, but also to be able to do a subsequent replacement that varies depending on the string that is matched.

As an example, suppose that you have written a program that is to become a Macintosh application (i.e., it uses the Macintosh Toolbox instead of *stdio* for the user interface). Suppose also that you have discovered that you have forgotten to put a `\p` at the beginning of your string constants, so that your program will pass C strings instead of Pascal strings to the Toolbox (which expects Pascal strings as arguments). You can easily change all your C strings to Pascal strings by specifying

```
"\(.*\)"
```

as the search pattern and

```
"\\p\1"
```

as the replacement string.

4
The Compiler

Code Generation

v1.02

Register Variables

There are actually more register variables available than the *LightspeedC User's Manual* indicates. In fact, five (5) data registers and three (3) address registers are available to hold register variables. (Of course, any registers not used to hold register variables are available to hold temporary results during expression evaluation.) In a desk accessory, device driver or code resource, address register `A4` is reserved to point to the global data area, so only two (2) address register variables are available.

Short Branches

The code generator produces short branches when it can. Backward branches are always generated correctly and optimally. Forward branches are always generated correctly, but occasionally not optimally: some long forward branches are generated which could be short. (Nobody's perfect.)

On average, this optimization seems to be good for about 4% to 8% reduction in code size.

Stack Frames and the "Macsbug Symbols" Option

Previously, setting the **Macsbug Symbols** option forced the compiler to generate MC68000 `LINK` and `UNLK` instructions to create a stack frame for each function. This is because Macsbug looks for these instructions to find functions whose names are embedded in the code. When the **Macsbug Symbols** option was unchecked, only functions which had arguments, non-register local variables, or compiler-generated temporaries were given a stack frame.

Now, the setting of the **Macsbug Symbols** option has no effect on the code generated for each function. A stack frame is never generated solely for Macsbug's benefit, and function names are embedded in the code only for functions with stack frames. (You can force a function to have a stack frame by declaring a dummy argument or non-register local variable.)

Note that, as before, the **Profile** option forces a stack frame, even if none would otherwise have been needed. Some assembly language routines (e.g. those in *stringsasm.c*) are not expecting stack frames and should not be compiled with the **Profile** option.

Floating to Integer Conversions

Floating-point values converted to arithmetic types are now truncated, rather than rounded, to integral values.

Function Prototypes

This is an ANSI extension to C. A function prototype is a function declaration with additional information describing the arguments. For example:

```
extern int strcpy(char *dest, char *source);
extern int printf(char *, ...);
```

Argument identifiers are optional and are ignored if supplied. Functions with variable numbers of arguments can be specified by using . . . as the last argument; no information is provided about the additional arguments beyond that they are allowed.

A function declaration with no argument information, e.g.

```
extern long foo();
```

is not a prototype and supplies no information about the arguments. It only declares the return type. To give a prototype stating that the function takes no arguments, use

```
extern long foo(void);
```

(This is a special case and does not mean that the function takes a `void` argument!)

If a prototype is in effect when a function is called, the actual arguments are checked against the prototype. Unless the prototype ends in . . ., the number of arguments must match exactly. The types of the arguments must be assignment-compatible (the state of the **Check Pointer Types** option is honored). Appropriate conversions are applied to arithmetic (integral and floating-point) values. Additional arguments allowed by the . . . are not checked or converted.

If a prototype is in effect when a function is defined, the definition is checked against the prototype. Unless the prototype ends in . . ., the number of arguments must match exactly. The types of the arguments must be identical. Additional arguments allowed by the . . . are not checked.

If no prototype is in effect when a function is called, the "null" prototype `( . . . )` is assumed. However, if the **Require Prototypes** option is checked, and the function is not a built-in Macintosh call, an error is signalled.

If no prototype is in effect when a function is defined, the "null" prototype `( . . . )` is assumed. However, if the **Require Prototypes** option is checked, an error is signalled.

Prototypes are optional (unless you have checked **Require Prototypes**), but if supplied must appear before the first definition or use of the function. A function definition is not itself a prototype, so the following example will not work:

```
f(x)
long x;
{
        ...
}

g()
{
        ...
        f(0);
        ...
}
```

The 0 argument will be passed to `f` as an `int`, not a `long`. To get automatic type coercion, a separate prototype—`int f(long)`—must be supplied before the function `f` is called.

You can supply prototype information for a built-in Macintosh call; it will be checked against the built-in size and count information, and the prototype will subsequently be used in preference to the built-in information. For example:

```
extern pascal Handle GetResource(OSType, int);
```

Full prototype information is *not* built in for the Macintosh calls.

"Require Prototypes" Option

This option forces very strict type checking. When set, you can neither define nor use any function (except for built-in Macintosh calls) without providing a prototype first. By default, the option is not set.

To take best advantage of this feature, place all your prototypes in include-files so that all definitions and uses of functions will be checked against the same specification.

"Check Pointer Types" Option

This option is set by default. If it is cleared, all pointer types are considered compatible, and the "`pointer types do not match`" error message is not generated. (However, when subtracting two pointers, the two types must be pointers to objects of the same size.)

The Inline Assembler

Assembly Syntax

Assembly language statements may be interspersed with C code. The syntax is as follows:

```
asm {
     ...        ; assembly instructions, one per line
}
```

This is syntactically a C statement and can appear anywhere a statement can; therefore, it must appear inside of a function definition. End-of-line is significant: only one assembly instruction can appear on each line. Assembly-style semicolon-to-end-of-line comments are allowed. But it's still C, so C-style comments are also recognized, as are preprocessor commands. Preprocessor macros are expanded whenever they appear.

Standard MC68000 assembly syntax conventions are followed with a few exceptions. All instructions are available, as is the `DC` ("define constant") directive to place literal values in the code stream. No other assembly directives are recognized; use C to declare data, to define symbolic constants, and to import and export symbols. The assembler is case-insensitive with respect to instruction mnemonics, register names, and size specifications (`.B`, `.W`, `.S`, `.L`).

Arbitrary C constant expressions may appear wherever a constant would be legal.

Use of C Identifiers

C identifiers may be referenced by name in inline assembly. The base register (`A6` for locals, `A5` or `A4` for globals) is optional, but if supplied it must be correct. Fields of structure variables may also be referenced directly. Register variables may be referenced by name wherever a register would be legal. The assembler is case-sensitive with respect to C identifiers.

C identifiers which conflict with register names (`D0-D7`, `A0-A7`, `SP`, `USP`, `SR`, and `CCR`) cannot be referenced by name in inline assembly.

Offsets of fields in a structure may be referred to symbolically using the macro `OFFSET`, defined in *asm.h*. For example,

```
long refcon(wp) /*  same as "GetWRefCon"  */
WindowPtr wp;
{
     asm {
          move.l    wp,a0
          move.l    OFFSET(WindowRecord,refCon)(a0),d0
     }
}
```

Note, however, that fields of variables may be referenced directly, e.g.:

```
long myRefcon() /*  refCon of myWindow  */
{
     extern WindowRecord myWindow;
     asm {
          move.l     myWindow.refCon,d0
     }    /*  same as "return(myWindow.refCon)"!  */
}
```

In inline assembly, no implicit declaration is made for the identifier appearing in a `JSR`. All functions must be declared before they can be used in a `JSR` statement.

Labels and Branching

Instructions may be labelled with C or assembler labels. Assembler labels consist of an at-sign (`@`) followed by one or more digits; a colon is optional following an assembler label. C labels may appear within inline assembly; they must be followed by a colon in accordance with the C syntax. You may `goto` such a label from C code! It is also possible to refer to a C label from inline assembly, whether the label appears in assembly code or in C code, but the label must be preceded by `@` to indicate to the assembler that it is a label. (This is necessary to avoid ambiguity in statements such as: `LEA FOO, A1`.)

Certain C statements that branch are also allowed inside inline assembly:

```
break      ; exits the surrounding loop or switch
continue   ; skips to next iteration of the surrounding loop
return     ; exits function
goto label; same as "bra @label"
```

Do not use the `RTS` instruction unless you are absolutely sure you know what you are doing. Use `return`, or simply "fall through" to the end of the function, to clean up the C stack frame properly. Do not specify a return value in the `return` statement; if the function returns a value, place it in the proper place (usually `D0`) before returning. (Refer to Chapter 9 of the *User's Manual* for information on calling conventions.)

A short branch (`BRA.S`) to the immediately following instruction is an error which is not detected by the inline assembler. (It generates an 8-bit zero displacement, which results in the next instruction word being used as a 16-bit displacement for a long branch rather than being executed as an instruction. Refer to the MC68000 manual for more details.)

ROM Traps

The built-in *Inside Macintosh* calls, except for those which are `[Not in ROM]`, may be used as instructions; the appropriate trap number will be assembled. The assembler is case-sensitive with respect to trap names. Trap names may optionally be preceded by an underscore. Register-based traps are assembled inline even though they generate calls to *MacTraps* "glue" when used from C. (Note hat on 64K ROMs, Memory Manager traps do not set the low-memory global `MemErr`, though the glue does. You can force a call to the glue by using a `JSR`.) You can provide an optional argument, a

2-bit value to be placed in bits 9..10 of the trap. This allows you to set trap modifier bits, such as AUTOPOP, SYS, CLEAR, and ASYNC. The various trap modifier bits are defined in *asm.h*. For example:

```
Handle NewSysHandle(size)
{
      asm {
            move.l      size,d0
            NewHandle   CLEAR+SYS
            move.l      a0,d0
      }
}
```

When you issue a JSR to a built-in routine, the routine's identifier, like all function identifiers in inline assembly, must be declared before it can be used. For example:

```
extern pascal void GetIndString();
...
asm {
      ...
      JSR  GetIndString
      ...
}
```

Register Usage

You may modify registers D0, D1, D2, A0, and A1, as well as registers being used to hold register variables. All other registers should be saved and restored if you need them. If you want to use a register other than for scratch purposes, declare a register variable; you will be able to refer to it by name, and you won't have to bother to save and restore its value.

If intervening C code is executed between two stretches of inline assembly, you can assume that the C code preserves the values of registers A5, A6, and A7—and A4 as well for drivers and code resources. All other registers may have been modified. It is safe to leave things on the stack while C code is executing, provided the stack is cleaned up before returning from the function.

For further information on calling and register conventions, refer to Chapter 9 of the *User's Manual*.

Differences From Other Assemblers

Assembler labels are not local to the sequence of inline assembly in which they appear. Their scope extends throughout the function in which they appear, as C labels do.

The DC.B directive always generates an even number of bytes. A zero pad byte is generated if an odd number of values are specified. For example:

```
DC.B 'a','b','c','d'    ; Assembles as four packed bytes

DC.B 'a'                ; These assemble as four words with
DC.B 'b'                ; zero pad in the low byte
DC.B 'c'
DC.B 'd'
```

The syntax $NNNN is not available to designate a hex constant. Use the C syntax 0xNNNN instead.

The difference of two addresses is not a constant expression; therefore instructions like

```
move.w  #@2-@1,d0
```

are not possible. Similarly, LightspeedC does not support the syntax

```
dc.w    @1-*
```

to assemble the PC-relative offset of the label @1. Use

```
dc.w    @1
```

instead. For example, to code a dispatch table, use:

```
;  d0 contains 0,1,2,...
     add.w   d0,d0
     add.w   @0(d0.w),d0
     jmp     @0(d0.w)
@0   dc.w    @1              ;  case 0
     dc.w    @2              ;  case 1
     ...                     ;  ...
```

The alternative syntax for PC-relative addressing—@1(PC) instead of @1, or @1(PC,D0) instead of @1(D0)—is not supported.

Omitting a zero displacement in the Address Register Indirect with Index addressing mode—(A1,D2.W) instead of 0(A1,D2.W)—is not supported.

Syntax for Absolute Variables

There is a new syntax for defining absolute variables, such as low-memory globals. Previously, one used, e.g.:

```
#define MemErr  * (int *) 0x220
```

The following can now also be used:

```
extern int MemErr : 0x220;
```

Both forms work in C, but only the new form allows low-memory globals to be used in inline assembly. The Macintosh include-files have been updated to use the new syntax. This feature is not supported for absolute variables at locations greater than 0xFFFF.

Redeclaring Built-In Functions

Previously, redeclaring a built-in Macintosh call caused the internal definition to be overridden. It was therefore necessary to use

```
#define NewHandle    (Handle) NewHandle
#define FrontWindow  (WindowPtr) FrontWindow
```

though it might be considered more natural to use

```
extern pascal Handle NewHandle();
extern pascal WindowPtr FrontWindow();
```

The latter would override the built-in definitions, preventing the argument checking which would normally be done, and leading to a link error "`undefined: FrontWindow`".

This has been fixed, and the Macintosh include-files have been updated to use the new syntax. Note that built-in functions can still be overridden by providing definitions for them, or by declaring them to be C (rather than `pascal`) functions.

5
Running with Switcher

Note: Switcher version 5.0 or greater is required to use the Switcher-**Run** feature. Do not use LightspeedC with earlier versions of Switcher.

If you run LightspeedC under Switcher, when you **Run** (⌘R) your project it is placed into a Switcher partition allocated from LightspeedC's own memory. When execution of your project terminates, you are returned to LightspeedC instantly! **Transfer...** behaves similarly, running a selected application rather than the project.

Furthermore, during execution you can switch back to LightspeedC by clicking in the Switcher arrow at the upper right portion of the screen. You can now view your sources, edit files, etc. Your project remains in its partition; you can switch back to it by selecting **Resume** (⌘R) from the **Project** menu (the **Run** command changes to **Resume**). You can switch back and forth between LightspeedC and your executing project in this way. While in LightspeedC, you cannot compile or do anything else which might modify the project. Nor can you close the project, or **Quit** (⌘Q) or **Transfer...**, since that would leave the executing project in limbo.

You will discover that your project is running in a Switcher partition which only LightspeedC knows about. If you return to Switcher itself, you will not see it listed among the current applications. You cannot switch to the project by using the Switcher arrows: the only way to switch to it is to use the **Resume** command from LightspeedC. Once you've **Resumed**, the Switcher arrows will only switch you back to LightspeedC.

Before Switcher-**Run**ning your project, LightspeedC saves all modified edit windows, just as it does normally for **Run**, but it does not close them. If you have the **Confirm Saves** option checked, you are asked whether this is OK. You are asked only once for all the files. Note that clicking **No** does not mean that pending edits will be discarded, only that they will remain pending. (Of course, pending edits will be lost if your program crashes so badly that it cannot return to LightspeedC.)

LightspeedC allocates as much memory as it can for your project to **Run**. It reserves a little extra for itself, so that there will be enough memory to edit files in. To be sure, though, you might want to open any files that you may want to edit while the project is running before issuing the **Run** command. Conversely, if you have a lot of files open you might need to close some first so that the project will have enough memory to run.

The Switcher-**Run** feature is of limited usefulness on a 512K Macintosh. Only small projects (such as desk accessories) can be developed in this way. Accordingly, LightspeedC comes pre-configured for use with Switcher on a 1024K Macintosh; if you try to run it under Switcher on a 512K Macintosh you will be told "there is not enough memory to do that". To run LightspeedC under Switcher on a 512K Macintosh, use the Switcher command **Configure then Install...**; allocate LightspeedC as much memory as Switcher will let you.

LightspeedC's use of ⌘[and ⌘] for the **Shift Left** and **Shift Right** commands conflicts with Switcher's use of these key combinations to cycle among the active tasks running under Switcher. Therefore, you should check **Disable Keyboard Switching** in Switcher's **Options...** dialog.

Programs that do not respond properly to update events, or that do not use windows, may not redisplay their screens correctly after switching out and back in. (The *Pongerang* demo distributed with LightspeedC is such a program.)

Caution: Do not **Quit** from Switcher without first **Quit**ting from LightspeedC. Crashes will result.

6
Drivers and Code Resources

Running a Desk Accessory

The **Run** (⌘R) command in the **Project** menu is now enabled for desk accessory projects. The DA is built (exactly as by **Build Desk Accessory...**), and then a DA shell program is launched. The shell program is called *DAShell* and must appear in the same volume (and folder under HFS) as LightspeedC itself.

The DA shell does little besides support desk accessories. Your DA appears in the menu, as do all system DA's except for those whose name or number (12) conflict with yours. When you quit the shell you will return to LightspeedC. To test goodbye-kisses without returning to LightspeedC, the shell has a **Relaunch** command which restarts the shell.

v1.02

Global Data Area in Drivers

Usually, your driver gets control when the function `main` is called by the Device Manager, as described in Chapter 9 of the *User's Manual*. If any routine in your driver is called in any other way, address register `A4` will not be set up to point to the global data area, and your global variables will not be accessible.

For example, suppose your driver puts up a dialog box and you supply a dialog filter procedure (refer to the "Dialog Manager" chapter of *Inside Macintosh* for details):

```
engage_in_dialog()
{
   extern pascal Boolean myFilter();
   int item;

   ...
   ModalDialog(myFilter, &item);
   ...
}
```

When `ModalDialog` calls the filter procedure, register `A4` may not point to your global area. The solution is to use `SetUpA4` and `RestoreA4`:

```
pascal Boolean myFilter(dp, eventp, itemp)
DialogPtr dp;
EventRecord *eventp;
int *itemp;
{
   Boolean result;

   SetUpA4();
   ...
   RestoreA4();
   return(result);
}
```

The same holds for other call-back routines. Of course, if your routine does not need to access your driver's globals, you needn't bother to call `SetUpA4` and `RestoreA4`.

`SetUpA4` and `RestoreA4` are in the *MacTraps* library, even though they are not described in *Inside Macintosh*. They are analogous to `SetUpA5` and `RestoreA5` (described in the "OS Utilities" chapter of *Inside Macintosh*), which an application program uses to gain access to its globals from an interrupt handler such as a completion routine. (The Toolbox always makes sure register `A5` is set up when calling an ordinary call-back routine such as a filter procedure, so applications don't ordinarily need to use these routines.)

The sample desk accessory *Windows* included with LightspeedC illustrates the use of `SetUpA4` and `RestoreA4` to allow access to the driver's globals from within a trap intercept routine.

Global Data Area in Code Resources

Code resources may now have global and static data. As in the case of drivers, the global data area is addressed off of `A4`. Unfortunately, `A4` is not set up automatically as it is with drivers; you have to take care of it by yourself. When `main` is called, `A0` points to the global data area. Therefore, `A0` must be moved into `A4`. One way to do this is:

```
#define SetUpA4()    asm {move.l  a4,-(sp)   \
                          move.l  a0,a4}
#define RestoreA4()  asm {move.l  (sp)+,a4   }
```

Use `SetUpA4()` immediately when `main` is entered, and use `RestoreA4()` before returning.

(Note that these `#defines` are <u>not</u> the same as the functions `SetUpA4()` and `RestoreA4()` described in the preceding section about "Global Data Area in Drivers", but are named the same to reflect their similar functionality.)

This isn't enough if you require a callback routine (e.g. filter proc) to have access to your globals. This somewhat more complex scheme will do the job:

```
#define RememberA4() _A4_(1)
#define SetUpA4()    asm { move.l  a4,-(sp) }; _A4_(0)
#define RestoreA4()  asm { move.l  (sp)+,a4 }

_A4_(remember)
{
   asm {
        bra.s   @1
save:   dc.l    0     ;  keep saved A4 here
@1      lea     @save,a1
   }
   if (remember)
        asm { move.l  a0,(a1) }
   else
        asm { move.l  (a1),a4 }
}
```

Use `RememberA4()` immediately when `main` is entered, then use `SetUpA4()` and `RestoreA4()` as required.

v1.02

"Opening" an Open Driver

The `open` entry point of a driver (`main`'s third argument = 0) may be called even if the driver is already open. This happens, for example, when the user selects from the menu the name of a desk accessory that's already on the screen. The driver should check to see if it is already open and avoid repeating its initialization sequence if so.

Chapter 9 of the *User's Manual* recommends setting the fields of the device control entry directly in order to inform the Device Manager that the driver must be locked between calls, needs to be called periodically, etc. But the Device Manager copies the `dCtlFlags`, `dCtlMenu`, `dCtlDelay`, and `dCtlEMask` fields of the driver's header to the corresponding fields of the device control entry each time the driver is opened, even if it is already open. Therefore these fields must be set to their proper values each time. The `open` routine of the driver (called from `main` when the third argument is 0) should look something like this:

```
doOpen()
{
   dce->dCtlFlags|= dNeedLocked;  /*  or whatever  */
   if (already_open)
      return;
   already_open = 1;
   /*  one-time initialization  */
   ...
}
```

The sample desk accessory *Windows* included with LightspeedC illustrates this technique.

Open/Close Result Codes in Drivers

The new ROM supports result codes passed back from the Open and Close entries. If a negative result is returned from the Open entry, the driver is not opened; if `closeErr (-24)` is returned from the Close entry, the driver is not closed.

The LightspeedC interface glue now recognizes these cases and manages the `DATA` resource, used to store the driver's global data, correctly.

Warning: For your drivers to work correctly on 64K ROMs, you must be sure to return zero from your Open and Close entries. (Previously, it didn't matter what you returned.)

An additional special case is recognized: if `1` (one) is returned from the Close entry, the handle in the `dCtlStorage` field of the device control entry is not deallocated. A result code of zero is actually returned to the Device Manager. You might use this to keep your globals around until the driver is re-opened.

Runtime Routines in Code Resources

Some 400+ bytes of runtime routines are linked into code resources to implement the `switch` statement and the multiply, divide, and modulo operations on `long` operands. Previously, these routines were always linked in. Now, they are linked in only if they are needed. (However, the routines are a unit, so if any routine is used, they are all linked in.) This optimization applies to code resources only.

Additional Header Information for Drivers and Code Resources

The resource name of a desk accessory or device driver is now placed in the driver header beginning at offset 18. Exactly 32 bytes are reserved in the header for this purpose, so names longer than 31 characters are truncated.

Six bytes of resource type and ID are now placed starting at offset 4 of a code resource. Additionally, the two bytes starting at offset 2 and the six bytes starting at offset 10 are set to zero; these are unused and may be patched as desired.

7
Libraries

128K ROM and HFS support

LightspeedC has been fully upgraded to conform to Volume 4 of *Inside Macintosh*. The new calls are recognized and argument-checked in the usual way. The *MacTraps* library and the Macintosh include-files have been updated appropriately; files *HFS.h*, *ListMgr.h*, *SCSIMgr.h*, and *TimeMgr.h* are new. There are several minor issues which bear mentioning:

- The new Toolbox Utilities routines `Fix2X` and `Frac2X` return SANE Extended (equivalent to LightspeedC `double`). Their Pascal declarations are:

```
FUNCTION Fix2X (x:Fixed): Extended;
FUNCTION Frac2X (x:Fract): Extended;
```

  Unfortunately, returning an Extended type is illegal for a pascal routine in LightspeedC. Instead these must be called as C functions (via glue in *Mactraps*). Before calling them, they must be declared as follows:

```
double Fix2X(Fixed);
double Frac2X(Fract);
```

  ToolboxUtil.h contains these declarations. (If you forget these declarations, you will get the error "`wrong number of arguments`".) If you prefer, they can be called directly from inline assembly.

- The List Manager function `LLastClick` returns a `Cell` (same as a `Point`); this is also illegal for a pascal function. In LightspeedC, it returns a `long` instead.

- The new File Manager chapter in Volume 4 of *Inside Macintosh* gives definitions for the types `VCB`, `DrvQEl`, and `DrvQElPtr` which conflict with the original definitions. The new types have instead been defined as `HVCB`, `HDrvQEl`, and `HDrvQElPtr` so that both the original and current definitions of these types are available.

Other Macintosh Library Changes

- In Release 1.02, `SetUpA5` and `RestoreA5` were functions in the *MacTraps* library. In Release 2.01, they are macros in the header file *OSUtil.h*. Be sure to include *OSUtil.h* when you want to use `SetUpA5` and `RestoreA5`.

v1.02

- Glue for the Launch and Chain traps is provided in *MacTraps* and can be called from LightspeedC:

```
pascal void Launch(int config, char *filename);
pascal void Chain(int config, char *filename);
```

v1.02

- Interfaces for the RAM Serial Drivers are provided in the *MacTraps* library. Interfaces for Appletalk are provided in the *AppletalkLib* library. However, to use either of these facilities, additional resources must be included in the resource file for your program. These resources are supplied with LightspeedC in the following two files:

SERD	which contains RAM Serial Driver resources
ATalk/ABPackage	which contains Appletalk resources

v1.02

- The *pascal.h* file may be found in the Mac #includes folder and contains the declarations for the following functions:

```
pascal void CallPascal();
pascal char CallPascalB();
pascal int  CallPascalW();
pascal long CallPascalL();
char *CtoPstr();
char *PtoCstr();
```

These functions are in the *MacTraps* library, even though they they are not described in *Inside Macintosh*. This file should be included whenever any of these functions are used. (See "Making Indirect Calls" in Chapter 9 of the *User's Manual*.)

Enhancements to Standard C Libraries

All projects that contained only one file have been changed to libraries to save space on the disk. These libraries are: *math*, *storage*, *storageu*, *strings*, and *unix_strings*. To rebuild one of these libraries, create a **New** project, **Add** the corresponding source file, and **Build Library....**

The source code to the libraries has grown, but in general the object size has shrunk somewhat due to code improvements, except for *printf-2-w.c*, which adds about 4K to *stdio* for the new window functions described below. If you would rather use the old version, replace *printf-2-w.c* with *printf-2.c* in *stdio*, recompile, and reload the *stdio* library.

The *strings* library has been rewritten in assembly language for efficiency. The source code is provided as *stringsasm.c*. The old version is provided as *strings.c* for those who still want to have Macsbug symbols or use the profiler with these functions.

proto.h contains function prototypes for use with the *math*, *stdio*, *storage*, *storageu*, *strings*,

unix, and *unix_strings* libraries. Its use is optional, but is useful if you check the **Require Function Prototypes** option. If you choose to use it, please note that it complements, but does not replace, the usual include-files for each library.

There are many new features and functions in the *stdio* library. A real Macintosh window titled "`console`" is automatically created for plain C applications the first time keyboard input or screen output is requested through the Standard I/O functions. In addition, , **File** and and **Edit** menus are added to the menu bar. Whenever keyboard input is expected, the commands in the menu bar created by the stdio library can be accessed. **Open** in the **File** menu will show the console window if it is hidden. **Close** in the **File** menu when the console window is frontmost (or clicking in its close box) will hide it; **Close** in the **File** menu when any other *stdio* window is frontmost (or clicking in its close box) will execute an `fclose` on that window. **Quit** in the **File** menu will do an `ExitToShell`. Also, when keyboard input is expected, desk accessories and FKEYs can be accessed.

Support for use of device drivers has been added to *stdio*. Device driver names can be passed to `fopen` and will be handled correctly. This has been done primarily to allow the use of the four standard Macintosh serial drivers (`.AIn`, `.AOut`, `.BIn`, `.BOut`); this allows the use of *stdio* with printers, modems, etc.

A new library, *sprintf/sscanf*, is provided. It is a subset of *stdio* allowing you to use the `sprintf` and `sscanf` functions in your programs without having to load the large *stdio* library into your project. *Stdio* itself has been better modularized, so that when a project is built into an application, many more unused functions will be stripped out by the linker than in release 1.02.

Note: The libraries *stdio* and *sprintf/sscanf* contain the library *setjmp.Lib*. If your project uses the `setjmp` and `longjmp` routines as well as *stdio* or *sprintf/sscanf*, do not explicitly add the *setjmp.Lib* library.

The *unix* library has also been enhanced to provide additional Unix emulation.

New Standard C Library Functions

The following pages describe some new library functions:

break to a debugger or exit

abort (unix)

SYNTAX

```
#include <stdio.h>
void abort()
```

DESCRIPTION

`Abort` automatically breaks to a debugger if one is present; otherwise the program is exited.

— enable/disable "Click to Continue" on exit

Click_on (stdio)

SYNTAX

```
#include <stdio.h>
void Click_On(flag)
Boolean flag;
```

DESCRIPTION

This function controls the "click mouse to continue" option. If `flag` is 1, the option is enabled; if `flag` is 0 the option is disabled. If this option is enabled, when a program that uses *stdio* terminates, a window (titled "`Exit Window`") will appear. Only when the user clicks in the close box, presses the return key, or (if the menus were created by *stdio*) selects the **Quit** command from the **File** menu will the program actually exit. This option is initially enabled.

turn printer echo on or off

Echo_to_Printer (stdio)

SYNTAX

```
#include <stdio.h>
void Echo_to_Printer(flag)
Boolean flag;
```

DESCRIPTION

Call this function with the argument `TRUE` to turn on printer echoing. When printer echoing is on, the console window text output will be echoed to the printer. If input is being echoed to the console window, it will be echoed to the printer as well.

Call this function with the argument `FALSE` to turn off printer echoing. This is the default condition.

launch a program

execl
execv
execle
execve
(unix)

SYNTAX

```
#include <stdio.h>
int execl(path)
char *path;

int execv(path, argv)
char *path; char *argv[];

int execle(path)
char *path;

int execve(path, argv, envp)
char *path, *argv[], *envp[];
```

DESCRIPTION

These functions just launch the file name indicated by `path`. There is no support for arguments or environment variables. (Refer to the System V Unix manual for a complete explanation.)

RETURN VALUE

These functions never return.

open a new *stdio* window

fopenw (stdio)

SYNTAX

```
#include <stdio.h>
FILE *fopenw(title, upperLeftCorner, optionsPtr)
char *title;
Point upperLeftCorner;
StdWindowOptions *optionsPtr;
```

DESCRIPTION

`fopenw` opens a new window with the specified title at the designated position (in global coordinates).

`optionsPtr` is a pointer to a `StdWindowOptions` structure as defined in *fopenw.h*. If the pointer is `NULL` (`0L`), then the window will be created with the following defaults: a 24-by-80 window, visible cursor, input echoed, tab width of 4 spaces, window brought forward on output to it (including echoing of input), grow box, draggable, go-away box, scrolling, and line wrap-around. (On big-screen Macintoshes, the default screen will be larger.) After calling `fopenw()`, the memory pointed to by `optionsPtr` is no longer needed.

RETURN VALUE

Returns an ordinary `FILE` pointer variable which can be passed to `fprintf`, `fclose`, etc.

v1.02

get cursor position

getxpos
getypos
(stdio)

SYNTAX

```
#include <stdio.h>
int getxpos()
int getypos()
```

DESCRIPTION

These functions return the x and y coordinates of the cursor in the current window.

RETURN VALUE

x or y coordinates of the cursor.

SEE ALSO

`setwindow`

get the screen buffer of *stdio* window

Get_ScreenPtr (stdio)

SYNTAX

```
#include <stdio.h>
char *Get_ScreenPtr(windowFileVar)
FILE *windowFileVar;
```

DESCRIPTION

This function returns the address of the screen buffer for the window associated with the `FILE` variable `windowFileVar`. The buffer can be treated as a two-dimensional array of characters whose size is given by the `maxcol` and `maxrow` fields of the `StdWindowOptions` structure defined in *fopenw.h*.

RETURN VALUE

The address of the screen buffer associated with `windowFileVar`.

generate a signal

kill (unix)

SYNTAX

```
#include <stdio.h>
int kill(pid, sig)
int pid, int sig;
```

DESCRIPTION

This generates a signal. Use `getpid` to determine the right value for `pid`.

RETURN VALUE

0 if successful, -1 otherwise (`errno` contains the actual error number).

set the current *stdio* window

setwindow (stdio)

SYNTAX

```
#include <stdio.h>
void setwindow(windowFileVar)
FILE *windowFileVar;
```

DESCRIPTION

The specified `FILE` variable becomes the "current" window. The functions `Set_Echo`, `Set_Tab`, `gotoxy`, `getxpos`, `getypos`, and `wputs` implicitly apply to the current window.

(Functions which implicitly apply to `stdout`, such as `printf`, or to the console, such as `cprintf`, are not affected by which window is current.)

Initially, `stdout` is the current window. `fopenw` does not make the new window the current window.

signal (unix)

SYNTAX

```
#include <stdio.h>
int (*signal(sig, func))()
int sig;
int (*func)();
```

DESCRIPTION

Setting the `SIGINT` signal allows ⌘C and ⌘. (command-period) to be trapped. Setting the `SIGTERM` signal is just like calling `onexit`. Other signals can be set, but are only generated by explicit calls to `kill`. Each signal is automatically reset to `SIG_DFL` once invoked.

Signals set to `SIG_IGN` are ignored. All signals are initially set to `SIG_IGN`. Signals set to `SIG_DFL` will cause the program to exit, or will break to the debugger if one is present.

RETURN VALUE

The previous function set for `sig`.

handle event in a *stdio* window

StdEvent (stdio)

SYNTAX

```
#include <stdio.h>
Boolean StdEvent(theEvent)
EventRecord *theEvent;
```

DESCRIPTION

Passed a pointer to an event record, this procedure determines if the event relates to one of the windows created with `fopenw`. If so, it handles the event and returns `TRUE`; otherwise it returns `FALSE`. If your application has its own event loop, you should call `StdEvent` after calling `GetNextEvent`, and handle the event yourself only if `StdEvent` returns `FALSE`.

RETURN VALUE

`TRUE` if event relates to stdio window, `FALSE` otherwise.

Stdio_MacInit (stdio)

SYNTAX

```
#include <stdio.h>
void Stdio_MacInit(flag)
Boolean flag;
```

DESCRIPTION

To prevent *stdio* functions from calling `InitGraf`, `InitFonts`, `InitWindows`, `InitDialogs`, `TEInit`, and `InitMenus`, and from setting up the *stdio* menus, etc., call this function with the argument `TRUE` prior to any *stdio* calls. This will allow you to use *stdio* windows without conflicting with your application's own windows and menus.

get version of *stdio* library

std_ver (stdio)

SYNTAX

```
#include <stdio.h>
char *std_ver()
```

DESCRIPTION

This returns a pointer to a string containing the version specification of the *stdio* library. This string is currently "LightspeedC™ Libraries 1.57".

RETURN VALUE

String representation of *stdio* library version.

SYNTAX

```
#include <stdio.h>
void wputs(s)
char *s;
```

DESCRIPTION

This is the same as `cputs`, but output goes to the current window rather than to the console window.

SEE ALSO

`setwindow`

8
Tips from Customer Support

We at THINK Technologies want our users to get the most out of LightspeedC. This means providing comprehensive customer support, something we try hard to do. We discovered soon after our first LightspeedC release that there were a few issues that were not clearly covered by our documentation, so this chapter tries to address some of these.

The following is included in this chapter:

- suggested layouts for various hardware configurations
- a step-by-step illustration of the process of creating a program—the famous Kernighan and Ritchie "hello, world" example
- a list of many frequently encountered problems
- tips and suggestions about avoiding errors and improving code
- a couple of examples of **Grep** usage
- a list of a few books you might find useful

So, before calling THINK Customer Support, please check this chapter to see if your question is answered here. If you still have a situation you can't figure out, don't hesitate to call us at (617) 863–1099—that's what we're here for. Our customer support hours are from 9:00 AM to 5:00 PM Eastern Time.

Recommended Disk Layouts

The following are some suggested disk layouts for various hardware configurations. If you don't have enough room for a big system file and your LightspeedC files, remove unneeded fonts and desk accessories from the system file using Font/DA Mover. If you're using HFS, files may be put into sub-directories within the suggested folders as you see fit.

- For two 400K disks, primarily using the Standard C libraries such as *stdio* for `printf`:

  Disk 1: *System Folder* (including *System* and *Finder*); *LightspeedC™* and *MacTraps* from the LS1.System disk; and the *#includes files* folder and any C libraries you need from the LS2.Libraries disk. Do not copy the *Library Sources* folder: you won't have enough disk space. See Chapter 13 of the *User's Manual* to find out which library a function is in.

  Disk 2: Your project and its source and header files, plus any libraries and include-files you need that would not fit on Disk 1.

- For two 400K disks, primarily using the Macintosh libraries:

  Disk 1: *System Folder* (including *System* and *Finder*); *LightspeedC™*, *MacTraps*, any other Mac libraries and resources you need, and any files you need from the *Mac #includes* folder from the LS1.System disk.

  Disk 2: Your project and its source and header files, plus any libraries and include-files you need that would not fit on Disk 1.

- For one 800K disk or larger HFS volume, primarily using the Standard C libraries such as *stdio* for `printf`, we suggest the following:

  System Folder: *System* and *Finder* (and Imagewriter or Laserwriter files if desired).

  C Folder: *LightspeedC™* and *MacTraps* from the LS1.System disk; and the *#include files* folders and any C libraries you want from the LS2.Libraries disk. If you have an 800K floppy, do not copy the *Library Sources* folder: you won't have enough disk space. See Chapter 13 of the *User's Manual* to find out which library a function is in.

  Project folder: Your project and its source and header files.

  Since you are using HFS, do not put *C Folder* inside the project folder or vice versa.

- For one 800K disk or larger HFS volume, primarily using the Macintosh libraries, we suggest the following:

  System Folder: *System* and *Finder* (and Imagewriter or Laserwriter files if desired).

  C Folder: *LightspeedC™*, *MacTraps*, any other Mac libraries and resources you need, and the *Mac #includes* folder from the LS1.System disk.

  Project folder: Your project and its source and header files.

  Since you are using HFS, do not put *C Folder* inside the project folder or vice versa.

"hello, world" in LightspeedC

Many of our users start out by programming a simple example program from Kernighan and Ritchie's *The C Programming Language*, commonly known as "hello, world". This section explains how to get this example running in LightspeedC.

1. Install LightspeedC on your system. See "Recommended Disk Layouts", above.

2. Run LightspeedC. It will prompt you for a project with a Macintosh Standard File dialog. Click the button marked **New**.

3. You will be asked to create your new project. Call the project *hello project*, and store it on a disk that will have room for it. (About 72K of disk space will be required for this project.)

4. Once the project is created, select **New** from the **File** menu. A window titled "`Untitled`" will appear. Enter into the window the following program:

```
#include "stdio.h"
main()
{
   printf("hello, world\n");
}
```

5. Select **Save** from the **File** menu. A dialog box will appear asking you to name the file. Name it *hello.c* and click the **Save** button. (Your source files names must end in *.c*.)

5. Select **Compile** from the **Source** menu. Your program will compile and will be entered in the project window.

7. Select **Add...** from the **Source** menu. Add the file *stdio* (to obtain access to the `printf` function) and the file *MacTraps* (to let the `printf` function have access to the Macintosh Toolbox), then select **Cancel** to end the **Add...** process.

8. Select **Run** from the **Project** menu. LightspeedC will ask you if you want to bring your project up to date. Click the **Yes** button. LightspeedC will load the libraries in the project, link it and run it. It should display a window named "`console`" with the greeting "`hello, world`" in it.

Congratulations! You're a C programmer!

Some Common Problems

Problem: Link errors. Typically, the message "`not defined: printf`" or something similar will appear in a window titled "`Link Errors`".

Possible Cause

You have not added a necessary library (using **Add...**). If a missing symbol is a Macintosh Toolbox function, it may be that *MacTraps* hasn't been included. The *MacTraps* library must be included in nearly all projects, since it holds the QuickDraw globals and "glue" code that accesses the register-based Macintosh Toolbox functions, as well as the `[Not in ROM]` routines.

Similarly, if you're using any C library routines, you may not have included the library that contains the code for that routine. Check the documentation for the routine in Chapter 13 of the *LightspeedC User's Manual*. You will see the routine name at the top of the page in large bold print. Below that, in parentheses, you will see the name of the library you need. Make sure that that library is added to your project, using the **Add...** command.

Possible Cause

You have made a spelling or capitalization error. C is a *case-sensitive* language, which means that `printf` is not the same as `Printf`. Check to see that the undefined function name is spelled and capitalized correctly and matches the spelling of the function in the manual in Chapter 13 (for C functions) or Chapter 14 (for Macintosh functions).

Possible Cause

You have redefined a Macintosh or C library function as "`extern`". For example:

```
extern void SetRect();
```

Such re-definitions are unnecessary, since Toolbox function definitions are already built in to LightspeedC. If one appears, the linker will try to resolve the reference with a user-defined function. This is fine if what you want to do is replace a standard function, but will cause a linker error if no replacement function is provided. (If you want to supply return type or prototype information for Toolbox calls, see page 16 of this supplement.)

Problem: Running the *MiniEdit* example gives error "`Can't open resource file.`"

Probable Cause

MiniEdit needs the resources in the *new project.rsrc*. In order for LightspeedC to add the resources to the MiniEdit project, the project should be named *new project*. If you're running under HFS, both files must be in the same folder.

Problem: The Link Error window says "`Code segment too large`". This error occurs when one of the segments of your project contains more than 32K of code.

Solution

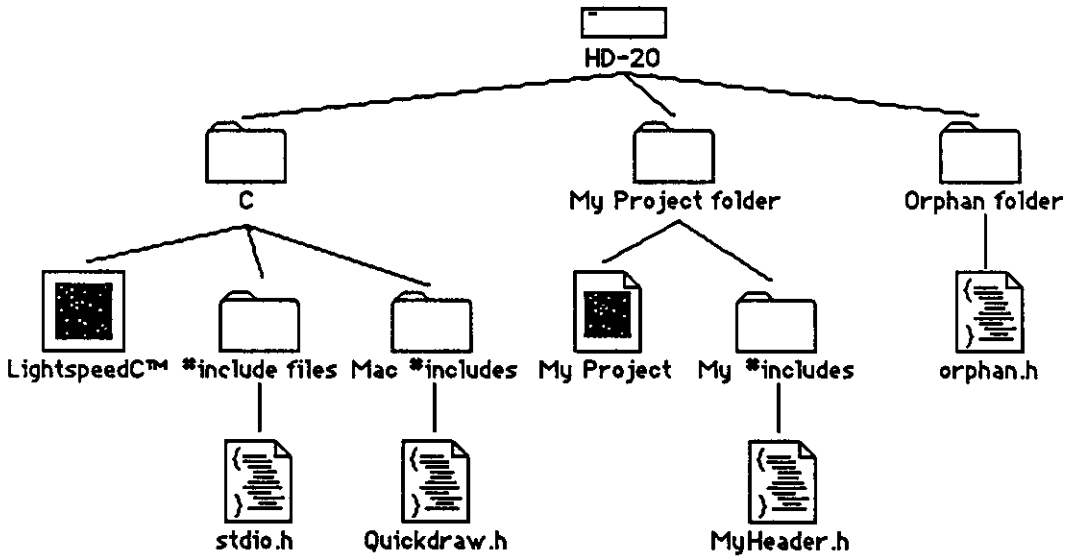
Move files out of the too-large segment until it contains less than 32K of code. See page 5-3 of the *User's Manual* for instructions on moving files between segments. You can check the amount of code contained in a segment by selecting a file in the project window and selecting the **Get Info** command from the **Source** menu.

Problem: Can't find an include-file.

Possible Cause

You have an HFS machine and the specified file is not in the correct search path. The basic search rules are explained on page 4-4 of the *User's Manual* and page 4 of this supplement. In this release of LightspeedC, these search rules have been expanded under HFS to include "all descendents of the folder being searched".

Let's assume that you've set up your HFS file system like this:



The file *Quickdraw.h* can be referenced by the include directive:

```
#include <Quickdraw.h>
```

or the directive

```
#include "Quickdraw.h"     /* searches the project tree
                              first */
```

The file *stdio.h* can be referenced by the directive:

```
#include <stdio.h>
```

or the directive

```
#include "stdio.h"         /* searches the project tree
                              first */
```

The file *MyHeader.h* can be found with the directive:

```
#include "MyHeader.h"
```

but not with the directive

```
#include <MyHeader.h>      /* the project tree will not
                              get searched for files
                              enclosed in <> */
```

However, you can only reference the file *orphan.h* by specifying an absolute path name:

```
#include "HD-20:orphan folder:orphan.h"
```

Problem: Bomb message saying "ID=02" (Odd Address Error) when you run your program. There are many possible causes here; some are very complex. A few common causes are:

Possible Cause

QuickDraw was not initialized. If your program does graphics using the Macintosh's QuickDraw library, the drawing environment must be initialized with a call to `InitGraf`. (See the "QuickDraw" chapter of *Inside Macintosh.*) Note that if you are using the *stdio* library, this initialization is done for you unless you call `Stdio_MacInit(TRUE)` before any calls to stdio routines. In any event, you should not initialize QuickDraw twice!

Possible Cause

You have referenced a null handle or pointer. A null handle can result from several things. First, there may be a missing resource file. If your program uses a resource file that is not found, a `GetResource` (or `GetNewMenu`, `GetNewWindow`, etc.) call will return a null handle. Make sure that if you have an auxiliary resource file it is on the same disk (and in the same folder as your project file under HFS), and that it is named *<your project file name>.rsrc*. (See page 4 of this supplement.)

Second, you may not have enough memory for a `NewHandle` or `NewPtr` call, resulting in a null return value. Lastly, you may have not initialized a pointer variable.

Possible Cause

You've called a Toolbox routine with bad or inappropriate data. For example, you are likely to crash if you call `DisposHandle` with a handle to a resource. Use `ReleaseResource` instead.

Problem: Can't load library.

Possible Cause

You have an old version of the *HD-20* file used in HFS systems running with old ROMs. If you have old ROMS (i. e. a 512K Mac which has not been upgraded) and you use an HD-20, you have to start your machine with a floppy disk which "kick-starts" your HD-20. On the floppy disk there is a system file named *HD-20*. The original version of this file (v1.0, created in September 1985) has a bug which will cause LightspeedC not to find libraries if they are in folders. You should get a newer version of the *HD-20* file from your dealer. Until you do, libraries must be on the root volume of the HD-20 to work correctly.

Problem: Data segment too large or stack frame too large. This error occurs when your project has more than 32K of global data or a function has more than 32K of local data.

Solution

If you want to be able to access global or local memory larger than the 32K limit imposed by the Macintosh/MC68000 architecture, you must allocate the memory dynamically via a pointer or handle. Because the C language blurs the distinction between arrays and pointers, you can then use the pointer with an array notation to access elements in dynamic memory. (Refer to a good C language manual, such as Kernighan and Ritchie's *The C Programming Language* for more details.)

Example:

```
#define SIZE          100000
#define SIZE2D        100

int BigBadArray[SIZE];                  /* a too big array */
int *LooksLikeAnArray;
int BigBad2DimArray[SIZE2D][SIZE];      /*too big 2D array*/
int *LooksLikeA2DArray[SIZE2D];         /*array of pointers*/

proc()
{
     register long i;

     /* allocate "array"*/
     LooksLikeAnArray = (int *)NewPtr(sizeof(int)*SIZE);
     LooksLikeAnArray[60000] = 5;   /* can index it just
                                       like an array */
     /*   allocate 2D "array"*/
     for (i = 0; i < SIZE2D; i++)
          LooksLikeA2DArray[i] =
               (int *)NewPtr(sizeof(int)*SIZE);

     LooksLikeA2DArray[10][20] = 7; /* can index it
                                       just like an
                                       array */
}
```

Problem: `printf` and `scanf` don't seem to work correctly with long or double data.

Solution

A common misconception about `printf` is that `printf` "knows" about its arguments. If you pass a long expression to `printf`, you must specify in the format string that you want a long expression to be printed. Example:

```
int anInt;
long aLong;
printf("long is %ld, int is %d\n", aLong, anInt);
```

In `scanf`, the same goes for floats and doubles:

```
float aFloat;
double aDouble;
printf("Input a double and a float:");
scanf("%lf %f", &aDouble, &aFloat);
```

Problem: "I included *stdio.h* in my source file, so why do I get a link error for `printf`?"

Solution

Many novice users (and not-so-novice users!) get confused about the difference between "header" files and "library" files. *Header files* (which conventionally have names which end in *.h*) contain source statements: definitions and declarations which allow the compiler to make sense of source code calls to a library function. *Libraries* contain the actual object code for the functions themselves, which the linker references in building the project. Including a header file is for the compiler only; it tells the linker nothing. `#include`-ing a library in a source file is an error.

Some Useful Tips

Arrays

To work with arrays greater than 32K, see "Problem: Data segment too large" on page 51 of this supplement.

Typically, arrays are accessed within loops, so small gains in efficiency can be greatly magnified. Here are a couple of tips:

- Indexing of arrays whose elements are a size that is a power of two is more efficient if you declare the index to be of type `register long`. For arrays of `doubles`, or `structs` whose size is not a power of two, use `register int`. Example:

```
int array[10];
register long FastIndex;  /* int is not as fast */
int SlowIndex;

array[SlowIndex] = 5;     /* Slower */
array[FastIndex] = 5;     /* Faster */
```

- Sequential access of arrays can be done more efficiently with pointers. Further optimization can be done with inline assembly, if desired. Example:

```
int array[MAXSIZE];
register int *p;
register long i;

/* Slower: index calculations (multiplication and
   additions) are done */
   for (i = 0; i < MAXSIZE; i++)
      array[i] = i;

/* Faster: index calculations are avoided, one addition is
   done */
   for (i = 0, p = array; i < MAXSIZE; i++, p++)
      *p = i;
```

Inline Assembly

Inline assembly can be tricky if you are not familiar with assembly language. It can be especially dangerous if you are used to thinking in terms of high-level languages. The following problems described are not specific to LightspeedC; they are common to assembly language in general.

- Truncation of constants is not detected. Examples:

```
add.W   #0x12345678, D0    ; gets truncated to 0x5678
dc.W    0x12345678         ; gets truncated to 0x5678
```

- Don't forget the # sign when using an immediate constant. The result will be very different than what you intended.

```
extern int MemErr : 0x220;      /* declare MemErr low
                                   memory global */

asm{
   MOVE.W  0x220, D0  ;moves contents of location 0x220
                      ;(MemErr) into D0
   MOVE.W  MemErr, D0 ;same way of writing the above
                      ;symbolically
   MOVE.W  #0x220, D0 ;moves the value 0x220 into D0
   MOVE.W  5, D0      ;WRONG: this will cause an odd address
                      ;error!
   MOVE.W  #5, D0     ;Right
}
```

- The assembly directive `DS`, which would create global storage space, is not allowed. Use C variables instead. If you use the directive `DC` instead to declare storage space, storage will be allocated in your code segment. Most of the time, this is not what you want.

- Be sure to use the right-sized instruction when referring to variables. Example:

```
function()
{
   int anInt;
   int GetsTrashed;

   asm{
      move.L  #3, anInt ;WRONG: will overwrite
                        ;variable GetsTrashed
      move.W  #3, anInt ;RIGHT: Word size matches int.
   }
}
```

- Quick instructions are automatically used whenever possible. Therefore, you don't have to painstakingly hand-optimize MOVE to MOVEQ instructions. Examples:

```
MOVE.L  #4, D0          ; This instruction assembles as...
MOVEQ   #4, D0          ; ...which is faster and smaller.
ADD.L   #aConstant, D0  ; You don't even have to know if
                        ; aConstant is the right size for
                        ; a Quick instruction.
                        ; An ADDQ is generated if 1 <=
                        ; aConstant <= 8
```

- Make sure your code doesn't run into your DC's. Example:

```
            MOVE.W  @value1, D0
@value1:    DC.W    -1      ; WRONG: the Macintosh will try to
                            ; execute this "instruction"
                            ; following the MOVE.W instruction

            MOVE.W  @value2, D0
            BRA.S   @1      ; RIGHT: branch around inline data.
@value2:    DC.W    -1
@1:
```

Floating Point Arithmetic

The LightspeedC type `double` actually corresponds to the SANE type Extended. This type takes up 10 bytes of storage, but is the fastest floating point type. Conversely, the LightspeedC type `float` actually corresponds to the SANE type Single. This type takes up 4 bytes of storage, but is the second fastest floating point type. The LightspeedC type `short double` actually corresponds to the SANE type Double. This type takes up 8 bytes of storage, but strangely enough is the slowest floating point type. If you want speed, stick with `double`. If you want to save space and are willing to sacrifice some accuracy and speed, then use `float`. If you are willing to sacrifice more accuracy for speed, consider using the fixed math library.

Some other suggestions:

- In floating point operations, use floating point constants. Example:

```
double1 = double2 * 5;    /* Slower: 5 has to be converted
                             to double at runtime */
double1 = double2 * 5.0;  /* Faster: 5.0 doesn't have
                             to be converted */
```

- Multiplication is faster than division. Example:

```
double1 = double2 / 5.0;       /* Slower */
double1 = double2 * 0.2;       /* Faster */
```

And of course addition is faster than multiplication by 2:

```
double1 = double2 * 2.0;       /* Slower */
double1 = double2 + double2;   /* Faster */
```

- Use the +=, *=, etc. operators wherever possible. Due to the way SANE works, storing the result of a floating point operation into an operand's memory is more efficient. Example:

```
double result, a, b, c;

result = a * b * c;    /*  Slower and larger */
result = a;            /*  Faster and smaller: avoids
                           temp variables */
result *= b;
result *= c;
```

- Evaluate floating point constant expressions. LightspeedC does not pre-calculate constant floating point expressions at compile time. (To do so, LightspeedC would have to make assumptions about what the SANE environment will be at runtime.) Example:

```
double1 *= 4.5 + 2.4;   /* Slower: the addition is
                           done at run time */
double1 *= 6.9;         /* Faster */
```

- It's better to return a `double` variable than a `double` expression. Example:

```
return(aLocalDouble + 4.0); /*Larger and slightly slower*/
aLocalDouble += 4.0;        /*Smaller and slightly faster*/
return(aLocalDouble);
```

Efficiency

- `SetRect`, `SetPt`, etc., take more time than setting up the coordinates yourself, because of the overhead of the Macintosh trap dispatcher. However, significant improvements will be noticed only if you are executing lots of them.

- Use register variables whenever a local variable or parameter is used more than a few times or is heavily used in a loop.

- The fastest way to do something N times is:

```
register int i; /* will not work if i is unsigned! */

i = N;
while (--i >= 0)
   something();
```

- Bitfield variables save storage space, but it takes more time and code space to access the data in bitfield variables larger than one bit. However, the setting and clearing of single bits uses the same amount of code space.

Memory Management

- If you are dereferencing `Handles`, then make sure that the memory the `Handle` points to won't move while using it. Otherwise, `HLock` it (and `HUnlock` it later). Example:

```
HLock(aTEH);
/* Without the previous HLock, the next two calls may
   not always work*/
/*(1) printf calls Memory Mgr: */
printf("first character in text handle is %c\n",
     **(**aTEH).hText );
/*(2) Handle dereference on the left is done BEFORE call:*/
(**aTEH).hText = (Handle)NewHandle(somesize);
HUnlock(aTEH);
```

- To increase your zone (the memory available to your program) to the maximum, call the procedure `MaxApplZone()`.

Miscellaneous

- Don't forget to make all necessary Macintosh initialization calls. In LightspeedC, you must make all the necessary calls yourself. (Some other Macintosh C compilers do some initializations for you.)

  An exception to this occurs when using the *stdio* library. Initializations are automatically done the first time a *stdio* procedure (that needs it) is called. You can turn this off if you are doing your own initializations—for example, if you are using the standard output window for debugging. The automatic initialization is for users of LightspeedC who want to have the more traditional, non-Macintosh C environment. See "Enhancements to Standard C Libraries" on page 29 of this supplement.

- The `Byte` and `Char` types in Pascal actually correspond to the `int` type in C. The three places that this has caused our users problems are the functions `ATPOpenSocket`, `GetItemMark`, and `GetItemIcon`. In these functions, a `Byte` argument is passed by reference. The proper way to do this in C is to pass a pointer to an `int`.

- TMON doesn't know that the `DebugStr` trap (`0xABFF`) takes a string argument and doesn't clean up the stack correctly. Don't use `DebugStr` if TMON is your debugger.

Some Grep Examples

Here is a grep example useful for converting assembly language to LightspeedC inline:

- To convert

```
symbol  equ  (expression+4)          ; a comment
```

to

```
#define symbol (expression+4)        /* ; a comment */
```

grep-search for

```
\(\<..*\>\)[space tab]*\<equ\>\([^;]*\)\(.*\)
```

and replace with

```
#define \1 \2 /* \3 */
```

Explanation:

- `\<..*\>` matches a symbol.
- The surrounding `\(` and `\)` allows the symbol to be in the replacement string as `\1`.
- The `[space tab]*` matches any number of spaces or tabs between the symbol and the key word `equ`.
- `\<equ\>` matches the word `equ`. It will not match `equ` if it is part of another word, for example `equal`. `\<equ\>` is not surrounded by `\(` and `\)` because it will be thrown away in the replacement string.
- `[^;]*` matches an expression formed by any number of characters up to but not including a `;` (semi-colon).
- The surrounding `\(` and `\)` allows the expression to be remembered in the replacement string as `\2`.
- The `.*` matches the comment which is the rest of the line.
- The surrounding `\(` and `\)` allows the comment to be remembered in the replacement string as `\3`.
- If there was no ; (semi-colon) in the line, then `\2` will consist of everything after the `equ` to the end of the line and `\3` will be an empty string.
- To insert a tab in the **Find...** dialog, type ⌘Tab (Command-Tab).

- To convert `$HHHH` to `0xHHHH`, where `H` is a hexadecimal digit, grep search for

  ```
  $\([0-9A-Fa-f][0-9A-Fa-f]*\)
  ```

  and replace with

  ```
  0x\1
  ```

  Explanation:
  - `$` matches a `$`. `[0-9A-Fa-f]` matches one hex digit.
  - `[0-9A-Fa-f][0-9A-Fa-f]*` matches one or more hex digits. (The grep search string `[0-9A-Fa-f]*` matches zero or more hex digits.)
  - The surrounding `\(` and `\)` allows the hex digits to be remembered in the replacement string as `\1`.

If you like, save complicated grep search and replace strings in a file so you can copy and paste them into the **Find...** dialog box.

Further Reading

- *Inside Macintosh* is, of course, vital.

- If you want to make your program smaller or run faster, an excellent place to start is by reading *Writing Efficient Programs* by Jon Louis Bentley (Prentice-Hall 1982). In most compute-bound programs, about 50% of the time is spent in about 4% of the code. Most code optimizations don't optimize as much as you might think. Although the suggestions in here may not produce large improvements, they are relatively cheap to put in.

- We highly recommend the book *How to Write Macintosh Software* by Scott Knaster (Hayden Book Company 1986) for many useful tips, tricks and warnings about programming the Macintosh.

- The book *Using the Macintosh Toolbox with C* by Jim Takatsuka et al. (SYBEX 1986) is a good introduction. However, be aware that it emphasizes Consulair's Mac C rather than LightspeedC, so some examples will have to be modified (see Chapter 10 of the *User's Manual* for notes about translating from other C's to LightspeedC).

Appendix A
Corrections to the *User's Manual*

page 1-1:
The compile times claimed should read:
"A typical 15,000 line program in 20 source files (XLISP V1.4) takes slightly more than three minutes to compile on a 10-megabyte HyperDrive™. (Over half of this time is spent in disk access, so the actual compile time is about 90 seconds.)" (The raw compile speed numbers are still accurate —the compiler actually compiles more than 22,000 lines due to `#included` files.)

page 3-7:
The manual says "pull down the **Source** menu and select **Run**". It should read "pull down the **Project** menu and select **Run**".

page 10-7:
The line
```
mode = "w+";
```
should be
```
mode = "a+";
```

page 13-1:
The name of the Unix memory allocation functions library is *storageu*, not *storageu.lib*.

page 13-14:
In the `calloc` example, the line 3 lines from the bottom should read:
```
if ( (newmonth = (date *) calloc(31, sizeof(date)) )
```

page 13-24:
The function `_closeall()` is called automatically on all program terminations that call the *stdio* file functions. `ExitToShell` from Macsbug will also call `_closeall()`.

page 13-23:
The sentence
`fclose()` closes a file when given a file descriptor number.
should be
`fclose()` closes a file when given a file pointer.

page 13-32:
The `#include` statement should be
```
#include "ctype.h"
```

`isascii(c)` is TRUE if `c` lies in the range of `0` to `127 (0x7f)`.
`iscntrl(c)` is also TRUE if `c == 127 (0x7f)`.
`ispunct(c)` is also TRUE if `c == '@'` or `c == '^'`.
`isspace(c)` is also TRUE if `c == 0x3`, the value of the Macintosh Enter key.

Note that the actual `__ctype` data table is in the file *stddata_ctype.c* which is part of the *stdio* library. You can add either (but not both) files to your project to get the table.

page 13-33:
The line
```
_space_  178
```
should be
```
_space_  128
```

page 13-39:
Buffers do not have to be flushed before closing files. You may want to use `fflush()` after writing to a file or doing a `printf` to the printer.

page 13-57:
`getchar()` and `fgetc(stdin)` echo their output depending on the state of echo. See `Set_Echo()` in the *User's Manual.*

page 13-85:
In the `printf` function example, the format specifier is given as `"%03 d"`. It should read `"%03d"` (without a space between the 3 and the d).

page 13-110:
In the `scanf` description of right bracket (`[`), the last two paragraphs on the page are wrong. `scanf` does not support the "range of consecutive characters" feature described, nor does it allow a right bracket (`]`) in the scanset.

page 13-110:
If scanf() is being used to input numbers of type `float`, `short double`, or `double`, the specifications should be `%f`, `%hf`, `%lf`, respectively.

page 13-114:
The functions `setjmp` and `longjmp` and the variable `jmpbuf` are ascribed to the *unix* library. They are really in the library *setjmp.lib*. Include *setjmp.h* to get access to the definition of these items. In this release, `setjmp` and `longjmp` are also included in *stdio*, so don't include both *setjmp.lib* and *stdio*.

page 13-123:
At the end of this appendix is a replacement manual page.

page 13-168:
Using `ungetc()` to push back EOF has no effect on the stream. EOF is returned.

page 13-169:
The declaration should be
```
int ungetch(c);
int c;
```

page 14-27:

The declaration for `GetItemMark` and `GetItemIcon` should be

```
void GetItemIcon(menu, item, iconNum)
MenuHandle menu;
int item;
int *iconNum;      /*  NOT Byte *iconNum */

void GetItemMark(menu, item, markChar)
MenuHandle menu;
int item;
int *markChar;     /*  NOT char *markChar */
```

This change is due to the fact that, in Pascal, the `Byte` and `Char` types actually correspond to the C type `int`, unless a variable is explicitly packed in a record or an array. In the LightspeedC header file *MacTypes.h*, `Byte` is actually defined to be `unsigned char`, because `Bytes` are usually packed.

page 14-45:

The correct spelling is `UnloadSeg()`, not `UnLoadSeg()`.

v1.02

get an argument from a string

stcarg (unix_strings)

SYNTAX

```
#include <strings.h>
int stcarg(s,b)
char *s, *b;
```

DESCRIPTION

`stcarg()` scans the text string `s` until a character in the break string `b` is found or until it reaches the end of `s`. It returns the length of the sub-string prior to the break character. If no match was found, the length of `s` is returned. Anything in the string enclosed in single or double quotes will not match characters in the break string. A backslash in either string is treated as an escape character.

EXAMPLES

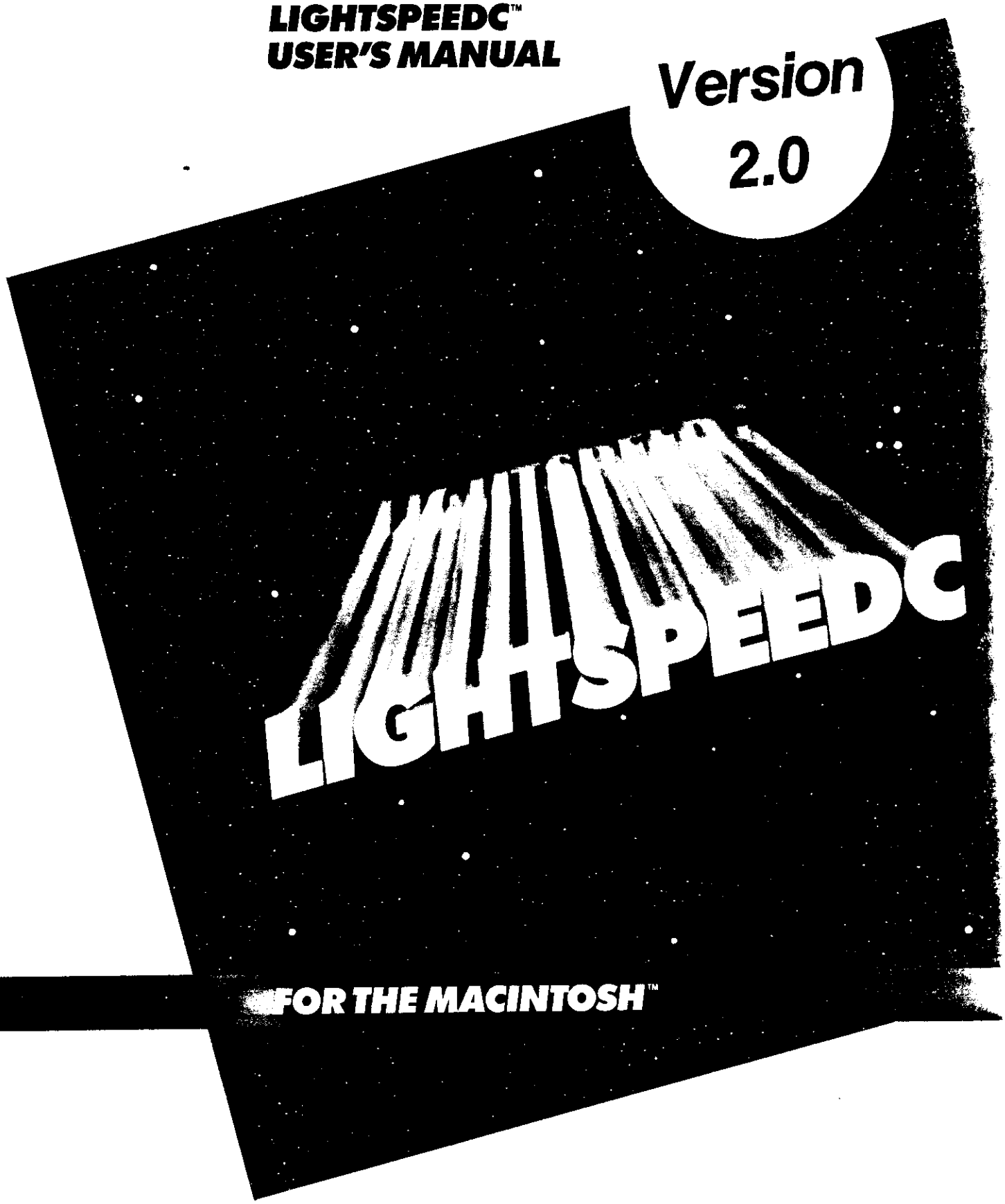
```
stcarg("abcde","12345")        /* returns 5 - no match */
stcarg("aabcd","aeiou")        /* returns 0 */
stcarg("aeiou","aabcd")        /* returns 0 */
```

RETURN VALUE

the length of the sub-string prior to the break character. If no match was found, the length of `s` is returned.

LIGHTSPEEDC™ USER'S MANUAL

Version 2.0



THINK
TECHNOLOGIES, INC.

LightspeedC™

User's Manual

(Version 1, First Edition)

THINK
TECHNOLOGIES, INC.

420 Bedford Street • Suite 350 • Lexington • Massachusetts 02173

First Printing March 1986.
Printed in the United States of America.

The LightspeedC programming environment was conceived by Michael Kahl and Andrew Singer, and was developed at THINK Technologies by Michael Kahl with assistance from Jon Hueras.

The Standard C Libraries for LightspeedC were developed by Robert Alpert, Steve Adams, and Paul Garmon.

The Software was tested by Clem Wang, with assistance from Paul Garmon, Robert Alpert, and Rick Tompkins.

The User's Manual was created by Paul Garmon, Steve Adams, Steve Stein, Michael Kahl, Andrew Singer, and Clem Wang; and by Tim O'Reilly and John Strang of O'Reilly & Associates, Inc., Newton, MA.

The project was managed by Andrew Singer with the assistance of Steve Stein, Steve Adams, and Fleet Hill.

Special thanks to our early users, Phil Di Bello, Steve Lawrence, Mike Boich, Kerry Lynn, and the ingenious Steve "the Finder" Capps.

CONTENTS

PART ONE: USING *LIGHTSPEEDC*

1 Introduction 1–1

2 Installing LightspeedC 2–1

3 Getting Started 3–1
- Project Organization 3–1
- Before You Start 3–2
- Starting LightspeedC 3–3
- Editing 3–6
- Compiling and Running the Project 3–7
- Loading Libraries 3–11
- Debugging 3–12
- Running the Project, Again 3–12
- Building an Application 3–13
- Closing a Project and Quitting LightspeedC 3–13
- Where to Go From Here 3–13

4 More on Editing 4–1
- Introduction 4–1
- Moving Around in a File 4–2
- Changing the Size of the Window 4–2
- Working with Multiple Files 4–3
- A Word About File Specification 4–4
- Editing Text 4–5
- Program Indentation 4–5
- Search and Replace Functions 4–6
- Search Options 4–8
- Searching Through Multiple Files 4–8
- Printing Your Files 4–9
- Saving Your Edits 4–10

5 Advanced Features 5–1
- Make: Compiling Files *en masse* 5–1
- The Confirm Auto-Make Option 5–2
- Segmentation 5–3
- Setting the Project Type 5–4
- Creating Libraries 5–5

6 Menu Reference 6–1
Introduction 6–1
The Menu 6–1
The File Menu 6–2
The Edit Menu 6–6
The Search Menu 6–7
The Project Menu 6–10
The Source Menu 6–12
The Options Menu 6–15

PART TWO: PROGRAMMING AT *LIGHTSPEED*

7 Debugging 7–1
Introduction 7–1
If You've Never Used Macsbug Before 7–1
Displaying and Modifying Memory 7–3
Controlling Your Program: Breakpoints and Stepping 7–5
Tracing 7–7
The Heap 7–7
Terminating the Program 7–9
Debugging a LightspeedC Program 7–9
Tracing Through switch Statements 7–10
Advanced Macsbug Commands 7–11

8 Using Resources 8–1
Introduction 8–1
Using a Separate Resource File 8–1
Combining Resource Files 8–1
An Example 8–2

9 Running at Lightspeed 9–1
Introduction 9–1
C Calling Conventions 9–2
Pascal Calling Conventions 9–4
Interfacing with Assembly Language 9–7
Desk Accessories and Device Drivers 9–8
Code Resources 9–12
The Components of a LightspeedC Program 9–15

10 Achieving Lightspeed 10–1
Introduction 10–1
Sizes of Numbers 10–2
Passing a struct as an Argument 10–3
Passing a Point as an Argument 10–3
Identifier Length and Capitalization 10–4
Runtime Environment 10–5
Use of Assembly Language 10–5
Compiler Issues 10–6
Converting from UNIX 10–6

11 Epilogue 11–1
A Warp Drive For C Developers 11–1

PART THREE: LANGUAGE REFERENCE MANUAL

12 C Language Reference 12–1

13 Standard C Libraries 13–1

14 Calling Sequences for Macintosh Toolbox Routines 14-1

PART FOUR: APPENDICES

A MacsBug Reference A–1

B RMaker Reference B–1

C ResEdit User's Guide C–1

D Compiler Error Messages D–1

E Data Sheet E–1

F Benchmarks F–1

G The Code Profiler G-1

INDEX TO FUNCTIONS

INDEX

PART ONE

USING *LIGHTSPEEDC*

1
Introduction

LightspeedC is more than just another C compiler for the Macintosh. It is a quiet revolution in programming. It is not just that LightspeedC is fast—it is so fast that it will change the way you work.

Typically, program development proceeds in an iterative fashion. A portion of a program is coded, then it is compiled, linked with other portions of the program, and tested. Changes are made incrementally—at least in theory. In reality, programmers tend to "batch" changes, because the program build time (the time it takes to compile, link, and launch a new version of the program) is prohibitive. Especially when working with large programs, we tend to make a group of changes before recompiling and linking the program. And of course, when one of those changes introduces bugs into other parts of the program, it is that much harder to figure out which one caused the problem.

With LightspeedC, program build time is short enough that you will find yourself making a single change to a program, then compiling and running the program to see how it works. No more coffee breaks while you wait for the application to be rebuilt—you can edit a program, compile, link and run it, and be back in the editor while most compilers and linkers are still grinding away.

In effect, you have a compiler that allows interactive program development. Sure, interpreters have provided this for years—with the penalty of slow performance when it actually comes time to run the program. LightspeedC is a production class compiler, that produces compact, optimized code that runs as fast, or faster, than the code produced by existing products.

You may have a hard time believing that LightspeedC can be fast enough to make a real difference. We don't plan to argue the point—the product itself will do that. However, we do want to introduce some of the features of LightspeedC that contribute to the "quiet revolution."

1. As already mentioned, the compiler itself is fast. Its raw, RAM-based compile speed varies between 250 and 500 lines/second, depending on the expression density of lines. A typical 15,000 line program in 20 source files (XLISP 1.4) takes slightly more than two minutes to compile on a 10 megabyte HyperDrive. (Over half of this time is spent in disk access, so the actual compile time is about 50 seconds.)

2. Link time is less than 5 seconds, even for a large application. Most of the time, modifying a large program only requires recompiling a few files at a time, so it is the link time that becomes the dominant factor in turnaround. The LightspeedC linker is blindingly fast.

3. LightspeedC provides an integrated program development environment. It uses a special data structure known as a *project* to track all of the source files that make up a program, and to contain all of the associated object code and libraries. The editing, compiling, linking, and

program execution phases of program development are all accessible from the project. What's more, they are closely coupled. When the compiler detects an error, the editor is automatically invoked, with the edit caret at the start of the offending source line. Likewise, if you try to run the program without recompiling an edited source file, you are warned that the project is not up to date. By setting project options, you can even arrange to have modified files automatically compiled, updated and relinked when you want to run the program, allowing you to simply edit your sources and then run the changed program.

4. LightspeedC provides an automated "make" facility. As suggested above, this facility keeps track of changes in program source files, `#include` files, and library files so that all program builds are made correctly. This eliminates build errors due to mismatched library and `#include` files, and human errors in constructing the inputs to manual make facilities.

So where's the catch? It may be hard to believe, but there isn't one.

Compact code?	Yes
Execution speed?	Excellent
Complete language implementation?	Yes
Unix compatibility?	Yes
All Macintosh Toolbox and OS calls supported?	Yes
Links to assembly language?	Yes
Library source code included?	Yes

You don't have to take any of this on faith. We've kept this introduction brief so that you can get started using the product. Demo programs are included that you can compile immediately. They are included on the distribution disk in source form only, so that you can get an immediate sense of how LightspeedC works by compiling them as you read the manual.

The manual is divided into four parts:

Using LightspeedC. This part is the "user's guide" for the product. It describes the user interface—how to invoke the various commands and facilities provided by LightspeedC. If you are familiar with the Mac, you may only want to skim this section for the highlights.

Programming at Lightspeed. This part describes issues related to actually using LightspeedC for program development. It includes a discussion of issues such as interfacing with assembly language or Pascal, debugging, converting Macintosh Development System (MDS) *.Rel* files to LightspeedC libraries, and various technical tips.

Language Reference Manual. This part includes a description of the actual C language implemented by LightspeedC, with an emphasis on extensions to the standard. It also includes a complete reference to all of the libraries supplied with LightspeedC, and C interfaces for the Macintosh Toolbox.

Appendices. Appendices include quick reference material, additional documentation on Apple programming tools distributed with LightspeedC but not actually a part of it (such as the Macsbug debuggers and the ResEdit resource editor). And, of course, there is a complete index.

We have made a couple of assumptions in writing this manual:

1. You are a competent C programmer. No attempt is made to teach the C language. If you are not familiar with C, you should obtain a copy of a C language tutorial, and Harbison & Steele's *C: A Reference Manual* (Prentice-Hall, 1984), the most up-to-date reference work on the C language. Kernighan and Ritchie's book, *The C Programming Language* (Prentice-Hall, 1978), the original work on C by the developers of the language, is also a useful reference work, although at this point it is slightly out of date. (Nonetheless, in many ways it defines the current standards for the language.) Steve Kochan's *Programming in C* (Hayden, 1985) is one of the better introductory works on C.

   This manual includes reference material covering all extensions to the C language supported by the LightspeedC programming environment. These extensions are presented in the form of addenda to the relevant sections of Kernighan and Ritchie's book (henceforth referred to as *K&R*). In addition, all libraries supplied with LightspeedC are completely documented in a reference format.

2. You are familiar with programming the Macintosh. This is not essential for writing portable C programs. (It used to be that programmers developed applications for the Mac on a larger machine, and ported them to the Mac only when they were substantially complete. With LightspeedC, you may want to develop initial versions of programs for other machines on the Mac.) However, it is essential if you are planning to write applications for the Mac.

   In Part Two of the manual, we have tried to highlight some of the issues involved in programming the Macintosh in C. Part Three includes a reference section documenting the C interfaces for all of the Macintosh Toolbox calls. This will be enough to get you started. However, you should have a copy of Apple's manual *Inside Macintosh*, Volumes 1, 2, and 3 (Addison-Wesley, 1986), or Steven Chernicoff's *Macintosh Revealed*, Volumes 1 and 2 (Hayden, 1985). *Using the Macintosh Toolbox with C*, by Jim Takatsuka, Fred Huxham and David Burnard, (Sybex, 1986) is an excellent introduction to C on the Macintosh. This book is oriented to Consulair C, which differs in some ways from LightspeedC.

2
Installing LightspeedC

LightspeedC can be run on a Macintosh with 512K of memory and one 400K disk drive. Because of the inconvience of disk swapping, and the space required for the extensive libraries provided, it is strongly recommended that you have at least two 400K disk drives or an 800K drive. A hard disk or a RAMdisk used with additional memory will substantially improve programming throughput. LightspeedC will take advantage of as much memory as is available. For extremely large projects, a megabyte of memory may be necessary. LightspeedC itself has nearly a megabyte of source code in over 80 files and was developed using LightspeedC on a 512K Macintosh equipped with a 10 megabyte GCC Hyperdrive.

LightspeedC will work with either the old Macintosh 64K ROMs or the new Macintosh 128K ROMs.

The software is distributed on three disks. The first disk contains a System Folder and LightspeedC itself. The second disk contains libraries and header files. (All of the standard C libraries, Unix compatibility libraries, and MacTraps libraries are distributed as project documents. This allows you to modify the supplied sources, and in addition, allows LightspeedC to make intelligent linking decisions when including these libraries in other projects.) The third disk contains the LightspeedC utility RelConv, and the Apple utilities RMaker, ResEdit, and the Macsbug debuggers, as well as LightspeedC demo programs. See the *Release Notes* file on the first disk for a detailed description of the contents of the distribution disks.

If you are working without a hard disk, your first step should be to create backup copies of the distribution disks. Write protect the distribution disks, and never use them except to make backup copies (for your own use). The software is not copy protected.

It is preferable to put the Macintosh and C `#include` files supplied with LightspeedC (the folders Mac #includes and Standard #includes) on the same volume as LightspeedC itself. If you are using the new Hierarchical File System (HFS), they should go in the same directory as LightspeedC. This will enable the macro preprocessor to find them automatically when they are referenced by any of your programs, without the need to specify a volume (or directory) name in the reference. (If you don't want to put them on the same volume, LightspeedC will recognize volume names or HFS directory path names specified in the `#include` statement.)

It is also recommended that you put Macsbug on your boot disk, in case you want to debug or recover from simple crashes of programs you are developing and return to Lightspeed without rebooting. See Chapter 7 for information on installing Macsbug.

If you are working with a hard disk, copy the files on both disks to your hard disk by dragging their icons to the hard disk in the usual Macintosh fashion. Even though you have copied the files to the hard disk, it is still a good idea to make backup floppies of the distribution disks.

3
Getting Started

Project Organization

One of the secrets of LightspeedC's amazing speed is its unique way of tracking the files that collectively make up a software development project. The central organizing concept is the *project document.*

In a conventional language development system, there are source files, object files, a link-control file, an executable image, and possibly other administrative files. With LightspeedC, you have only source files and a project document that contains a list of all the files and/or libraries belonging to that project. It also contains compiled code for the files in the project as well as information about relationships between those files. While editing using the LightspeedC text editor, the project document maintains information about which files need to be recompiled because of changes to them or to any of their `#include` files.

This chapter provides a walkthrough of the basic features of the LightspeedC environment. It describes how to create a new project, add files to the project, open files for editing, compile them, and run the project. For those of you in a hurry, we've first provided a quick checklist. If you have time, read the rest of the chapter for a step-by-step description.

Task	What to Do
Start up LightspeedC	Double click on the LightspeedC icon or on an existing project document.
Open an existing project document	Click on **Open** from the dialog box.
Create a new project document	Click on **New** from the dialog box.
Add files to a project	Click on **Add...** from the **Source** menu to add multiple files without compiling. Or select **Open...** from the **File** menu to open a file for editing, then compile it to add it to the project. (You can also use the **Add** command on the **Source** menu to add a file in an edit window that will not yet compile successfully.)
Open a file for editing	If it is in the project window, simply double click on its name to open it. Or click on **Open...** from the **File** menu.

Create a new file	Click on **New...** from the **File** menu. This will open up an *Untitled* edit window, which you can save as a file after editing. A new file can be added to the project by saving it with a `.c` or `.C` extension and then compiling it. If it will not yet compile successfully, you can add it with the **Add** command.
Compile a source file	If the file is in the front editing window, simply click on **Compile** from the **Project** menu, or use the **⌘K** keyboard command. If the file is not open, click once on its name in the project window to select it, then issue the **Compile** command. Note that in LightspeedC's integrated development environment, you never really need to compile an individual source file. It is much more common simply to edit a file, and then **Run** the project. The built-in Auto-Make facility will automatically compile any files that have been updated since the project was last run.
Run a project	Click on the **Run** command from the **Project** menu, or use the **⌘R** keyboard command. If any files need recompiling you will be asked if you want to bring the project up to date.
Exit LightspeedC	Select **Quit** from the **File** menu.

Before You Start

The rest of this chapter contains a walkthrough of some of the most common commands you will be using in LightspeedC. It shows how to create a new project from scratch, using as source material a C version of the MiniEdit program from *Inside Macintosh*. While this demonstration is not rigorously designed as an on-screen tutorial, you should be able to follow it on your system as well as on paper. However, since the organization of the files on your disk will probably not match those shown here exactly, you shouldn't take all of the illustrations literally. In particular, the contents of file selection boxes will probably be different on your system.

This walkthrough assumes that the source files and libraries used in the MiniEdit project are on the same disk as LightspeedC itself.

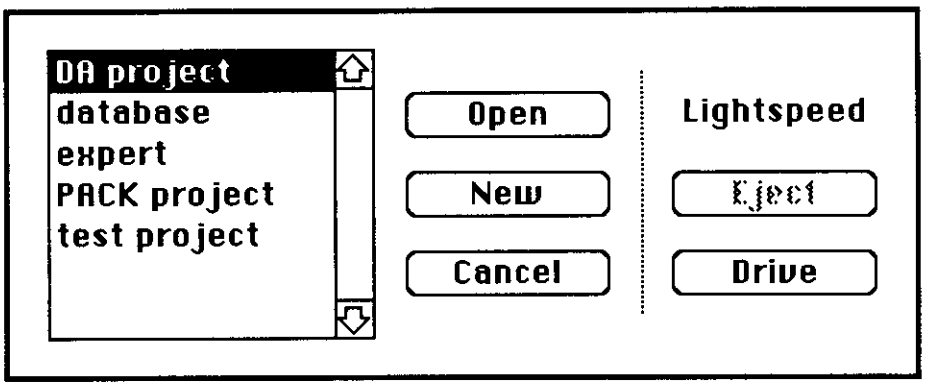
If you are working on a hard disk, there should be no problem, since you should have copied all of the files from the distribution onto your disk. If you are working with two floppy drives, you may need to do a little juggling of files to get what you need from the three distribution disks onto two working disks. And you may occasionally have to click the **Drive** button in a file selection box to find the files mentioned in the walkthrough.

The MiniEdit source files are contained in a folder called Demo Project on the third of the three LightspeedC distribution disks. You will also need the MacTraps project, since it is included as a library in the MiniEdit program.

Starting LightspeedC

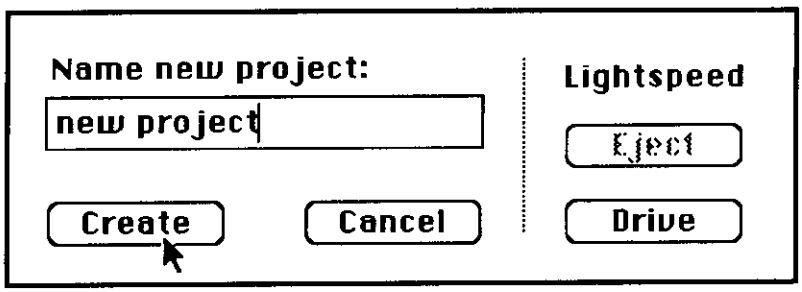
Double-clicking on the LightspeedC icon will start up the LightspeedC development environment.

A file selection box will appear showing all projects on the current disk. You have the option to create a new project or open up an existing one. For example:

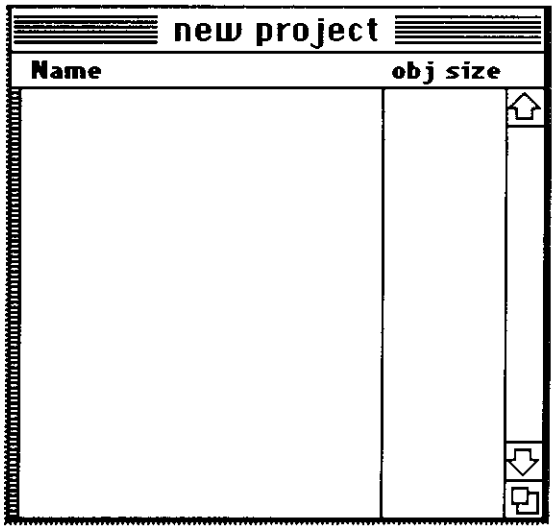


There are several demo projects provided with LightspeedC. (Their names do not match those shown in the file selection box above.) Double-clicking on the name of an existing project document will open up that project for you. However, for the moment, let's create a new project. If you're not in too much of a hurry, we'd like to walk you through the entire development process. It won't take long.

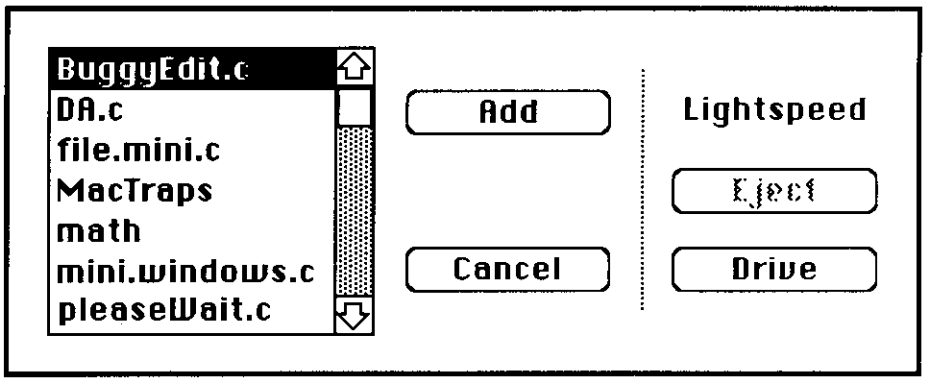
Press the **New** button in the file selection box. You will be asked to name the project. We're calling it *new project.*



A *project window* will open up in the upper right corner of your screen. At the moment, there's nothing in it.



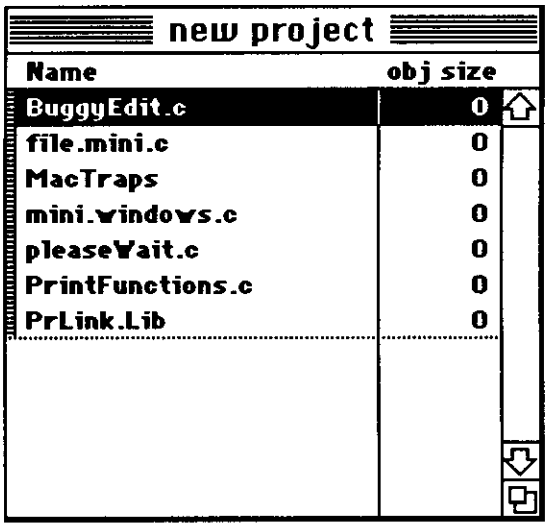
Now pull down the **Source** menu. The **Add...** command will open up a dialog box that lets you see what files you can add to the project.



Three types of files may appear in the file-selection box: text files with a *.c* or *.C* extension in their names, library files (often, but not necessarily, with a *.Lib* extension), and other projects. All other files are filtered out, since only these types of files can be added to a project. You can open any text file for editing, using the **Open...** command on the **File** menu, but only *.c* or *.C* files can be compiled. (`#include` files are implicitly included in the project, but do not appear in the project window.)

Using **Add...**, add the files *BuggyEdit.c*, *file.mini.c*, *MacTraps*, *mini.windows.c*, *pleaseWait.c*, *PrintFunctions.c*, and *PrLink.lib* to the new project. Each time you select a file (by clicking on its name, and then clicking the **Add** button), the **Add...** dialog box will disappear briefly while the file is being added to the project. Once the file has been added, the dialog box will reappear, so you can select another file. When you are all done, click the **Cancel** button to end **Add...**.

Your project window should now look like this:



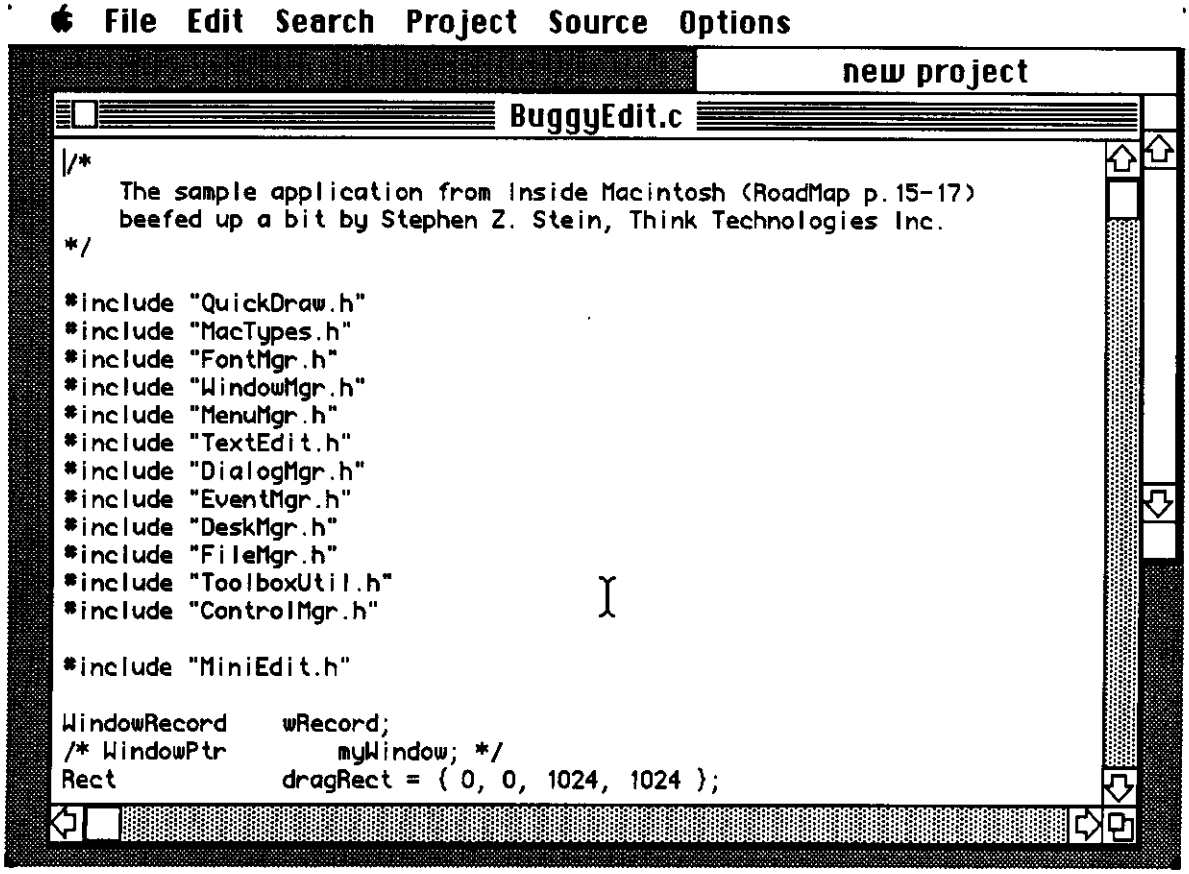
You will notice a column in the project window labeled **obj size**. At present, this column should show a zero for each file you have added, but once a source file is compiled (or a library loaded), it will show the size of the object code in bytes. This number will be zero if the file contains only static variable declarations.

Now, to open any one of the source files listed in the project window, all you need to do is to double-click on its name in the project window.

Editing

LightspeedC has a built-in text editor which edits any Macintosh file of type TEXT. Double-clicking on a file name in the project window will open a text editing window for the file or bring its window to the front if the file is already open.

Double-click on the file *BuggyEdit.c* (that is, on its name showing in the project window). An edit window containing the file will open up:



Take a few moments to become familiar with the LightspeedC editor. The editor's features are described in Chapter 4, but most of its operations should be self-evident if you are familiar with the Macintosh. (If you are not familiar with the Macintosh, you should first read *Macintosh*, the Macintosh owner's manual. You might also want to read its chapter on editing now.)

If you have made any changes to the file, you can save them using one of three commands:

Save — Save the file under its current name. (If it is a new file, you will first be asked to name the file.)

Save As... — Save the file under a new name. The old version remains intact, and the new name becomes the current file in the edit window. If

possible, the new version will also replace the old one in the project window. (See Chapter 4, "More on Editing", for details.)

Save a Copy As... Save a copy of the file under a new name. The old version remains in the edit window, and is available for continued editing.

If you close the editing window (either with the **Close** command on the **File** menu, or by clicking on the close box in the upper left corner of the window) without saving your changes, you will be asked if you want to save them. Clicking on the **Save** button in the dialog box or pressing the return key (since **Save** is the default) will save the changes before closing the file. **Discard** will throw away the changes. **Cancel** will cancel the close command.

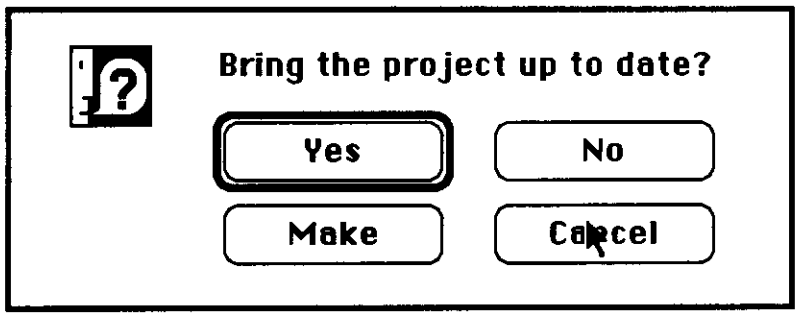
When you are renaming files (or naming a new file for the first time), you should be aware that only files with a *.c* or *.C* extension in their names can be compiled and added to a project. (Files without a *.c* suffix can have their syntax checked by the compiler, but cannot be added to the project.)

Compiling and Running the Project

In most compiler environments, the next step is to compile the program. However, in LightspeedC's integrated environment, you can go right ahead and **Run** the project. Any new source files, any existing source files that have been edited, or any source files whose `#include` files have been edited will automatically be compiled. To see this feature at work, pull down the **Source** menu, and select **Run**, or use the keyboard command **⌘R**. (If you are old fashioned, and feel more secure compiling one file at a time, there is also a **Compile** command (**⌘K**) on the **Source** menu.)

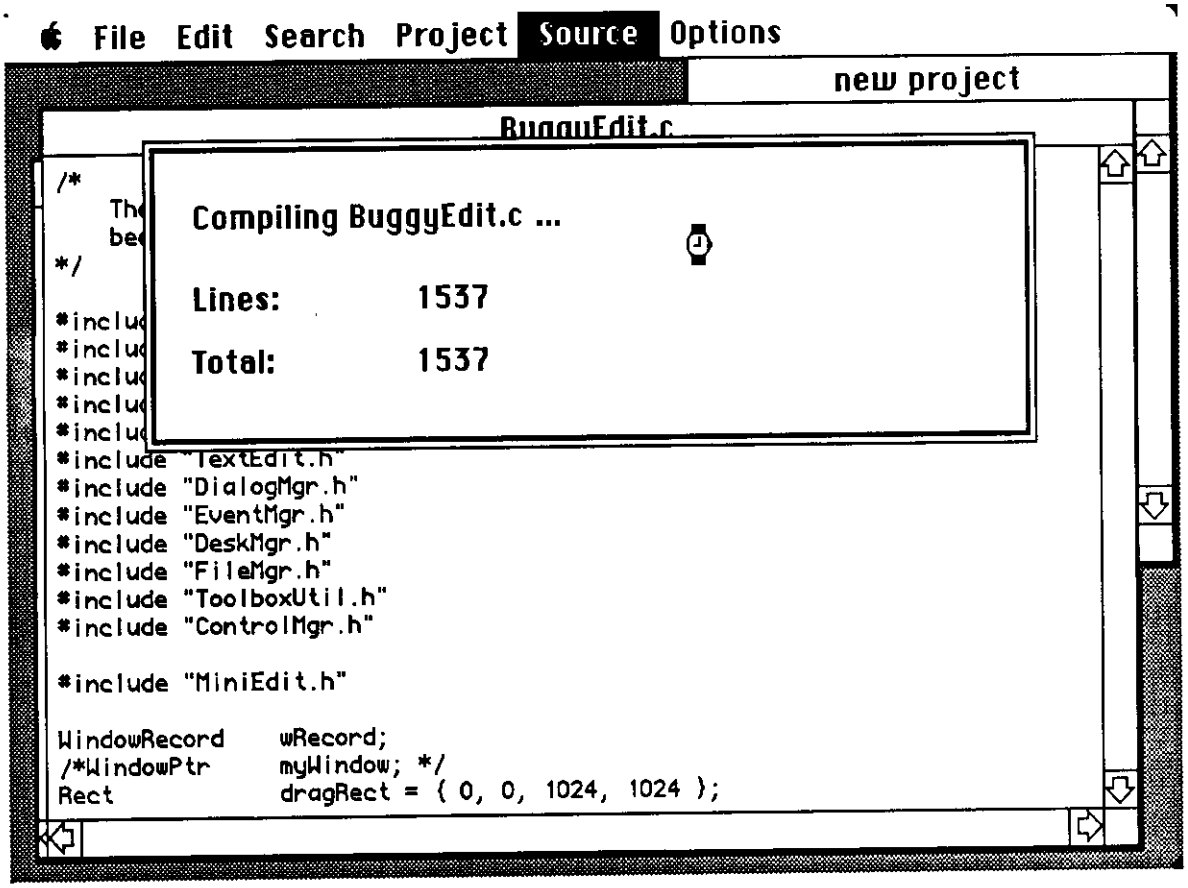
Go ahead and issue a **Run** command, either by picking it from the **Project** menu, or with the keyboard command **⌘R**.

LightspeedC will ask if you want to bring the project up to date:



Click the **Yes** button with the mouse, or simply press the return key, since **Yes** is the default answer. All of the remaining source files that you added to the project will be compiled, and all of the remaining libraries loaded.

While the file is compiling, a window will display how many lines have been compiled. Particularly if you are using a hard disk, you will see just how fast LightspeedC is. The count of lines compiled will pause briefly whenever there is a disk access; it will flash by when LightspeedC is actually compiling. The total number of lines compiled is also shown. For example:



If any errors are found, an error window will announce the error, and the file will automatically open up to the offending source line, so you can fix the problem. If the problem is not in a file that is currently open for editing, the appropriate file will be opened and brought to the front.

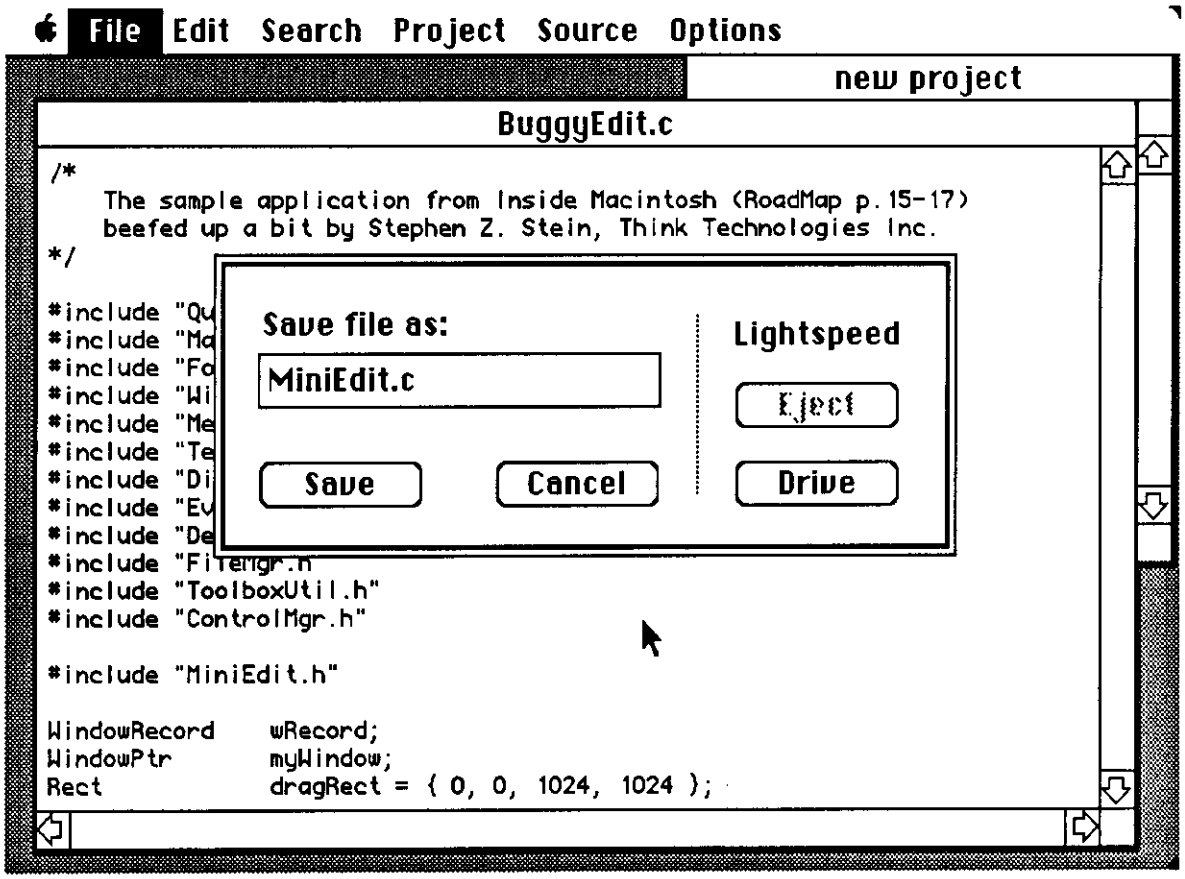
We have deliberately introduced a small "bug" into the sample program by commenting out a definition in the file *BuggyEdit.c*. As you can see if you have followed the instructions so far, the file *BuggyEdit.c* should now be displayed in the edit window, with an error message printed in the "bug" window at the top of the screen:

If you go back to the beginning of the file, you will see that the definition of `myWindow` was commented out.

Now, uncomment the definition and recompile. Note that you don't have to save the file in order to recompile. This allows you to make a change, and test that the modified program actually compiles without error without saving your changes.

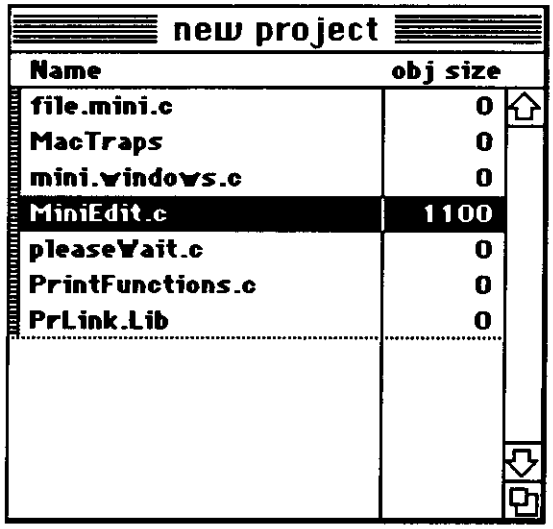
(You can even run a project without saving changes to edited files. However, in this case, LightspeedC will ask if you want to save or discard the changes, so you are forced to make a choice of which version(s) to keep. This feature is useful when debugging, since you can insert temporary debugging statements, and compile and run the program, but not actually save the debugging statements in your sources.)

Rather than recompiling right now, let's look at another feature of the LightspeedC environment. Rather than using the **Save** command on the **File** menu to save *BuggyEdit.c*, use **Save As...** to save it under another name, *MiniEdit.c*, as shown on the next page.



If you click on the visible portion of the project window, and bring it to the front, you will notice that *MiniEdit.c* has replaced *BuggyEdit.c* in the project. (We'll show you this in a moment.) The **Save As...** allows you to replace an older file with a newer version without manually adding or deleting files from the project.

At this point, use the **Compile** command to recompile the current file rather than issuing another **Run** command (which will bring everything up to date and run the project). When the compilation is complete, you should see that the project window now shows the size of the object code that was produced.



While this information may be useful to you, its main purpose is to remind you that the project document is more than just a link list. It is a data structure that contains pointers to the source files, as well as the actual object code, libraries, and various implicit rules for linking the program. The results of a compilation are automatically posted to the project document.

Loading Libraries

Source files, libraries and projects can be added to a project with the **Add...** command, but, as was the case with source files, the actual object code associated with the file is not immediately brought into the project document's data structure. Source files must be compiled, and the precompiled object code contained in libraries or other projects must be loaded into the project.

In both cases, the easiest way to do this is simply to **Run** the project and have LightspeedC's Auto-Make facility figure out what needs to be compiled or loaded. However, just as you can manually compile a single file, you can also manually load a library or another project.

The **Load Library** command on the **Source** menu is used to load a library's object code into the project. This command is only active when the name of a library or a project is selected in the project window. If the name of a project is selected, the command changes to **Load Project.** (There isn't a separate command to load a project, because a project is really a type of library; but the name changes to remind you that there is a difference between the two. Chapter 5 gives an explanation of the difference between ordinary libraries and projects loaded as libraries.) Libraries can also be loaded by the **Make...** command (described in Chapter 5, "Advanced Features").

In order to exercise the **Load Library** command, load MacTraps into the project. This project contains the definition of Macintosh Toolbox and Operating System routines that are not implemented as in-line calls in the LightspeedC compiler.

LightspeedC will display a status message while it is loading the library. When it is done, you will see the object code size of the library listed in the **obj size** column of the project window, just as you do for a source file after it is compiled.

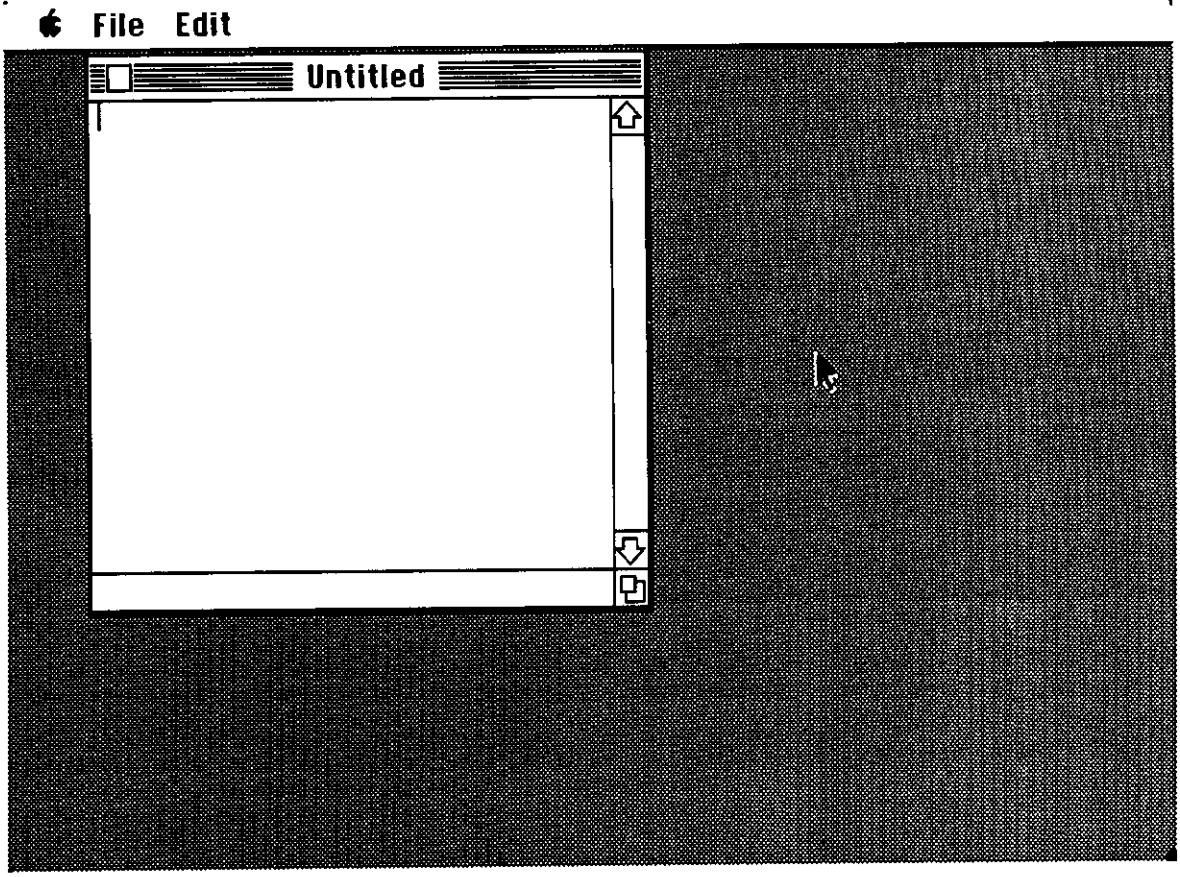
Debugging

A LightspeedC option (see Chapter 5, "Advanced Features") allows you to produce symbols for Apple's Macsbug assembly language debugger when you compile your code. Some tips on using Macsbug with LightspeedC are included in Part Two of this manual.

Running the Project, Again

Now that we've taken all the necessary excursions, let's get back to running the project. Issue a **Run** command either from the **Project** menu or by using the keyboard command **⌘R**. As before, you will be asked if you want to bring the project up to date. Click **Yes**. All of the source files will be compiled, the remaining library loaded, and the project will be linked and run.

The LightspeedC environment will be replaced by an editing window put up by the demo project, the MiniEdit text editor.



Feel free to play with the demonstration program editor, and when you are done, select **Quit** from its **File** menu. You will be returned to the LightspeedC environment.

Building an Application

Once you are satisfied that your project runs correctly, you are ready to save it as an application that can be launched from the Finder independently of LightspeedC. The **Build Application...** command on the **Project** menu saves the project as an application. You will be prompted for the name you want to give the application. (You can also create other types of projects, such as desk accessories. See Chapter 5, "Advanced Features", and Chapter 9, "Running at Lightspeed", for details.)

Issue a **Build Application...** command and save *new project* as an application called *MiniEdit*.

(Note that if an application keeps its resources in a separate resource file, as *MiniEdit* does, this file must be present on disk when you run the saved application. Apple's RMaker and ResEdit programs allow you to move resources into an application to make it completely self-contained. Instructions for using RMaker and ResEdit are provided in the Appendices. MiniEdit's resource file is called *MiniEdit.rsrc*.)

If you change the project type, as described in Chapter 5, the name of this command will change from **Build Desk Accessory...**, **Build Desk Accessory...**, **Build Driver...** or **Build Code Resource...**, depending on which type of project you have selected.

Closing a Project and Quitting LightspeedC

At this point, you can close the demo project by selecting the **Close** command from the **Project** menu. (Don't confuse this with the **Close** command on the **File** menu, which simply closes the file in the current editing window.) If any files need to be saved, you will be asked if you want to save your changes before closing the project. (There is an option to disable this dialog and save your files automatically. See the description of the **Confirm Saves** option in Chapter 6 for details.) LightspeedC will display a file selection box showing the names of all the existing projects on the disk.

Selecting **Quit** from the **File** menu will exit LightspeedC. (If you select **Quit** while a project is still open, it will automatically be closed. If any editing windows have been modified, it will ask you if you want to save or discard your edits, or cancel the **Quit** command. You will not be asked if you want to update the project.)

Where to Go From Here

If you are fairly comfortable with the Macintosh interface, you may want simply to scan through Chapter 6, "Menu Reference", to familiarize yourself with the commands on each of the menus. If you want a little more guidance, read on. Subsequent chapters in Part One will discuss editing in more detail, followed by advanced features of the LightspeedC environment.

Part Two of this manual discusses programming issues in using LightspeedC, and Part Three provides details of the language and library implementation.

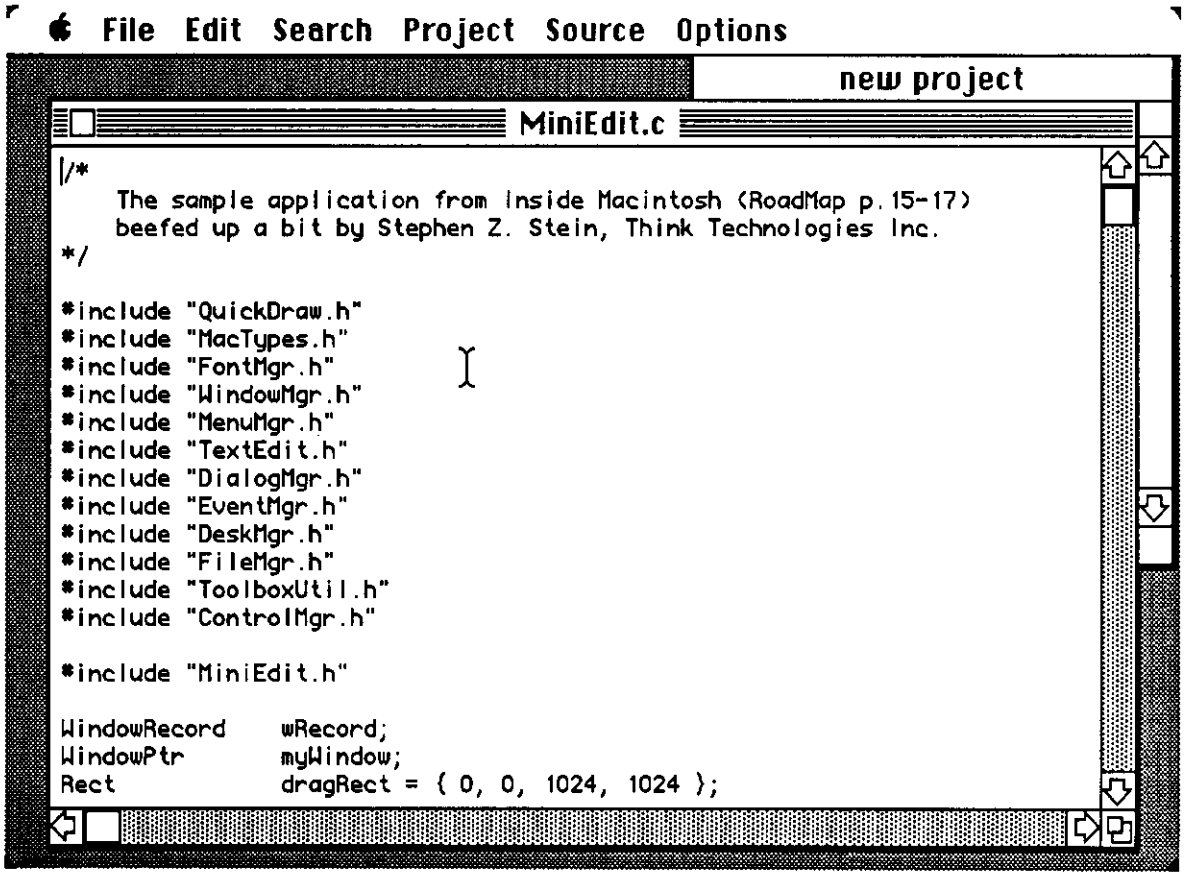
4
More on Editing

Introduction

As described in the previous section, LightspeedC includes its own text editor which can edit any Macintosh file of type TEXT. A file's size is limited only by available memory.

Double-clicking on a file name in the project window will open a text editing window for the file or bring its window to the front if the file is already open. You can also open a file for editing by picking the **Open...** or **New** command from the **File** menu.

The editing window will look something like this:



Moving Around in a File

In the usual Macintosh fashion, you can move around in the edit window by moving the mouse cursor to the point where you want to insert text. Then click the mouse button once to move the text cursor to that point. We'll call that the *insertion point* from now on.

You can control what part of the file is displayed in the window using the scroll bar. If the file is larger than will fit in the current window, you will notice that there is a little white box (called the thumb) in the gray bar on the right side of the screen. The position of the thumb in that bar is proportional to the position of the current screenful of text in the body of the file. To shift the window to view another part of the file, use the mouse to move the thumb up or down in the scroll bar. The window will shift accordingly. If the entire contents of the file will fit in the current window, the thumb will not be visible in the scroll bar.

You can also move up or down a line at a time by clicking on the arrow at the top or bottom of the scroll bar. Clicking in the gray area above or below the thumb moves a screenful at a time. Holding the mouse button down in either location causes repeated scrolling of the text.

You may notice that there is a horizontal scroll bar as well as a vertical one, allowing you to scroll sideways to examine long source lines. (Note that, unlike the vertical scroll bar, the horizontal bar allows you to scroll sideways even when there is no text that extends beyond the current window boundaries.)

Shifting the window with the scroll bar does not change the insertion point. If you want to change the insertion point after you have scrolled the window, move the mouse cursor to the desired position, and click once to position the text cursor.

LightspeedC includes a feature that allows you to examine other areas of the text, then instantly jump back to where you were. Pressing the enter key while you are anywhere in the text will reposition the text to show the current insertion point or the start of a selected text region. Since scrolling in any direction does not affect the insertion point or selected area, this will take you back to your previous position in the file.

Take a moment to familiarize yourself with this feature:

1. Click once with the mouse to position the insertion point anywhere on the current screen.

2. Now use the thumb to move to the end of the file.

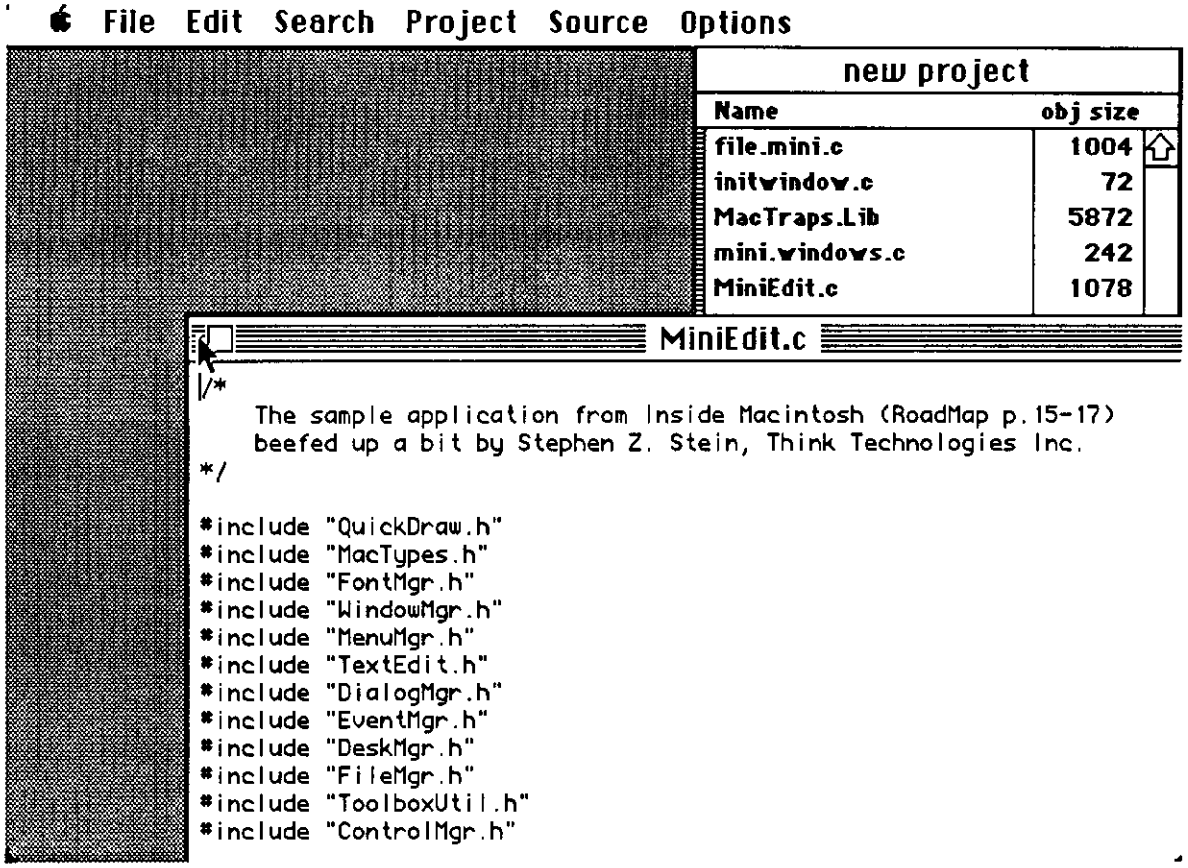
3. Press the enter key to flash back to the original insertion point.

Changing the Size of the Window

The editing window is growable in the standard Macintosh fashion by grabbing the bottom right corner with the mouse and dragging it to shrink or expand the window. You can move it by grabbing it anywhere in the title bar and dragging it to a new location.

The ability to shrink or grow windows and move them around on the screen is most useful when you want to switch back and forth between multiple files and want to make sure that a portion of each window is visible so that you can easily switch back and forth between them.

The figure below shows how to move the edit window by dragging the title bar of the window down and to the right.



Working with Multiple Files

Multiple files can be opened for editing. To open another file, bring the project window to the front by clicking once with the mouse on any visible portion of the window. Then select an additional file for editing by double-clicking on its name in the project window.

You can also use the **Open...** command on the **File** menu to open any text file whether or not it is contained in the project. In this case a standard file box will be presented showing the names of all text files on the current disk volume (or directory using HFS).

In addition, the **Open Selection** command on the **File** menu allows you to open an `#included` header file simply by selecting its name in the current source file (double clicking on the name works here). You need not select the *.h* extension. The selection will automatically be extended to include it as long as the rest of the file name is selected.

The maximum number of source files that can be edited simultaneously depends upon the amount of available memory. It may be necessary to close some files in order to compile. On the other hand, compilation is a trifle faster if the file being compiled or included is open.

A Word About File Specification

All of the examples of file specification shown so far have assumed that the files that were added to the project reside on the same volume as the project itself. However, there may be many cases where you will want to include files from another volume in a project. For example, you may keep LightspeedC and all of the standard libraries on one volume, and your own projects on another.

In this case, LightspeedC will use the following conventions to represent file names, both in the project window and in the titles of edit windows:

filename	Project volume
<filename>	LightspeedC volume
volume:filename	Other volume

LightspeedC also supports Apple's new Hierarchical File System (HFS). That is, all the file-handling mechanisms can deal with files that are not in the project or LightspeedC directories nor in the root directory of any volume. HFS-style path names are recognized by the editor and the `#include` processor and can be tracked by LightspeedC's Auto-Make facility.

The file name display conventions, used in the project window and for titling edit windows, are as follows:

filename	Project directory
<filename>	LightspeedC directory
volume:filename	Root directory
volume:directory:filename	Subdirectory of root
volume:...:directory:filename	Deeper subdirectory
volume:?:filename	Non-root directory on unmounted volume

The directory-search performed for `#include` files also works with the Hierarchical File System. A partial path name (a simple file name or path name beginning with `':'`) enclosed in angle-brackets is interpreted relative to the LightspeedC directory (that is, the directory containing the LightspeedC compiler). A partial path name enclosed in quotes is interpreted relative to the referencing directory (that is, the directory containing the file issuing the `#include` directive), then the project directory, and finally the LightspeedC directory until the file is found.

When an `#include` file name is enclosed by angle-brackets rather than quotes, only the volume containing LightspeedC is searched. This is most often used to indicate a LightspeedC `#include` file such as `<stdio.h>`.

The **Open Selection** command described above is also able to interpret HFS-style path names. If the character following the selection is `.` or `:`, the selection is implicitly extended to the end of the next word following, and this process is repeated. A partial path name is searched for as though it appeared in an `#include "..."` statement in the file being edited. (**Open Selection** is not smart enough to look for angle brackets.) If the editing window is untitled, only the project and LightspeedC directories are searched.

Editing Text

The **Cut** (⌘X), **Copy** (⌘C), and **Paste** (⌘V) commands on the **Edit** menu are supported in the standard Macintosh fashion.

To select text for these commands, move the mouse cursor to the beginning (or end) of the text to be selected. Then, while holding down the mouse button, move the mouse cursor to the opposite end of the text to be selected. The text that is inverted from black-on-white to white-on-black has been selected. You can also select a single word by double-clicking on it, or an entire line by triple-clicking.

The **Cut** command (⌘X) removes selected text and places it in the Clipboard. It replaces the current contents of the Clipboard (if there are any). Material that has been cut can be pasted back into the file, as long as nothing else has been cut or copied to the Clipboard, by moving the insertion point to the desired location and issuing a **Paste** (⌘V) command.

Text can also be deleted (without the option of pasting it back somewhere else) by selecting it, and then choosing **Clear** from the **Edit** menu or by pressing the backspace key.

The **Copy** command (⌘C) copies the selected text and places it in the Clipboard. The copy can now be pasted somewhere else using the **Paste** command (⌘V).

Selecting any region of text and pressing any key which would insert a character will replace the entire selection with the character generated by that keystroke. This is a standard, but dangerous, Macintosh feature. (Note that LightspeedC cannot undo editing changes from the menu, so this is doubly to be watched out for.)

The **Undo** command (⌘Z) on the **Edit** menu has no function in LightspeedC. It is provided for use by any desk accessories that support the **Undo** command.

Program Indentation

Proper indentation is an important part of C programming style. The LightspeedC editor provides several features that help you in maintaining consistent indentation. However, you are responsible for maintaining consistent indentation—that is, LightspeedC helps you but doesn't do it for you.

First of all, whenever you indent lines with leading tabs or spaces, all subseqent lines will automatically be indented by the same distance. Holding the option key down while typing a return inhibits auto-indent for the next line. (If you forget to inhibit the auto-indent, simply press the backspace key at the start of the line to delete one tab at a time.)

Secondly, there are two commands on the **Edit** menu that allow you to shift blocks of text. First, select a block of lines as described above with the **Copy** command.

Then use the **Shift Left** (⌘[) command to shift a selected range of lines to the left. It deletes the first character of each line in the selected range, provided that that character is a tab.

Use the **Shift Right** (⌘]) command to shift a selected range of lines to the right. It inserts a tab at the start of each line in the selected range.

By default, tabs are set to every four columns, but can be reset with the **Set Tabs...** command if desired. If you are using a proportionally spaced font, read "columns" as "em-spaces." (An "em-space" is the width of the letter "m" in a proportional font.)

Caution: Do not accidentally type anything while doing **Shift Left** or **Shift Right** on a selected region. This will, of course, replace the selected text with what you typed.

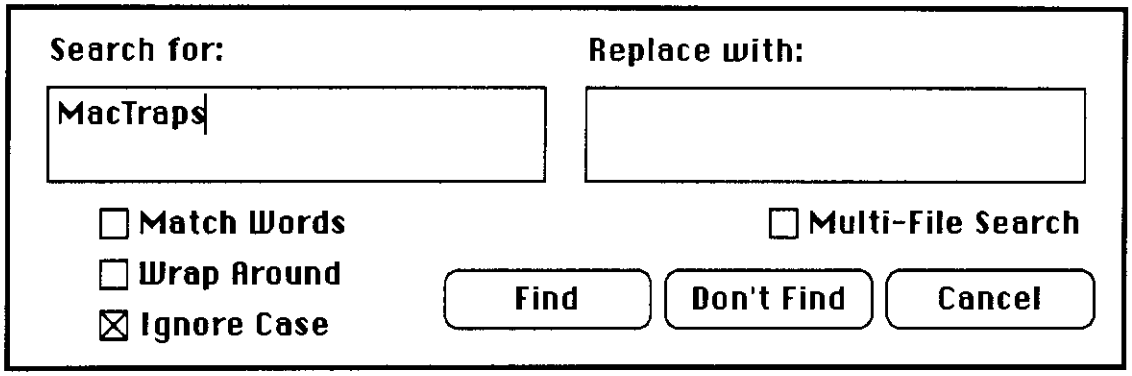
Search and Replace Functions

There are four basic types of search and replace functions that any text editor should support. These are:

- Find a string. This can be used just to move through a file, or to check if a particular string is used in the file.
- Find and replace a single string. This is not the most common operation, since it is often just as easy to move to the desired location and make the change manually.
- Interactively find and replace some, but not all, instances of a given string.
- Globally replace all instances of the search string.

The **Find...** command (**⌘F**) is used both when you simply want to find a string, as well as when you want to perform any of the desired replacement operations.

A dialog box appears in response to this command, and allows you to specify the string to search for, as well as an optional replacement string. For example:



Click the **Find** button in the dialog box when you are ready to start the search. If the string is found, it is highlighted. (This button will become active as soon as you type in the search string.) If it is not found, the editor simply beeps.

Since the string you have found is highlighted, you can replace it simply by typing in the replacement string at this point, even if you didn't enter a replacement string in the dialog box. To find the next instance of the string, use the **Find Again** command (**⌘A**).

If you want to interactively replace some but not all instances of the search string, you should enter a replacement string in the **Find...** command dialog box. Then, when the editor finds the first occurrence, you have the choice of using the **Find Again** command (**⌘A**) to go on to the next instance, **Replace** (**⌘P**), to replace it with the replacement string, or **Replace and Find Again** (**⌘W**) to replace the current instance and immediately go on to the next.

Replace All will replace every instance of the search string in the file (subject to the search options described below). If no replacement string has been specified, it will delete every instance of the search string (that is, it will replace it with nothing).

When you look at the **Find...** dialog box, you will also notice that in addition to the **Find** button ("Go ahead with the search") and the **Cancel** button ("Pretend I never invoked this command"), there is a **Don't Find** button. Pressing this button causes the editor to accept the new search and replace string and option settings but doesn't initiate a search. This is useful if you have entered a search string, then realize that you need to make some other minor edit first. You can do something else, then pick up without having to start the whole process over from the beginning.

A Helpful Hint. Certain non-printing characters (such as carriage return and tab) cannot be typed into the search or replace windows (they have special meanings in Macintosh dialog boxes). To insert these characters, use the **Copy** (**⌘C**) command to clip them from the text, then **Paste** (**⌘V**) them into the window.

A return signifies the end or beginning of a line in a string search.

Search Options

There are three options that control the action of the search and replace commands. These options can be set either from the **Options** menu, or within the dialog box put up by the **Find...** command. These options include:

Match Words
When the **Match Words** option is set, the editor's search and replace functions will work on whole words only. For example, a search for "add" will not match the word "added."

Wrap Around
When the **Wrap Around** option is set, the editor's search and replace functions will search the entire file, rather than from the current position to the end of the file. When the end of the file is reached, the search "wraps around" to the beginning of the file and continues.

Ignore Case
When the **Ignore Case** option is set, the editor's search and replace functions will disregard case when performing a search. A search string will match either upper or lower case.

The **Ignore Case** option is on by default; the other two are off. Settings made in either the **Find...** dialog box or on the **Options** menu will remain in effect during the current LightspeedC session. Use the **Save Settings** command on the **Options** menu to store them in your copy of LightspeedC and keep them in effect during future sessions.

Searching Through Multiple Files

As you may have noticed, there is another item on the **Search** menu, **Find in Next File**. This command allows you to search for a string through more than one file.

In order to use this command, you must check the **Multi-File Search** check box in the **Find...** dialog box. When you check this box, the **Find...** dialog box will be overlaid with another dialog box which displays all of the text files associated with the project. You can scroll through the list and select individual files by clicking on them to place a checkmark by the name, or you can use the buttons in the dialog box to **Check All**, **Check None**, **Check All .c**, or **Check All .h** files. (If a file is already selected, clicking on its name will remove the checkmark.)

When you have checked the files you want to have searched, click **OK** to return to the **Find...** dialog box.

LightspeedC will search for the string specified in the **Find...** dialog box through each of the files that have been checked, starting with the first file checked. If the search string is found in a given file, an edit window containing the file is opened up, and the search string is selected. At this point, you can go on and make any edits you choose. If you want to search further in the current file, you can use the **Find...**, **Find Again**, **Replace**, and **Replace All** commands, which work within the current file. When you are ready to go on with the multi-file search, use the **Find in Next File** command.

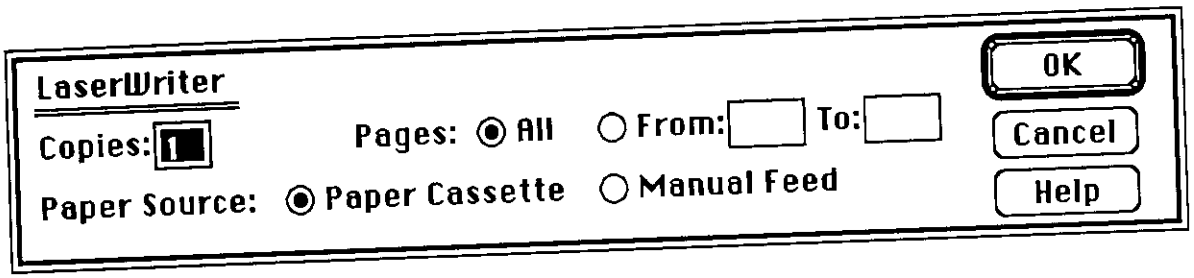
When using multi-file search, you should be aware that it finds each file containing the search string, and opens the file at the first instance of the string, but does not search for additional instances of the string in the file. Use **Find Again** for that. Once you issue a **Find in Next File** command, LightspeedC will begin searching in the next file you have checked, even if there are additional instances of the search string in the current file.

This feature is extremely useful when you are writing a program, and decide to modify a function that is used in multiple files. You can quickly open each of the files containing the search string, so you can switch back and forth between the various edit windows as necessary. (This is also helpful when you are tracking down link errors due to undefined or multiply defined symbols.)

Printing Your Files

What is a programming environment, however sophisticated, without the capability to print hardcopy listings of your source programs?

The **Print...** command on the **File** menu provides this necessary function. When you issue this command, a standard Macintosh print dialog box will prompt you for various printer settings. For example, if you are using a LaserWriter, you will see the following box:



If, as is likely, there are no changes to the default settings, simply press the return key or click **OK** to send the file to the printer. To change any of the defaults, simply click on the appropriate button.

Click **Cancel** to cancel the **Print...** command, or hold down the ⌘ key while pressing the period to stop printing that is in progress.

If there is no printer resource in the system folder, a warning appears when you choose this command. See your Macintosh owner's manual for details.

The **File** menu also contains a **Page Setup...** command that allows you to specify the size of the paper you're printing on, and whether the file should be printed upright on the page (tall orientation) or sideways (wide orientation). You will usually not need to use this command, unless your program has very long lines that need to be printed sideways on the page, or unless your printer uses a nonstandard paper size. See Chapter 6, "Menu Reference", for additional details.

Saving Your Edits

There are four different commands that you can use to save your edits.

The **Close** command closes the file in the active edit window. If the file has been modified since it was last saved, a dialog box will ask you if you want to **Save** the changes, **Discard** them, or **Cancel** the **Close** command. Issuing this command is the same as clicking on the close box in the top left corner of the edit window.

The **Save** command (⌘**S**) saves the file in the active edit window to disk. If you have opened the file with the **New...** command, the file is initially untitled, so a dialog box will ask you to name the file. Otherwise, the file will be written out to disk without further ado, and you can continue editing. You should save files frequently, especially during program development.

Note that an updated file can be compiled and added to the project without being saved, as long as it has been saved at least once and given a name with a *.c* extension. (However, if you try to run the program, a dialog box will ask you if you want to save the updated file.)

The **Save As...** command allows you to save the current file under another name. If you have made edits in the current session, they will be saved under the new name. The original file will remain unchanged, and as you continue editing, you will be editing the new file. This feature is useful for experimenting with a new version of a file, leaving the old file as a backup.

Save As... tries not to sever the tie between the file you are editing and its entry in the project window. If certain conditions are met, the entry for the file in the project window is changed to match the new file name. Here are the conditions:

1. The original file must already appear in the project window.
2. The new name you want to save it as must have a *.c* (or *.C*) extension.
3. The name you want to save it as can't already appear in the project window.

Use **Save a Copy As...** if you don't want the new file to replace the old one in the project.

Unlike **Save As...**, the **Save a Copy As...** command does not affect the status of the file currently being edited; it simply snapshots it to another file. This is a good way to make backups without finding yourself editing the backup!

Neither **Save As...** nor **Save a Copy As...** will let you save into a file already open in an edit window.

There is one other **File** menu command that you may find useful while editing. **Revert** restores the last-saved version of the current file, and discards any edits made in the current session. LightspeedC asks for confirmation before reverting.

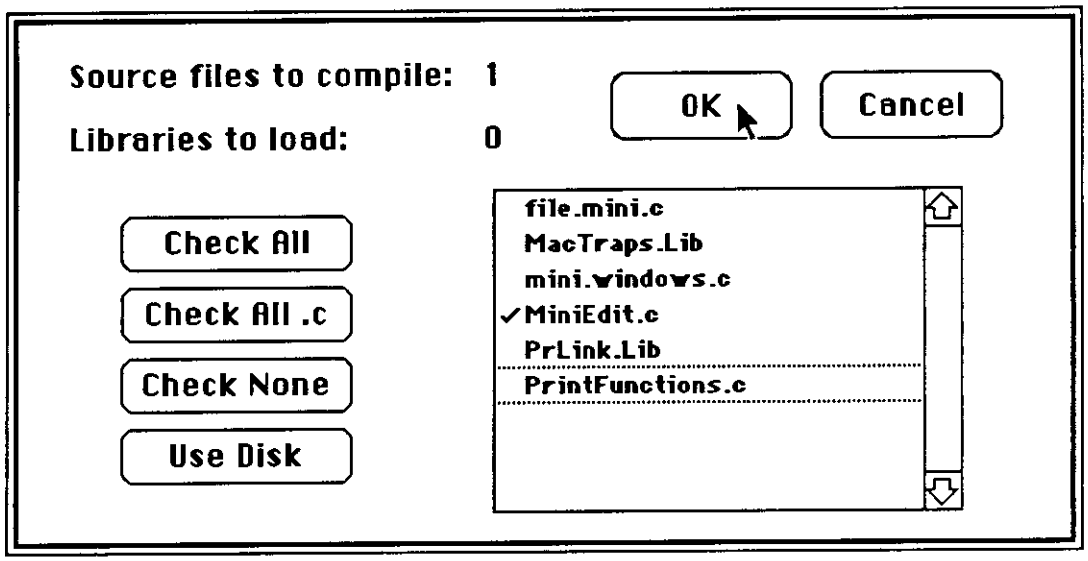
5
Advanced Features

Make: Compiling Files *en masse*

As described earlier in this manual, LightspeedC keeps track of the status of files in the project, and whenever you **Run** the project, makes sure that all changed files are recompiled.

While in most cases, this Auto-Make facility will be sufficient to ensure that the project is kept up to date, there are times when you may need to take more control. For example, sometimes a source file can belong to more than one project. When such a file is updated in one project, the other has no way of knowing that the file has been changed.

The **Make...** command on the **Source** menu (⌘M) allows you to manually specify which files you believe need to be brought up to date. When you issue this command, the *make window* is displayed. This window includes a scroll box containing the names of all of the source files and libraries that are contained in the project. (File names are alphabetical by segment. Refer to the section "Segmentation" in this chapter for more information.) For example:



LightspeedC puts a checkmark next to the name of each source file that it thinks needs to be recompiled. A file needs to be recompiled because either it or its `#include` files have been edited since it was last compiled.

The files to be recompiled or reloaded can be changed simply by clicking on the file names. A checkmark will appear (or disappear if you click on a file that is already checked) to the left of the filename. If you change your mind, and want to go back to the original settings, click the **Cancel** button.

The **Check All** or **Check None** buttons allow starting from either extreme, and can simplify marking a long list of files. **Check All .c** allows you to check all files with a *.c* or *.C* extension; LightspeedC will recompile all of your source files, but will not reload any libraries.

The **Use Disk** button causes LightspeedC to ignore all current checkmarks and compare the last known source and `#include` file compile date/time to the actual source file modification date/time stored with each file on the disk by the Macintosh file system. It will then place a check mark next to any file which has different date/time value than LightspeedC 's compiled version.

Use Disk is normally not necessary because LightspeedC's Auto-Make facility watches as you edit and knows which files have been changed. However, this knowledge is project-specific, so if you have a source file that belongs to two different projects and you change it in one, the other project won't know that it has been changed unless you say **Use Disk**. (You should also use **Use Disk** if you have edited a file outside of Lightspeed.) Also, if a library file or included project changes you will have to click **Use Disk** to let LightspeedC find out about it. Clicking **Cancel** does not undo the effect of **Use Disk**, since this command actually changes the date/time record associated with a file in the project.

Since **Use Disk** is somewhat slow, if you know that a library has changed, it is probably easier to reload it manually. You can do this either by selecting the library in the project window and then using the **Load Library** command, or simply by putting a checkmark next to its name in the make window.

Clicking **OK** causes any changes to the check-list you have made to become permanent and the specified source compilations and/or library loadings proceed in batch. You can abort the sequence with command-period (⌘ .). If there are any compile errors, the sequence will stop automatically, just as if the compilation had been started with the **Compile** or **Run** command.

In any case, Auto-Make keeps track of which actions have been completed, so you can issue another **Make...** command to pick up where you left off.

The Confirm Auto-Make Option

At the other end of the spectrum from the additional control provided by the **Make...** command is the **Confirm Auto-Make** command on the **Option** menu.. By default, the Auto-Make facility queries you when it finds that the project is not up to date, and you have tried to **Run** the project or issue any **Build...** command. You have the option at that point to bring the project up to date, go ahead without recompiling, bring up the make window, or cancel.

Setting the **Confirm Auto-Make** option off (it is a toggle, and on by default, so selecting it from the **Option** menu will turn it off), the Auto-Make facility will go ahead and automatically bring the project up to date whenever you issue a **Run** or any **Build...** command.

Segmentation

Segmentation is the division of compiled code modules (which are contained within the project document) into units which are considered a single entity. Some of the reasons for using segmentation are:

1. The program is too large to fit in memory all at once. It can be broken up into segments that are loaded and unloaded from memory as necessary. (See the "Segment Loader" chapter of *Inside Macintosh* for details.)

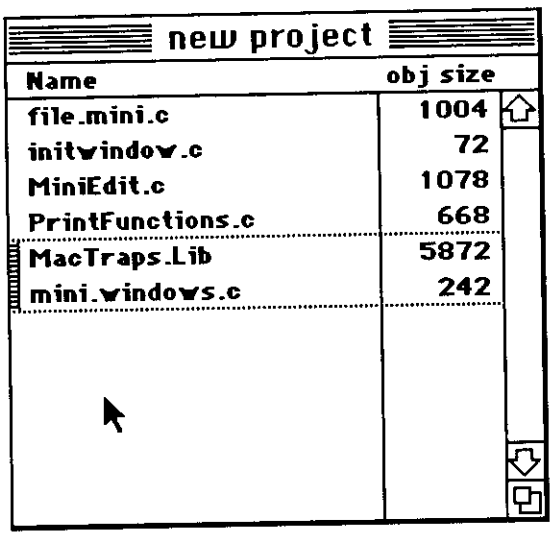
2. To cope with Macintosh Segment Loader and Resource Manager restrictions that allow only 32K or smaller segments.

3. To exploit the Motorola 68000 processor's relative jump instructions, which are limited to a range of 32K.

4. For ease in separating different logical sets of files within a single project. The files in each segment appear in the project window clustered together in alphabetical order within the segment.

Dragging a file name in the project window to a position just below the dotted line at the bottom of the project window creates a new segment containing only that file. Dragging a file onto another file in the project window moves it to the segment containing that file.

Each segment is separated by a dotted line. The currently active segment is indicated by a thin gray bar down the left side of the project window. To change which segment is active, simply click on the name of a file in that segment. In addition, you can click on the left margin of the project window, where the gray segment bar would be, to select a segment. In the figure below, the second segment is active.



Any segments that become empty are automatically deleted. New files are placed in the active (currently selected) segment.

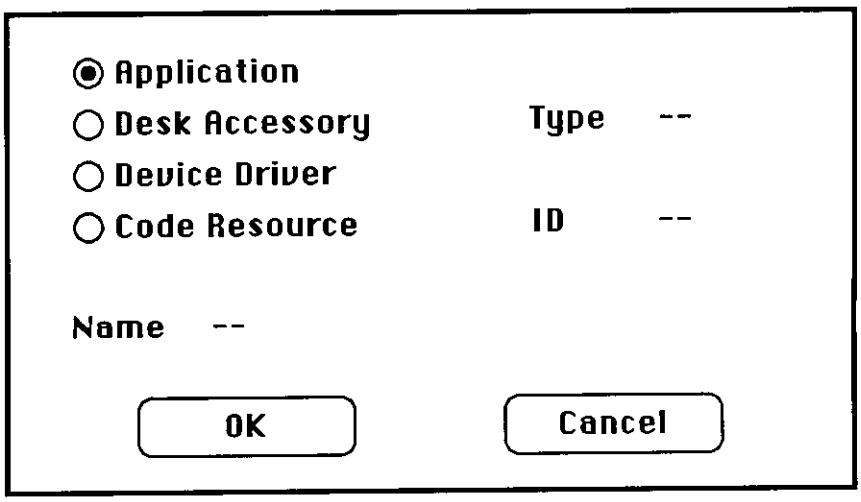
Moving files between segments takes a little more time if there is a lot of object code associated with each file. To easily segment a large project, use the **Add...** command from the **Source** menu to add all source files that you want in the first segment. Then select a new (empty) segment as described above, and add the files that you want in that segment.

Setting the Project Type

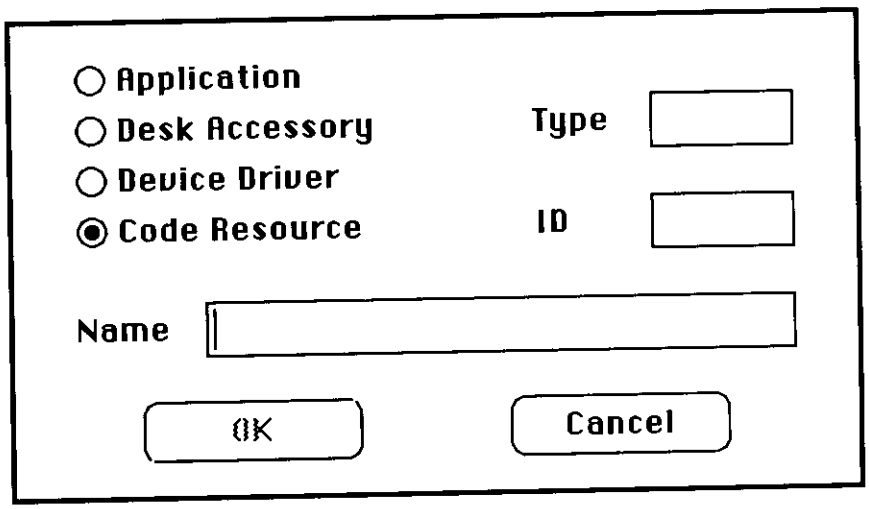
By default, LightspeedC is set up to build an application program. However, there are several other possible types of projects that you can create, including desk accessories, device drivers, and code resources. The compiler needs to treat such projects differently, since they use different register allocation and data storage conventions.

The **Set Project Type...** command on the **Project** menu allows you to set the project type. If the project is a desk accessory, device driver or code resource, it also allows you to specify other information as appropriate, such as a resource type, ID, and name.

This command puts up the following dialog box:



As long as the project type is set to **Application** (the default value), you can't fill in any of the other information. However, for the other types, there may be additional information, such as the resource name, type and ID, that you will need to fill in. For example, if you set the project type to **Code Resource**, the dialog box will look like this:



You should set the project type before compiling any of your sources, since LightspeedC will need to throw away any existing object code if you switch the project type from **Application** to **Desk Accessory**, **Device Driver** or **Code Resource**. If any objects are contained in the project, a dialog box will ask if you are sure you want to change the project type before going ahead with the conversion.

When you set the project type to something other than **Application**, the name of the **Build Application...** command on the **Project** menu will change accordingly. For example, if you set the project type to **Desk Accessory**, the menu will read **Build Desk Accessory...**.

Creating Libraries

A library is simply a binary representation of a project. There are three types of libraries that can be used with LightspeedC:

1. A project which has been stored as a library with the **Build Library...** command.
2. An MDS *.Rel* file that has been converted to a library using the RelConv utility. (See "Interfacing with Assembly Language" in Chapter 9, "Running at Lightspeed", for details.)
3. A project that has not been saved as a library, but is still in project form.

Other than their origins, there is no difference between the first two types of libraries. Each is a single binary file that will be linked as a whole into the project. A library file may have any name, but by convention, the first two types of libraries have a *.Lib* extension added to their name to distinguish them from the third type—an active project.

While a project that has not been saved using **Build Library...** may contain the same object code as one that has been saved with that command, it also contains data structures that allow LightspeedC to link it much more intelligently. Only the files in the library that are actually used

will be linked into the executable program when you **Build Application...** (or any other project type). Including one project in another is preferable to saving it as a single *.Lib* file with **Build Library...**, since it allows LightspeedC to link only those files that are actually used. It is for this reason that the various libraries included with LightspeedC are shipped as projects rather than as *.Lib* files.

Note that since a library file lives outside the project and is usually changed from outside the project, the project is never directly aware of when a library file on disk changes. You can reload a library with the **Load Library** command.

If you are not sure if a library has been changed, the **Use Disk** function under **Make...** will check the internal project copy date/time of a library against the disk date/time and will mark the library for reloading if a newer version must be copied into the project.

6
Menu Reference

Introduction

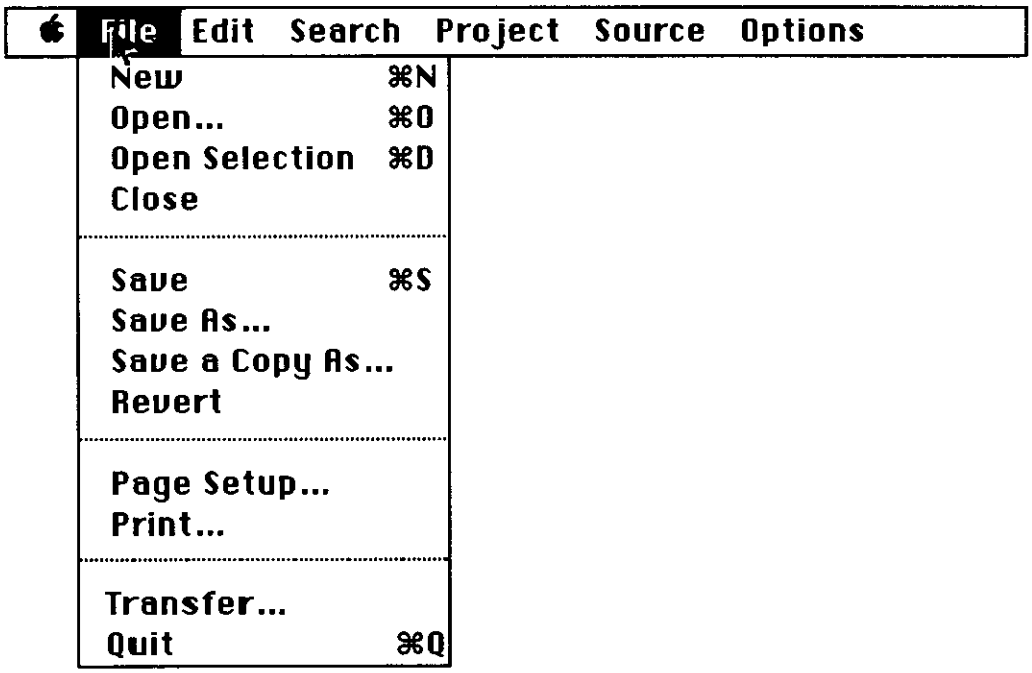
This chapter gives a brief summary of the function of each of the LightspeedC menu commands. It is organized by menu, from left to right along the menu bar. Within each menu, commands are described in the order in which they appear on the menu. This allows you to quickly look up the function of each of the commands.

In some cases, this chapter presents additional detail, but in many cases, it is simply a recap of information presented earlier. It is designed to give a quick overview for those who did not read the earlier chapters, or as a refresher and reference for those who did. If you have just read Chapters 1 through 5, you need not read this chapter now.

The Menu

About LightspeedC...
This command gives version information about LightspeedC in an entertaining display. Click the mouse button to end the display.

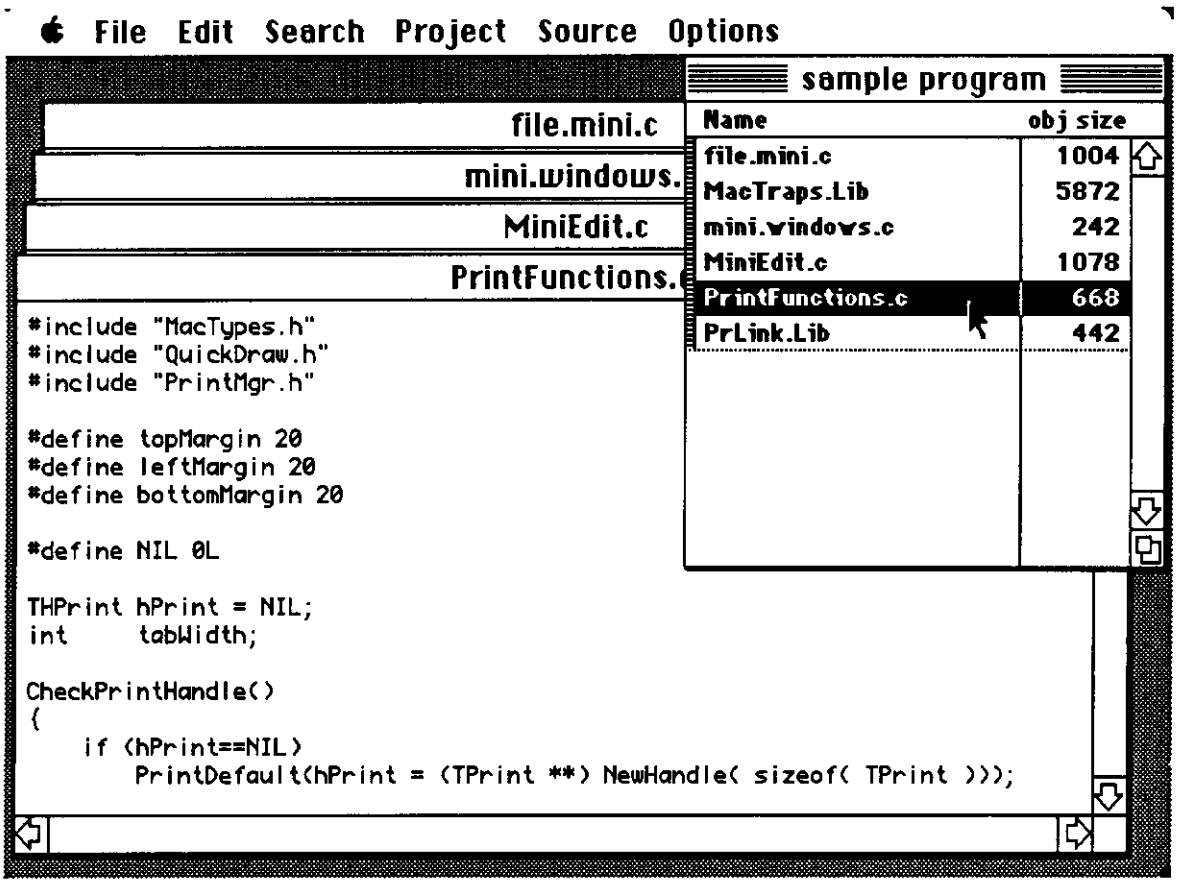
The File Menu



New (⌘N)
This command opens a new file, with a window name of *Untitled.* You must save this file with a *.c* extension if you plan to compile it or add it to the project. However, you can use the **Check Syntax** command on the **Source** menu to compile it without adding it to the project.

Open... (⌘O)
This command displays a dialog box that allows you to select from existing files on the disk and open them for editing. When you first begin a project, one way to add a file is to open it with this command, then compile it or add it with the **Add** command. (The **Add...** command allows you to add multiple files without compiling or opening the files. Once files have been added to the project, they can be opened by double-clicking on the file name in the project window.)

You can open multiple files and the edit windows will stack up with the titles showing, so you can easily click on any window to bring it to the front.



Due to memory requirements, if you open too many files at once, you may have to close some in order to compile.

Open Selection (⌘D)

This command allows you to open an `#included` header file simply by selecting its name in the current program text. You need not select the *.h* extension (or full path name for HFS files). The selection will automatically be extended to include it as long as the file name itself is selected.

Close

This command allows you to close any window with a *go away* box in its upper left corner, including the active edit window, the link error window, or a desk accessory. If you try to close an edit window, and the file has been modified since it was last saved, a dialog box will ask you if you want to save the changes, discard them, or cancel the **Close** command. Issuing this command is the same as clicking on the go away box. Note that this command cannot be used to close the project window. Use the **Close** command on the **Project** menu for that purpose.

If the **Confirm Saves** option on the **Options** menu has been unchecked, you will not be prompted about saving changes. Any changed file will automatically be saved when you close it.

Save (⌘S)
This command saves the file in the active edit window to disk. If the file is currently untitled, a dialog box will ask you to name the file.

Note that an updated file can be compiled and added to the project without being saved, as long as it has been saved at least once and given a name with a *.c* extension.

Save As...
This command allows you to save the current file under another name. If you have made edits in the current session, they will be saved under the new name. The original file will remain unchanged, and as you continue editing, you will be editing the new file. This feature is useful for switching to a new version of a file, leaving the old file as a backup.

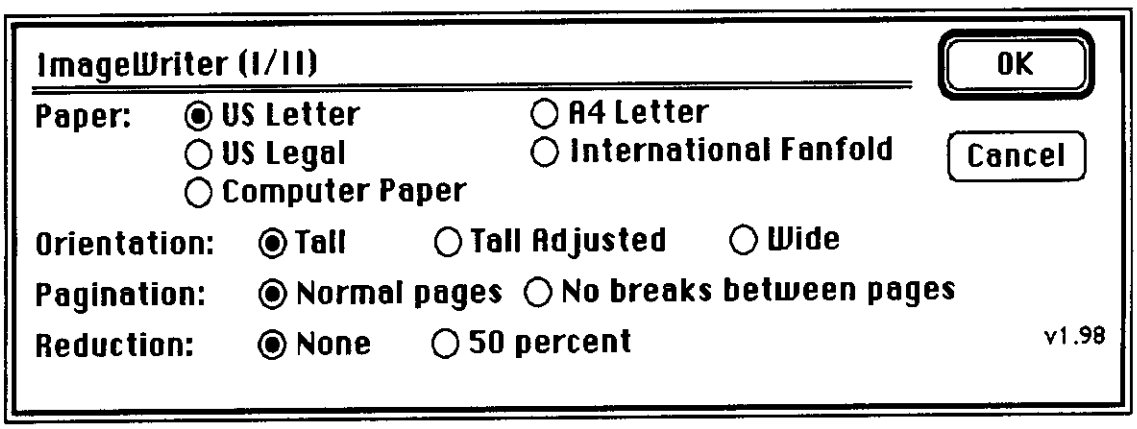
Save As... tries to preserve the tie between the file you are editing and its entry in the project window. If the file appears in the project window, and the name you want to save it as has a *.c* (or *.C*) extension, and if the new name doesn't already appear in the project window, then the entry for the file in the project window is changed to match the new file name. Use **Save a Copy As...** if you don't want this to happen.

Save a Copy As...
Unlike **Save As...**, this command does not affect the status of the file currently being edited; it simply snapshots it to another file. This is a good way to make backups without finding yourself editing the backup!

Neither **Save As...** nor **Save a Copy As...** will let you save into a file already open in an edit window.

Revert
This command restores the last-saved version of the current file, and discards any edits made in the current session.

Page Setup...
This command allows you to specify the size of the paper you're printing on, and whether the file should be printed upright on the page (tall orientation) or sideways (wide orientation).

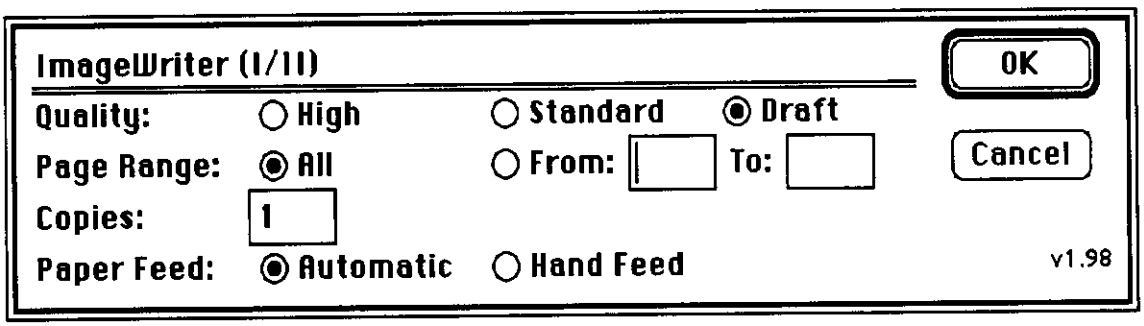


Paper sizes are as follows:

US Letter	8 1/2 by 11 inches
US Legal	8 1/2 by 14 inches
A4 Letter	8 1/4 by 11 2/3 inches (European standard)
International Fanfold	8 1/4 by 12 inches (International standard)

If there is no printer resource in the system folder, a warning appears when you choose this command. See *Macintosh,* your owner's manual, for details.

Print...
This command allows you to print the current file. A dialog box will prompt you for various settings. For example, if you are using an ImageWriter:



If, as is likely, there are no changes to the default settings, simply press the return key or click **OK** to send the file to the printer. To change any of the defaults, simply click on the appropriate button. **Quality** refers to the print density; there is a tradeoff between print quality and printing speed. **Standard** quality tries to strike a happy medium.

Paper feed should be **Continuous** unless you need to manually feed paper into your printer.

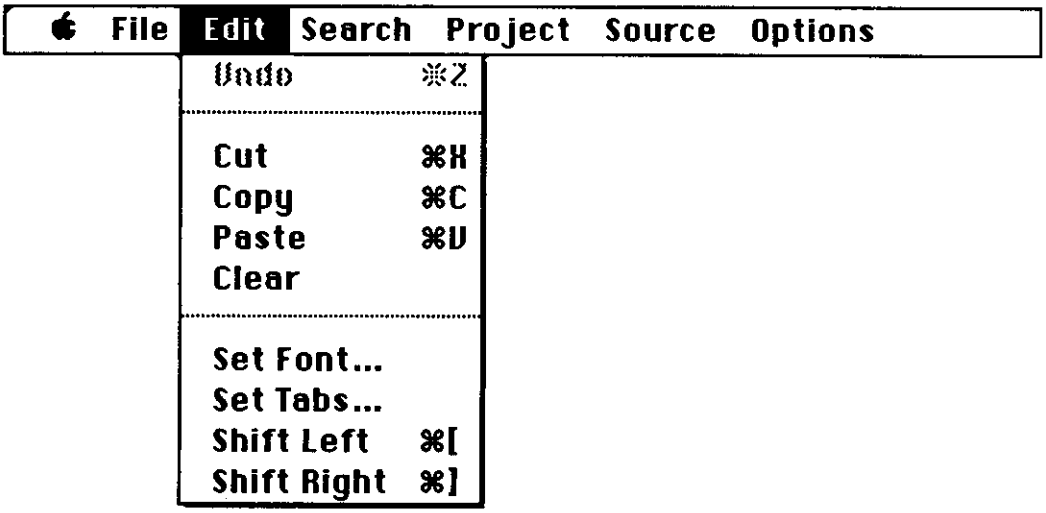
Click **Cancel** to cancel the **Print...** command, or hold down the command key ⌘ while pressing the period (.) to stop printing that is in progress.

If there is no printer resource in the system folder, a warning appears when you choose this command. See *Macintosh*, your owner's manual, for details.

Transfer...
This command allows you to launch another application without first returning to the desktop or MiniFinder. The dialog box associated with this command shows you a scroll box containing the names of all other applications on the volume.

Quit (⌘Q)
This command exits LightspeedC and returns to the desktop or MiniFinder.

The Edit Menu



The **Edit** menu contains the standard Macintosh editing functions **Undo, Cut, Copy, Paste,** and **Clear**, as well as the additional formatting commands, **Set Font..., Set Tabs..., Shift Left** and **Shift Right,** which are especially useful to programmers for producing nicely formatted code.

To select text for these commands, move the mouse cursor to one end of the text to be selected. Then, while holding down the mouse button, move the mouse cursor to the opposite end of the text to be selected. The text that is highlighted with a dark band has been selected. You can also select a single word by double-clicking on it, or a line by triple-clicking. Selected text can now be operated on by any of the commands on the **Edit** menu. (In addition, typing anything while text is selected will replace the selected text with what you have just typed.)

Undo (⌘Z)
This command does nothing in LightspeedC. It is only provided for use by any desk accessories that you may have available. If it is supported by a given desk accessory, this command will undo the effect of a change made in the desk accessory.

Cut (⌘X)
This command removes selected text and places it in the Clipboard. It replaces the current contents of the Clipboard (if there are any). Material that has been cut can be pasted back into the file, as long as nothing else has been cut or copied to the Clipboard, by moving the cursor to the desired location and issuing a **Paste (⌘V)** command. Text can also be deleted (without the option of pasting it back somewhere else) by selecting it, and then choosing **Clear** from the **Edit** menu or pressing the backspace key.

Copy (⌘C)
This command copies the selected text and places it in the Clipboard. The copy can now be pasted somewhere else using the **Paste** command **(⌘V)**.

Paste (⌘V)
This command copies the contents of the Clipboard into the file being edited at the insertion point. If text is currently selected, it is replaced.

Clear
This command clears the selected text. The selection is not placed on the Clipboard, and cannot be recovered.

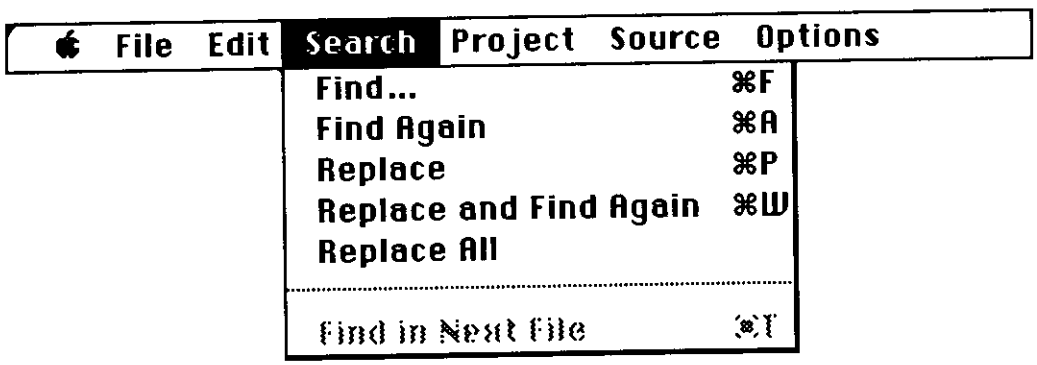
Set Font...
This command allows you to set the font in which text will be displayed by the LightspeedC editor. Selecting this command will display a dialog box which lists all available fonts. This dialog box is similar in format to that used by Apple's Font/DA Mover. Font names are shown in a scroll box, and as each is selected, a sample of the type contained in the font is shown at the bottom of the dialog box. Fonts can be set separately for each edit window.

Set Tabs...
This command sets tab stops. By default, tabs are set to 4 spaces (4 em-spaces if you are using a proportional font). To simplify consistent indentation, once you have typed a tab, successive lines will be indented by the same number of tab stops as the previous line. Press backspace to return a subsequent line to the previous tab stop or hold the option key while pressing return to move to the left margin on the next line. Tabs can be set separately for each edit window.

Shift Left (⌘[)
This command allows you to shift a selected range of lines to the left. It deletes the first character of each line in the selected range, provided that that character is a tab.

Shift Right (⌘])
This command allows you to shift a selected range of lines to the right. It inserts a tab at the start of each line in the selected range.

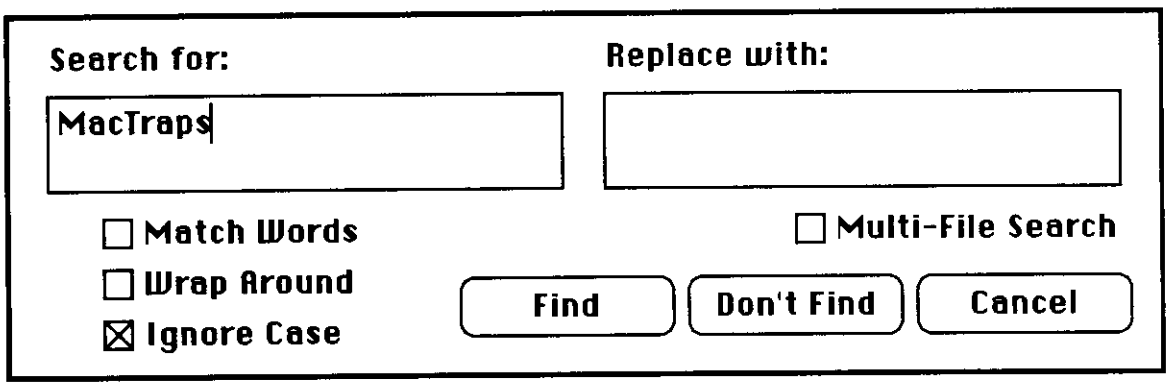
The Search Menu



The commands on the **Search** menu allow you to search for and replace arbitrary text strings in your file.

Find... (⌘F)
This command allows you to specify a string to search for. If the string is found, it is highlighted. If it is not found, the editor simply beeps.

At the start of an editing session, only the **Find...** command is active. The dialog box that appears in response to this command allows you to specify a string to search for, as well as an optional replacement string.



You can also set search options in this dialog box. These options include:

Match Words
When the **Match Words** option is set, the editor's search and replace functions will work on whole words only. For example, a search for "string" will not match the word "strings". This option is not set by default.

Wrap Around
When the **Wrap Around** option is set, the editor's search and replace functions will search the entire file, rather than from the current position to the end of the file. When the end of the file is reached, the search "wraps around" to the beginning of the file and continues. This option is not set by default.

Ignore Case
When the **Ignore Case** option is set, the editor's search and replace functions will disregard case when performing a search. A search string will match either upper or lower case. This option is set by default.

In addition to the **Find** button ("Go ahead with the search") and the **Cancel** button ("Pretend I never invoked this command"), there is a **Don't Find** button in the dialog box. Pressing this button causes the editor to accept the new string and option settings but doesn't initiate a search. This is useful for setting values for a replace operation without executing the first **Find**.

Find Again (⌘A)
This command searches for the next occurrence of a previously specified string.

Replace (⌘P)
This command replaces the first instance of the search string with a replacement string. If no replacement string has been entered in the dialog box resulting from the **Find...** command, this command will clear the string that has been found (that is, it will replace it with nothing).

Replace and Find Again (⌘W)
This command replaces the first instance of the search string with the replacement string, then finds the next instance of the search string, but does not replace it. Use this command to step through a series of replacements. After each replacement, you are shown the next instance of the search string, and can decide at that point whether or not you want to replace it. If you want to replace it, use ⌘P or ⌘W. If not, use ⌘A to find the next occurence.

If no replacement string has been entered in the dialog box resulting from the **Find...** command, this command will clear the string that has been found (that is, it will replace it with nothing).

Replace All
This command replaces every instance of the search string. If the **Wrap Around** option is set, it replaces every instance in the file; if the **Wrap Around** option is not set, it replaces every instance from the current cursor position to the end of the file. Use this command when you don't want to give your approval for every replacement. If no replacement string has been entered in the dialog box resulting from the **Find...** command, this command will clear all instances of the string that has been found (that is, it will replace it with nothing).

Find in Next File (⌘T)
This command allows you to search for a string through more than one file.

In order to use this command, you must check the **Multi-File Search** check box in the **Find...** dialog box. When you check this box, the **Find...** dialog box will be overlaid with another dialog box which displays all of the text files known to LightspeedC. You can scroll through the list and select individual files by clicking on them to place a checkmark by the name, or you can use the buttons in the dialog box to **Check All**, **Check None**, **Check All .c** or **Check All .h**. (If a file is already selected, clicking on its name will remove the checkmark.)

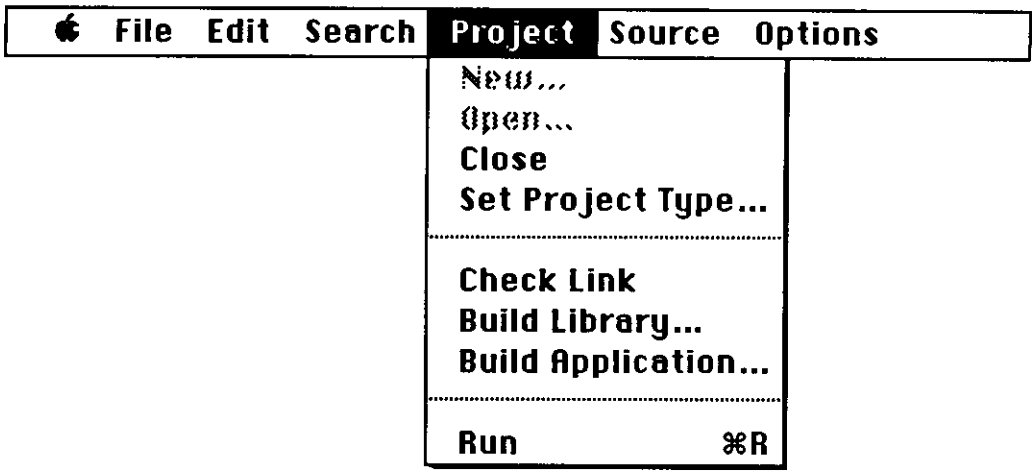
When you have checked the files you want to have included, click **OK** to return to the **Find...** dialog box.

LightspeedC will search for the string specified in the **Find...** dialog box through each of the files that have been checked, starting with the first file checked. If the search string is found in a given file, an edit window containing the file is opened up, and the search string is selected. At this point, you can go on and make any edits you choose. If you want to search further in the current file, you can use the **Find...**, **Find Again**, **Replace**, and **Replace All** commands, which work within the current file. When you are ready to go on with the multi-file search, use the **Find in Next File** command.

When using multi-file search, you should be aware that it finds each file containing the search string, and opens the file at the first instance of the string, but does not search for additional instances of the string in the file. Use **Find Again** for that. Once you issue a **Find in Next File** command, LightspeedC will begin searching in the next file you have checked, even if there are additional instances of the search string in the current file.

This feature is extremely useful when you are writing a program, and decide to modify a function that is used in multiple files. You can quickly open each of the files containing the search string, so you can switch back and forth between the various edit windows as necessary. (This is also helpful when you are tracking down link errors due to undefined or multiply defined symbols.)

The Project Menu



New...
This command creates a new project and opens an empty project window. Only one project can be open at a time.

Open...
This command opens an existing project. A file selection box allows you to select from the list of existing projects.

Close
This command closes the currently open project and brings up a dialog box while allows you to open an existing project or create a new project. If you try to close a project with open files whose latest versions have not been saved, a dialog box will ask you if you want to save your edits. This dialog can be disabled by turning off the **Confirm Saves** option on the **Options** menu. If you do this, changed files will automatically be saved when you close.

Set Project Type...
This command allows you to set the project type. (The type is set to **Application** by default, and can be changed to **Desk Accessory, Device Driver**, or **Code Resource**.) If the project is a desk accessory, device driver or code resource, it also allows you to specify other information as appropriate, such as a resource type, ID, and name.

You should set the project type before compiling any of your sources, since LightspeedC will need to throw away any existing object code if you switch the project type from an **Application** to a **Desk Accessory, Device Driver**, or **Code Resource**. If any objects are contained in the project, a dialog box will ask if you are sure you want to change the project type before going ahead with the conversion.

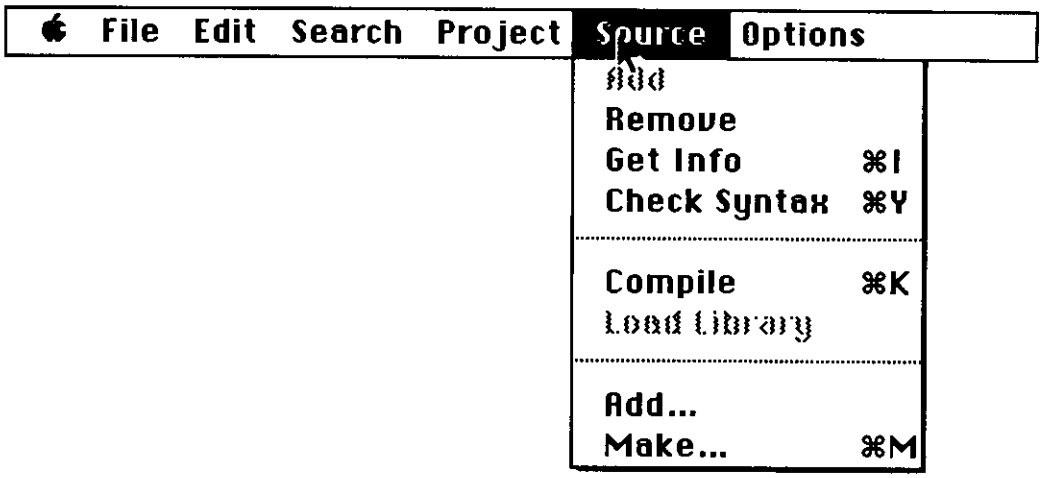
Check Link
This command checks for all the same error conditions as **Run** or **Build Application...** would, but without attempting to run the project or build an application. If any files need to be made, you will be asked whether you want to bring the project up to date, even if you have set **Confirm Auto-Make** off.

Build Library...
This command saves the current project as a single binary file that can be included as a library in other project documents. A dialog box prompts you for the name of the library file. The convention is *name.Lib*; however, a library may have any valid file name. Note that you can include a project in another project without first saving it explicitly as a library.

Build Application...
This command saves the current project as an application—a program that can be run independently of LightspeedC. If you have created the application using a separate resource file, in the recommended LightspeedC fashion, you may subsequently want to integrate the resources with your application by using a program such as ResEdit or RMaker to produce a final version of your application. See Chapter 8, "Using Resources" for more details. If you set a project type other than **Application**, the name of this command changes accordingly. (That is, it may appear as **Build Desk Accessory...**, **Build Device Driver...** or **Build Code Resource...** depending on the project type.)

Run (⌘R)
This command will run the program contained in the project. If the project is not up to date, a dialog box will ask if you want to bring it up to date. If the **Confirm Auto-Make** option on the **Options** menu is not checked, then the project will automatically be brought up to date. This allows you to edit a source file and run the changed program, without ever explicitly recompiling or relinking the program.

The Source Menu

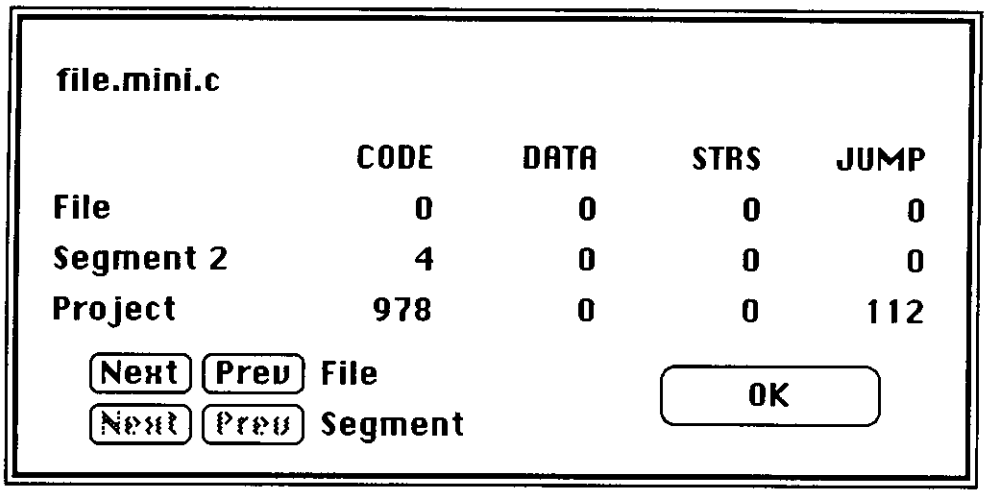


The **Source** menu allows you to add and remove source files and libraries from a project, as well as to compile sources and load library files to the project. Menu selections include:

Add
This command adds an open source file to the project without compiling it. The command is only active when a file with a *.c* extension is open in the current edit window.

Remove
This command removes a selected source file or library from the project.

Get Info (⌘I)
This command brings up an information box containing statistics about the currently selected file in the project window or about the file in the front edit window.



The box shows the size of the code, data, strings, and jump table for the selected file. If the file has not yet been compiled, these values will be zero. The box also shows the same statistics for the segment and the entire project. The **Next** and **Prev** buttons allow you to examine the other files and/or segments in the project while the **Get Info** window is displayed.

Check Syntax (⌘Y)

This command allows you to compile a file in order to check its syntax. This command compiles the front window but does not attempt to add the file to the project window or to post the results of the compilation to the project document. It may be issued on any edit window, even an untitled window. **Compile**, by contrast, is restricted to files with a *.c* or *.C* extension.

Compile (⌘K)

This command will compile either the contents of the active edit window, or the currently selected file in the project window. The results of the compilation will be posted to the project document.

Only files with *.c* or *.C* extensions can be compiled. To check the syntax of a source file without adding the results to the project document, use the **Check Syntax** (⌘Y) command instead. The **Compile** command will not be active when a library file is selected in the project window.

Load Library

A library is simply a binary representation of a project. There are three types of libraries that can be loaded with this command:

1. A project which has been stored as a library with the **Build Library** command.
2. An MDS *.Rel* file that has been converted to a library using the RelConv utility.
3. A project that has not been saved as a library, but is still in project form.

There is no difference between the first two types of libraries. Each is a single binary file that will be linked as a whole into the project. To distinguish them from projects which are loaded as libraries, libraries of either of these types are given names with a *.Lib* extension. (This is a convention only—libraries can have any legal Macintosh file name.)

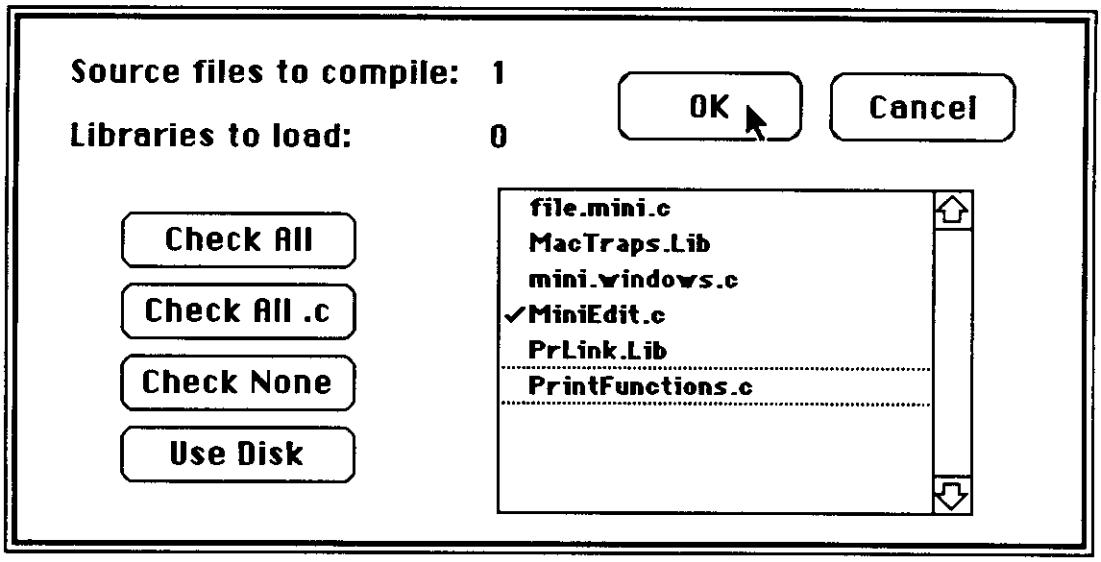
While a project that has not been saved as a library may contain the same object code as one saved with **Build Library...**, it also contains data structures that allow LightspeedC to link it much more intelligently. Only the files in the library that are actually used will be linked into the executable program when you **Build Application...** (or any other project type).

The **Load Library** command is only active when a library file (of any of the three types described above) is selected in the project window. When the selected library file is itself a project, the command changes to **Load Project** on the menu. To add a library to a project for the first time, use the **Add...** command instead.

Add...

This command allows you to add multiple existing source files and libraries to a project. It brings up a standard file box and lets you select a source file, library, or project to be added to the project. It keeps asking for more until you press the **Cancel** button. Only files which can be added (i.e., source files with a *.c* or *.C* extension, libraries, or other projects) are displayed in the scroll box.

Make...
When you select this command from the menu, a dialog box appears showing all the files in the project inside a scroll box. Those files that LightspeedC thinks need to be recompiled (or in the case or libraries, reloaded) are checked. You can edit this list if you like by using the cursor to check or uncheck files, or by clicking the **Check All, Check All .c** or **Check None** buttons. Your changes will not be remembered if you click the **Cancel** button.

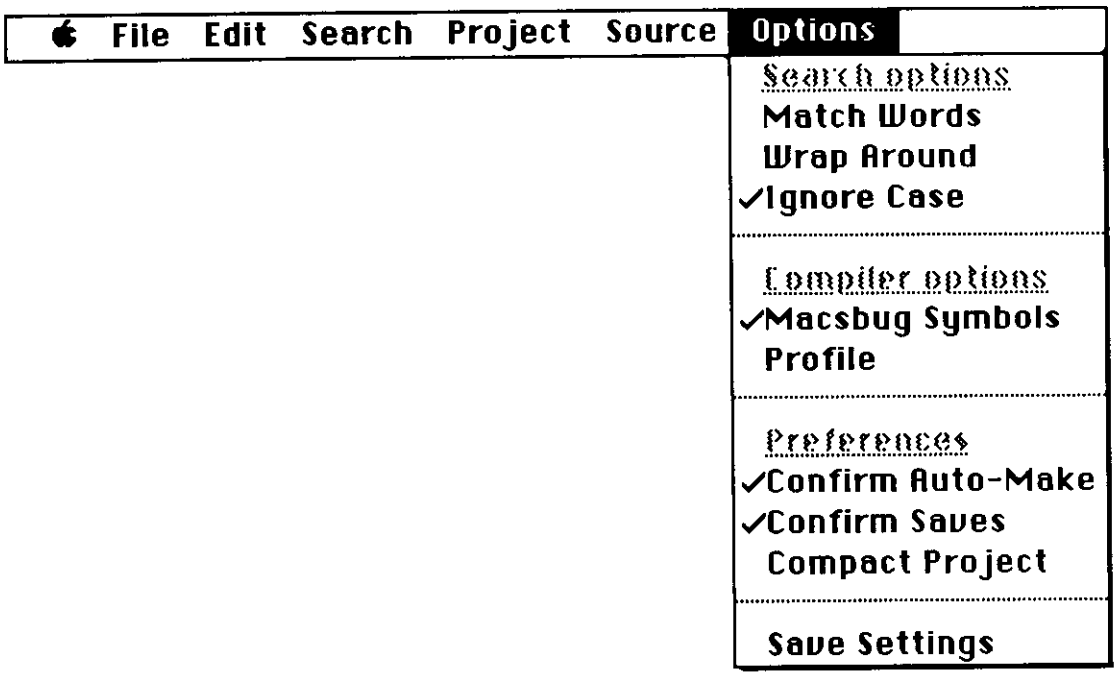


If you click the **Use Disk** button, all current checkmarks will be ignored and LightspeedC will recompute which files need to be recompiled or reloaded based on the date/time-modified of all the files involved. Normally this is not necessary because LightspeedC automatically tracks the changes you make as you edit. This knowledge is project-specific, though, so if you have a source file that belongs to two different projects and you change it in one, the other project won't know it's been changed unless you say **Use Disk.** Also, if a library file changes, you have to click **Use Disk** or explicitly check it to let LightspeedC know.

Clicking **Cancel** does not undo the effect of **Use Disk.** Unlike the other buttons, which simply add (or remove) a check mark to those specified by Auto-Make, **Use Disk** actually updates the date/time record associated with each file that is in the project. You can, of course, manually check or uncheck files after telling LightspeedC to **Use Disk.**

When you click **OK**, any changes to the check-list you have made become permanent, and the specified recompilations/reloadings proceed in batch. You can abort the sequence with ⌘-period. If there are any compile errors the sequence will stop automatically. In either case Auto-Make keeps track of which actions have been completed, so you can issue another **Make...** command to pick up where you left off. Pressing the return or enter key is the same as clicking **OK.**

The Options Menu



The **Options** menu allows you to set three groups of options: search options in the editor, compiler options, and project options. Settings can be made for the current working session, or can be saved as a new set of defaults. Currently selected options are indicated by a check mark next to the option on the menu. Clicking on an option with the mouse will toggle it on or off.

Search Options

The following search options can also be toggled on the dialog box put up by **Find...**.

Match Words
When this option is set, the editor's search and replace functions will work on whole words only. For example, a search for "string" will not match the word "strings." This option is off by default.

Wrap Around
When this option is set, the editor's search and replace functions will search the entire file, rather than from the current position to the end of the file. When the end of the file is reached, the search "wraps around" to the beginning of the file and continues. This option is off by default.

Ignore Case
When this option is set, the editor's search and replace functions will disregard case when performing a search. A search string will match either upper or lower case. This option is on by default.

Compiler Options

Macsbug Symbols
When this option is set, the compiler will produce symbols for Apple's Macsbug assembly language debuggers. Be aware that while Macsbug symbols are useful for debugging, they add 8 bytes to every procedure. This option is on by default.

Profile
When this option is set, the compiler will insert calls to a routine called `_profile_()` into your program. When the program is run, the LightspeedC profiler will produce execution statistics for the program. This option is off by default.

Preferences

Confirm Auto-Make
If you turn this option off you will not be subjected to the **Bring the project up to date?** dialog when you **Run** or issue any **Build...** command; LightspeedC assumes you would answer **Yes**. This option is on by default.

Confirm Saves
If you turn this option off, LightspeedC will automatically save changes to a file that you have modified without asking if you are sure you want to do so. This is a dangerous option to turn off, since it protects you from inadvertently replacing previous versions of your files with newly modified versions. It is, however, very convenient when you want to do program development in quick, incremental steps. This option is on by default.

Compact Project
In order to achieve its remarkable speed, LightspeedC pre-allocates space in the project for anticipated requirements, and does not necessarily free up the space when an item is deleted. As much as 20% of an uncompacted project document can contain unused space. If you turn this option on, the project document will be compacted when you **Close** the project, **Transfer...**, or **Quit** (but not when you **Run**). This option is off by default.

Compaction may be time-consuming and you will normally want to do it only when disk space is at a premium. The amount of space freed will vary.

Save Settings
When you select this option, the current settings of all options will be saved in your copy of LightspeedC, and will become the new defaults whenever you start up the program.

PART TWO

PROGRAMMING AT *LIGHTSPEED*

7
Debugging

Introduction

If all of your programs work perfectly the first time they are run, you should skip this section.

Is everybody still here? Good.

Writing a Macintosh program that works correctly is a rewarding but complex and sometimes frustrating task. To make this ordeal easier, LightspeedC is distributed with MaxBug and MacXLBug, two of the Macsbug family of Macintosh debuggers developed by Apple.

This chapter has three sections: First, an introduction to Macsbug for the first-time user; next, some LightspeedC-specific and advanced Macsbug techniques; and finally some tips to use in special situations. A full summary of Macsbug command syntax appears in Appendix B.

If You've Never Used Macsbug Before

The very first thing to know about Macsbug is that it is an assembly-language-oriented debugger. To use Macsbug well requires a minimal familiarity with the 68000 architecture. A good 68000 reference manual will come in handy. (With its Macintosh Development System, Apple distributes the *M68000 16/32-Bit Microprocessor Programmer's Reference Manual (fourth edition)* by Motorola, published in 1984 by Prentice-Hall.)

To use Macsbug, first install the programmer's switch on the bottom rear corner of the left hand side of your Macintosh if you have not already done so. Pushing the frontmost button on this switch will cause your system to reboot. The rearmost of this pair will cause a non-maskable interrupt which is used by Macsbug to stop the user's program and take control of the processor. On the Lisa, the - key on the numeric keypad generates the non-maskable interrupt.

To install the debugger on your system, first copy the appropriate debugger onto your system boot disk. Use MacXLBug if you've got a Macintosh XL or MacWorks on a Lisa, Maxbug otherwise. If your system uses the Hierarchical File System, the debugger file must be placed in the system folder. Rename the debugger file *Macsbug*. Now reboot your system. Beneath the familiar **Welcome to Macintosh** on the startup screen, the message **Macsbug Installed** will appear, to let you know it's around. (If you have a custom startup screen, you will not see anything different.)

When you push the interrupt switch (remember, it's the REAR one - you should learn this fast!), you will see something like:

```
00F580:                    SUBQ.W  #$2,A7
PC=0000F580 SR=00002000 TM=00000815
D0=0000FFFF D1=00007FFF D2=A0000000 D3=FFFFFFFF
D4=00001200 D5=00000B94 D6=00000020 D7=00000000
A0=00078280 A1=00018826 A2=00000960 A3=0007758E
A4=0000D756 A5=0007771C A6=00076CF8 A7=00076CE0
>
```

The first line is a display of the current program counter (00F580) and the instruction about to be executed. This is followed by a register dump: PC is the program counter, SR is the Status Register, D0 through D7 are the data registers and A0 through A7 are the address registers. A7 is also known as the "Stack Pointer", usually written SP.

At this point, you can use the ` key (at the upper left corner of the keyboard) to switch between views of the program screen and the debugger screen. Enter the G, Go, command and hit return to resume program execution.

Macsbug will also get command of the processor on most system errors, when the system error dialog box (the "bomb" box) would otherwise be seen. With Macsbug, you will be able to do post-mortems when your program crashes. Even if you get a bomb box, you will usually be able to get into Macsbug by hitting the interrupt button. Unfortunately, some crashes are so severe that Macsbug will no longer run.

For an example of this, enter and run the following program with LightspeedC. Before compiling the program, make sure that the **Macsbug Symbols** item is set on the **Options** menu:

```
/* A Sample Program that will cause an odd address crash: */

main()
{
     int *p;              /* declare a pointer */

     p = (int *)0x1;      /* give p the value hex 1, an odd value */
     *p = 0x1234;         /* assign hex 1234 to the address in p  */
                          /* this will cause an odd address error */
}
```

When you run this program, the Macintosh will crash. Macsbug will display something like:

```
ADDR ERR00000001
00DAD6:  MAIN+0016       RTS
PC=0000DAD6 SR=00002000 TM=0000781E
D0=00000000 D1=0000006A D2=00000000 D3=00000040
D4=00000040 D5=0000FFFF D6=494E4458 D7=00000081
A0=00000001 A1=0000D378 A2=00002B6A A3=0000CC58
A4=0000DA74 A5=00071006 A6=00070FFA A7=00070FF6
>
```

(The exact values you see may be different from those displayed above.)

Macsbug has reported an odd address error and reports the offending value as 00000001. (An odd address error occurs when an address register reference is made in a word- or long-mode instruction and that address register contains an odd value. This is one of the most common crashes on the Macintosh—without Macsbug you would get a **System Error ID=02**.) Look at the address register display. Note that the value 00000001 is in address register A0.

Also note the MAIN+0016. Since you compiled your program with the **Macsbug Symbols** option checked, your LightspeedC function names are compiled into your code. Macsbug knows about these function names, and will display them when it displays an instruction in a LightspeedC function. In this case, the processor stopped at offset 16 (hexadecimal) in the function MAIN.

Displaying and Modifying Memory

To list the instruction sequence that led to the crash, type IL MAIN. The IL command requests an Instruction Listing starting at the address given, in this case the symbol MAIN.

```
>IL MAIN
00DAC0: MAIN+0000       LINK    A6, #$FFFC
00DAC4: MAIN+0004       MOVE.L  #$00000001, $FFFC(A6)
00DACC: MAIN+000C       MOVEA.L $FFFC(A6), A0
00DAD0: MAIN+0010       MOVE.W  #$1234, (A0)
00DAD4: MAIN+0014       UNLK    A6
00DAD6: MAIN+0016  PC   RTS
>
```

By examining the listed code, you can see what the problem is. At MAIN+4 the value 1 was assigned into memory, and put into register A0 with the next instruction. The instruction at MAIN+10 tried to assign a word value into the address pointed to by A0, in this case the address 1. This caused the crash. (In this example, as with all address errors, the PC will have advanced an instruction or two past the point of the actual error.)

If you want to view an individual instruction, you use the Instruction Disassembly command ID:

```
>ID MAIN+10
00DAD0: MAIN+0010       MOVE.W  #$1234, (A0)
>
```

You can now return to LightspeedC by using the Exit to Shell command ES.

```
>ES
```

Let's try some more commands. Run your program again. It will crash. Now you can Display Memory with the DM command. The DM command takes two arguments: the address at which to start the display, and the number of bytes to display (this second argument is optional and defaults to hex 10). Remember, all arguments to Macsbug commands are given in hex. We want to dump memory at $FFFC(A6), which translates to "4 bytes before the value in register A6". ($FFFC is hex for -4.) This is the address used in the instructions MAIN+4 and MAIN+C. (In your program, A6 may not be 70FFA. If not, use your value of A6 to calculate the argument you will use for the DM command.)

```
>DM 70FF6
070FF6   0000 0001 0000 0000  0000 C1B0 0040 4970  ............@IP
>
```

Whenever Macsbug expects an address or a value, it will also accept an expression. Macsbug expressions allow addition and subtraction, and also accept certain special constructs. The expressions RA*n* (where *n* is an integer from 0 to 7) mean "the contents of address register *n*". Similarly for RD*n*, which stands for "the contents of data register *n*". For example, to display memory starting at four bytes before the contents of A6, we could just as well have typed:

```
>DM RA6-4
070FF6   0000 0001 0000 0000  0000 C1B0 0040 4970  ............@IP
>
```

or even:

```
>DM RA6+FFFFFFFC
070FF6   0000 0001 0000 0000  0000 C1B0 0040 4970  ............@IP
>
```

(The expression RA6+FFFC won't work, since Macsbug won't sign-extend the FFFC.)

The period . may be used in expressions. It means "the last address used in a DM, SM, IL, or ID command". Try the command "DM .":

```
>DM .
070FF6   0000 0001 0000 0000  0000 C1B0 0040 4970  ............@IP
>
```

Macsbug allows you to dump registers as a group or individually. TD gives a Total Display of the registers. You can display individual data registers with D*n* and address registers with A*n* (where *n* is 0 through 7), the program counter by typing PC and the status register with SR. Try these:

```
>TD
PC=0000DAD6 SR=00002000 TM=0000781E
D0=00000000 D1=0000006A D2=00000000 D3=00000040
D4=00000040 D5=0000FFFF D6=494E4458 D7=00000081
A0=00000001 A1=0000D378 A2=00002B6A A3=0000CC58
A4=0000DA74 A5=00071006 A6=00070FFA A7=00070FF6
>A3
A3=0000CC58
>PC
PC=0000DAD6
```

You can set registers by typing the name of the register and the value you wish to deposit, like this:

```
>A0 70FF6
>PC DACC
>ID PC
00DACC: MAIN+0010       MOVE.W #$1234,(A0)
>TD
PC=0000DACC SR=00002000 TM=0000781E
D0=00000000 D1=0000006A D2=00000000 D3=00000040
D4=00000040 D5=0000FFFF D6=494E4458 D7=00000081
A0=00070FF6 A1=0000D378 A2=00002B6A A3=0000CC58
A4=0000DA74 A5=00071006 A6=00070FFA A7=00070FF6
```

You can Set Memory with SM, which takes as arguments a starting address followed by the data you wish to deposit:

```
>SM 70FF6 00070FF6
>DM .
070FF6   0007 0FF6 0000 0000  0000 C1B0 0040 4970  ............@IP
>
```

Remember, if you go around setting registers and setting memory, you had better know what you're doing. It is possible to "repair" a crashed program by using these commands, but you can easily make things worse by monkeying with the memory or the registers.

Controlling Your Program: Breakpoints and Stepping

The Macsbug command BR *addr* is used to set a BReakpoint at the address given by *addr*. Typing BR without an argument displays a list of active breakpoints. Up to 6 breakpoints may be active at any given time. The command CL with an argument will CLear the breakpoint at the given address; CL without arguments will clear all breakpoints.

Warning: Macsbug doesn't forget about breakpoints when your program terminates. To be safe, make sure to clear all breakpoints when you are through debugging a program.

Use GT *addr* to Go unTil the PC reaches *addr*. (GT is, in effect, a one-time breakpoint that is cleared after it is reached.) Breakpoints and addresses to Go unTil must be in RAM. (Macsbug writes a special trap word into the location at which it wants to stop. The location therefore cannot be in ROM.)

If you want to stop at a particular address in ROM, you must use the command ST *addr*, Step unTil the given location. ST steps one instruction at a time, so it will be slow if many instructions are executed.

Macsbug watches for Macintosh Operating System and Toolbox calls. These calls are commonly referred to as *A-Traps*, since they are invoked by instructions that begin with the hex digit A. There are many commands triggered by A-traps. The basic ones are:

AB - Break on A-Traps
AT - Traces A-Traps
AX - Clears (X-out) all A-Trap commands

Only one A-Trap command may be active at a time.

Most A-Trap commands take six arguments, all of which are optional. The first two arguments are trap numbers that establish a trap range for which the command will apply. (The trap number for an A-Trap is the low-order 9 bits of the instruction. For example, the GetResource call is instruction $A9A0. Its trap number is 1A0.) The trap range arguments default to the range 0..1FE. If just one argument is present, the A-Trap command applies only to that trap.

The second two arguments to an A-Trap command establish the address range for the command. The address range arguments default to "all of memory". A useful range for this is @2AA..@114. (Location 2AA holds ApplZone, the application heap base address; location 114 holds HeapEnd, the top of the application heap.)

The third pair of arguments is the range of values of D0 for which the command is active. This range defaults to 0..FFFFFFFF. You will rarely need to use this pair of arguments.

Macsbug knows the name of every A-Trap, and you can use trap names in lieu of trap numbers. For instance, to break on all GetResource traps use the command:

```
>AB GETRESOURCE
```

To get a feel for how trap instructions work, break into Macsbug and type the commands:

```
>AT
>G
```

You have just told Macsbug to trace all A-Traps in the range 0 to 1FE. You will see the screen flicker wildly as it switches between Macsbug (to list the trap trace) and the current application. Now interrupt the Macintosh by pushing the interrupt switch and holding it until the machine stops. You will see a list of the traps that have executed since you started the trap trace.
Here are some examples of A-Trap commands:

`>AB GETNEWDIALOG`	Breaks on all GetNewDialog traps
`>AT 18D 190 @2AA @114`	Trace all traps between 18D and 190 (GetDItem through SetIText) that occur in the application heap.
`>AX`	Clear the currently active A-Trap Command

Tracing

The Macsbug instruction `T`, Trace allows you to single-step your program. `T` will treat A-Traps as single instructions. If you want to step into an A-Trap, you can use the `ST` *A-Trap name* command.

Sometimes when you're tracing, you'll reach a `JSR` in your code and you won't want to trace through the routine. Instead, you want to just execute the subroutine and break at the statement after the call. There are two ways of doing this. The first is `GT PC+4`. Since `JSR`'s are 4 bytes long, this will have the effect of setting a temporary breakpoint at the return address. The second method uses the command `MR`, Magic Return, which tells Macsbug to continue program execution until the PC reaches the value on the top of the stack. Trace into the function with `T`; the `JSR` will push the return address onto the top of the stack. Now type `MR`. The program will execute until it reaches the function's return address, which is now on the top of the stack.

Warning: When you have reached a `JSR` you wish to skip over, and you have done a `T` to step into the function, the instruction your program is about to execute is a `LINK` instruction. If you want to use `MR`, you must do it now! After you've executed the `LINK` instruction, the return address will no longer be on top of the stack!

Another Warning: If you are tracing through your code and come to a `JSR` when you are expecting code for a `switch` statement, DON'T use the above methods for jumping over the subroutine! This case is covered a bit later in this chapter, under the heading "Tracing Through `switch` Statements".

The Heap

Memory on the Macintosh is allocated for your program from two basic areas, the *stack* and the *heap*. Your program variables and parameters are allocated as needed on the stack. Space for resources and other dynamic objects used by the Macintosh Operating System (including your code) is allocated in the heap, which is generally much larger than the stack. The heap is allocated in blocks which are either relocatable (accessed by a handle—a pointer to a pointer), fixed (accessed by a pointer), or free. The heap can be thought of as a linked list of these blocks, with the linking information carried in the block itself.

While most interaction with this heap memory is controlled by the Toolbox and OS trap calls, you can (and often must) also modify data in the heap directly. One major source of problems in Macintosh programs arises from writing directly into an object in the heap without regard to the boundaries of the object. It is possible to write over the link information of the following block of memory, causing an inconsistency of the heap. Program behavior is quite unpredictable after such an indiscretion, and the eventual crash often occurs long after the offending code has executed.

Since the heap is of such central importance to Macintosh programs, Macsbug provides commands to view this data structure.

HD gives a Heap Dump, a listing of all of the blocks of the heap:

```
0000C000

0000C034 P 00000108              *
0000C13C H 0000063C A 0000C118 * 20 0001 CODE
0000C779 P 0000000C              *
   ...
00015186 F 0000166E

 HLP PF 0004 0002 0000 00000000  0004 000096F6
```

The first number you see in the heap display is the address of the heap base. Following this, the blocks of the heap are listed. The first number of a heap block display is the starting address of the block. Next comes a letter: P, H, or F which stand for Pointer (non-relocatable) block, Handle (relocatable) block or Free block, respectively. The next number is the size of the block. The next two fields apply to relocatable blocks. The first number is the attribute byte of the master pointer, followed by the master pointer location. (Master pointer attributes are explained in the "Memory Manager" chapter of *Inside Macintosh*.) The asterisk * denotes a locked or non-relocatable block. The final three fields apply to resources only. The first of these fields is the reference number of the resource file, the second number is the resource ID and the third field is the resource type.

The final line of the Heap Dump is the "Totals" line. The six characters 'HLP PF' represent the six values which follow: number of Handle (relocatable) blocks, number of Locked relocatable blocks, number of Purgeable blocks, the space (in bytes) occupied by purgeable blocks, the number of Pointer (non-relocatable) blocks and the Free space available in the heap.

HT will give the Heap Total which is just the last line of the Heap Dump.

HC will check the consistency of the heap, and print the offending location in the heap if there's a problem.

Warning: Sometimes HC will get into an infinite loop if the heap is really in a mess. If you think this may be the case, use HD. You can abort the HD command by hitting the backspace key.

HD and HT take an optional argument to limit the scope of the command:

'H':	Only the Handle (relocatable) blocks
'P':	Only the Pointer (non-relocatable) blocks
'F':	Only the Free blocks
'R':	Only Resource blocks
'xxxx':	Only resource blocks of type 'xxxx'

Terminating the Program

Finally, if you wish to terminate your program execution, you have three options. `EA`, Exit to Application will re-launch your program. `ES` will Exit to the Shell program. If you launched your program from LightspeedC with the **Run** command, LightspeedC is the shell; if you launched your program from the Finder, then that is the shell. `RB` will cause the Macintosh to ReBoot. Sometimes you will try `ES` and you will get a further Macintosh exception. In that case, the content of your Macintosh's memory is probably so inconsistent that the route to the shell is lost, and you must reboot.

Debugging a LightspeedC Program

So you've written your first LightspeedC program, and it doesn't work. Perhaps it just doesn't behave correctly; perhaps it crashes. Don't just sit there! Break out your Macsbug! Here are some tips that may help.

First of all, remember to set **Macsbug Symbols** on the **Options** menu. This will allow Macsbug to display your function names in its instruction listings. You'll also be able to use these symbols in expression arguments to commands like `BR`, `GT`, `IL` and `ID`.

Warning: Beware of name conflicts! While your function names may not conflict as far as LightspeedC is concerned, Macsbug may have problems telling them apart. Macsbug truncates names at eight characters and all symbols are upper-case. In addition, Macsbug will not handle names with underscores, so LightspeedC removes them from your names before placing them in your code. Another problem Macsbug has in this area is the way it deals with trap names. When Macsbug is interpreting a symbol, it looks first to its list of A-Trap names. If the name it is searching for is the same as the start of a trap name, it will interpret the symbol you entered as that trap name. For example, there is a trap named `ErrorSound`. If you have a function named `Error` and you attempt to use `Error` in a Macsbug command, Macsbug will mistake it for `ErrorSound`. If you type `BR Error`, meaning to place a breakpoint at the beginning of your error routine, Macsbug will try to place the breakpoint at `ErrorSound`, a location in the ROM. This will not have the effect you desire. To check, you can do an `IL` *symbol* to see how Macsbug will interpret your symbol.

In addition, LightspeedC provides two function calls that allow you to place Macsbug breaks into your source code:

`Debugger()` causes a Macsbug break.
`DebugStr` (*Pascal string*) causes a Macsbug break and outputs the *Pascal string*.

Now you are armed to hunt bugs. If you wish to investigate a particular function, insert the `Debugger()` call at the beginning of that function or set a breakpoint at the function name. Macsbug will break at the start of your function and you can trace through it. You can get Macsbug to display a specific string by using the `DebugStr` statement. For example:

```
DebugStr("\pHi There!");
```

If you want to see some specific data when you break, you can use a code sequence like:

```
char Observe[80];              /*length doesn't have to be 80*/

/* first convert variable values to a C string */
sprintf(Observe, /*formats and variables*/);

/* then convert the C string to a Pascal string, break into */
/* the debugger and print the Pascal string */
DebugStr(CtoPstr(Observe));
```

System errors when in the ROM are among the toughest nuts to crack. Try to determine what A-Trap your program was executing and what was the last statement your program executed before it crashed. You can use AR to determine the last A-Trap Remembered by typing AR 0 1FE before running your program. When the program crashes, enter AR. Macsbug will display the last trap that was executed before the crash. Run your program again and break on this trap. Check the arguments for this trap carefully.

To find out the last trap executed in RAM before a crash, use the above method, but enter AR 0 1FE @2AA @114 instead of just AR 0 1FE.

A tip: If your program crashes on a Memory Manager call, check to see if the program is trying to do something like set the size or dispose of a handle whose value is 0.

Tracing Through switch Statements

Tracing through your code is usually straightforward, but there are some tricky cases. One arises when you try to trace through a `switch` statement. When you get to your `switch` statement you will find a JSR to an A5-relative address:

```
JSR          xx(A5)
```

Trace into the call and do an IL PC. Look through the dumped instructions until you see an indexed indirect jump: JMP *x*(A0,D0.W). Do a GT to this instruction's location, and then trace with T. You are now in the code selected by your `switch` statement.

Another tricky tracing case is tracing inter-segment calls when the segment you're jumping into is not yet in memory. You will be tracing through your code and encounter a JSR xx(A5). When you will trace, one of two things will happen. The next instruction might be a JMP to some 32-bit address, or you will encounter Toolbox $A9F0; LOADSEG —the load segment trap—after tracing two instructions after the original JSR. If you get the JMP instruction, the segment you're jumping into is already loaded, so just continue tracing.

If you get the LoadSeg trap, DON'T use the T instruction to step over the trap to the next instruction, it won't work, since LoadSeg sends the PC somewhere else. To get out of this plight, set memory location $12D to 1 (SM 12D 1), and go (G). Macsbug will break at an RTS instruction. Trace one instruction and you will be pointing at a JMP to some absolute address. Trace again, and you will step into the function in the newly loaded segment. If you don't clear

location $12D (SM 12D 0), the next LoadSeg call will be trapped just before the next newly loaded segment is entered.

This feature of LoadSeg can be used to stop at the first statement in your LightspeedC program. Before you **Run** your program from LightspeedC, break into Macsbug, set $12D to 1 and go. When your program is launched, the first LoadSeg call loads some initialization code. You will break at the RTS which sends you to the JMP into this code. Go again and you will break at another RTS. Trace twice, and you're at the first statement in your program.

To view variables and parameters in your program, it is necessary to understand LightspeedC calling conventions which are covered in greater depth in Chapter 9, "Running at Lightspeed". Here's a brief overview.

Nearly all functions in LightspeedC begin with a LINK A6, #*n* instruction. This allows you to view local variables and parameters as A6-relative. Parameters are located starting at RA6+8 (or RA6+12 if your function returns a double, struct or union). The leftmost parameter of your function is at the lowest address. The return address is located at RA6+4. Local variables are located at negative offsets to A6. Global variables are stored at negative offsets to A5.

If you're interested in looking at a few particular variables, you can make them register variables. These variables will then show up in the register dump during tracing. It also makes it easier to understand what the disassembled code does.

Advanced Macsbug Commands

These are some miscellaneous useful Macsbug commands.

SC is the Stack Crawl command. This command takes advantage of the fact that functions start with LINK A6 instructions to display a list of the active stack frames and the calling points of active functions. It is very useful for post-mortem analysis; you can see what functions got called with what arguments, and many times deduce the reason for the program crash. (This is not always useful if your program dies in a memory manager call—since the memory manager uses A6.) Here's an example of the SC instruction's output.

```
>SC
SF @06FFDA  UPDATE+2C
SF @070004  MAIN+5A
```

In this example, the currently executing function's stack frame is located at 6FFDA, and was called from offset 2C in function UPDATE, which in turn has a stack frame at 70004, and was called from MAIN at offset 5A.

The CV, ConVert, instruction is an all-purpose programmer's calculator. It displays expressions in four ways: as an unsigned hexadecimal, as a signed hexadecimal, as a decimal integer and as a four-character ASCII string. The CV command, used with expressions, is your calculator.

```
>CV MAIN
$0000CFE0  0000CFE0  &53216  '....'
>CV 5A-MAIN
$FFFF307A  -0000CF86  &-53126  '..OZ'
>CV -&10
$FFFFFFF6  -0000000A  &-10 '....'
```

The WH or WHere command provides an index into the Toolbox and OS routines in the ROM. WH with a parameter less than 512 will give the trap name and address of the trap routine in the ROM. WH with a parameter greater than or equal to 512 will give the name and address of the "nearest" trap routine.

```
>WH GETRESOURCE
01A0 40D994 GETRESOURC
```

01A0 is the trap number for GetResource. (The actual trap instruction is $A9A0.) The code for GetResource starts at 40D994.

If you bomb in the ROM, the command WH PC will tell you what the nearest trap routine is to the PC. This may or may not be the trap that your program crashed in. If the result of WH PC is the routine HNOPURGE, this means your program crashed somewhere in the memory manager. If the result is something else, that is probably the trap your program crashed in. To make sure, enter AR 0 1FE and run your program again until it crashes. Then enter AR. Macsbug will tell you the last trap that executed.

The SS, Step Spy command is useful if you have some memory that's changing, but you don't know where it's getting changed. The command SS *addr1 addr2* will check the contents of all addresses in the range between *addr1* and *addr2* before each instruction is executed. If the contents of any locations in the range change, your program is halted and you drop into Macsbug.

Warning: As you might suspect, this instruction is slow. (But it's effective.)

There is a corresponding A-Trap Spy command AS *addr1 addr2* which monitors the address range at every trap call instead of every instruction.

The command AR is used to Remember A-Traps. AR takes the same arguments as AB or AT, and remembers the last trap that meets the condition set. Typing AR without arguments will display the last trap executed that meets the conditions set. Remember: you must set the AR command in advance. Once you've crashed, it's too late for AR to help you.

8
Using Resources

Introduction

A typical Macintosh application uses *resources* to define its menus, windows, dialogs, alerts, strings, pictures, and icons. In addition, the application's code itself consists of resources. These objects reside together in a single resource file. This chapter tells you a few special things you need to know to use resources with LightspeedC.

(If you are new to programming the Macintosh, it is important to learn about resources right away, as they pervade every aspect of the Macintosh Toolbox. Refer to *Inside Macintosh* or *Macintosh Revealed* for detailed information.)

Using a Separate Resource File

LightspeedC allows you to run a program without first building an application file. That, of course, is part of why it is so fast, but where do your program's resources go?

The solution is to use a separate resource file. Place your resources in a file called, for example, *myProg.rsrc*. (A good convention for separate resource files is to end them with *.rsrc*, but the choice is up to you.) Then insert the following statement near the beginning of your program:

```
OpenResFile("\pmyProg.rsrc");
```

Your resources will now be available. Except for this one statement, your program can take exactly the form it would if all your application's resources, including its code, were together in one file. Later when you do combine everything together into one file, your program will work without change. (You can remove the `OpenResFile` call at that time, if you choose.)

Combining Resource Files

When you are ready to prepare a stand-alone version of your program—that is, one which can be launched from the Finder rather than only from LightspeedC—you issue the **Build Application...** menu command. This creates a resource file containing the code resources that constitute your application. You can now launch this file, either from the Finder or using LightspeedC's **Transfer...** menu command, and it will work exactly as it did when you ran it using LightspeedC's **Run** command. But it will still be getting its resources from the separate resource file.

Apple's RMaker utility (supplied with LightspeedC and described in Appendix B) has a feature that enables you to easily merge resource files. If the file you created with **Build Application...** is called *myProg*, the following RMaker script will do the trick:

```
!myProg
APPLMINE

include myProg.rsrc
```

That's all there is to it. The ! in front of `myProg` tells RMaker to append to an existing resource file rather than create a new one. You should substitute your own creator letters for `MINE`.

If you are a purist, before issuing the **Build Application...** command you should remove the `OpenResFile` call that opens the separate resource file. But no serious harm will befall you if you leave it in. If the separate resource file is around when `OpenResFile` is called, resources will be found there just as they would if the application did not also supply them. And if the separate resource file isn't around, the `OpenResFile` call will fail but all resource requests will be met by the resources in the application itself.

An Example

Here is a sample program, written without using resources, that just opens a window and waits for you to click its "go-away" box.

```
#include "WindowMgr.h"
#include "EventMgr.h"

main()
{
    WindowPtr myWindow, whichWindow;
    EventRecord myEvent;
    static Rect bounds = { 100, 100, 300, 300 };

    InitGraf(&thePort);
    InitFonts();
    InitWindows();
    InitCursor();
    myWindow = NewWindow(0, &bounds, "\pSample Window", 1, 0,
                                -1, 1, 0);
    for (;;) {
        GetNextEvent(everyEvent, &myEvent);
        if (myEvent.what == mouseDown
        && FindWindow(myEvent.where, &whichWindow) == inGoAway
        && whichWindow == myWindow
        && TrackGoAway(myWindow, myEvent.where))
            return;
    }
}
```

Here is the same program written using resources:

```
#include "WindowMgr.h"
#include "EventMgr.h"

main()
{
    WindowPtr myWindow, whichWindow;
    EventRecord myEvent;

    InitGraf(&thePort);
    InitFonts();
    InitWindows();
    InitCursor();
    OpenResFile("\pSample.rsrc");
    myWindow = GetNewWindow(128, 0, -1);
    for (;;) {
        GetNextEvent(everyEvent, &myEvent);
        if (myEvent.what == mouseDown
        && FindWindow(myEvent.where, &whichWindow) == inGoAway
        && whichWindow == myWindow
        && TrackGoAway(myWindow, myEvent.where))
            return;
    }
}
```

Here is the RMaker script for the resource file *Sample.rsrc*. Of course, you could use ResEdit (supplied with LightspeedC and described in Appendix C) to create the file instead.

```
Sample.rsrc      ;; output file
????????         ;; file type and creator

Type WIND
  ,128           ;; resource ID
Sample Window
100 100 300 300  ;; window bounds
Visible GoAway
0                ;; a regular document window
0                ;; the refCon (not used in the sample program)
```

When you are ready to build a stand-alone application, optionally remove the `OpenResFile` call, use **Build Application...** to create a file called *Sample*, and run the following RMaker script:

```
!Sample
APPLSAMP

include Sample.rsrc
```

9
Running at Lightspeed

Introduction

This chapter takes a look at how code generated by LightspeedC communicates with the Macintosh software environment. It contains the information you need to know to interface with Pascal and assembly language, to use the Macintosh Toolbox and Operating System, and to write desk accessories, device drivers, and code resources.

Pascal, not C, is the Macintosh's "native" language. The Macintosh system software expects to be called from Pascal. (Some routines are designed to be called from assembly language, but a Pascal interface is also provided.) Similarly, when the system software in turn "calls back" functions that you provide, such as action procedures and filter procedures, it calls them as though they were written in Pascal. Communicating with the Macintosh means conforming to Pascal calling conventions which are different from those appropriate to C.

LightspeedC provides a way for you to write functions that act as though they were written in Pascal, and to call functions that expect to be called from Pascal. This is accomplished through the use of the `pascal` keyword. `extern` functions may be declared `pascal`; they will be called using Pascal calling conventions. Function definitions may be declared `pascal`; they will expect to be called using Pascal calling conventions. The Macintosh system routines are built-in and will always be treated as Pascal functions.

Keeping C and Pascal on speaking terms can be tricky, but LightspeedC tries to make it as painless as possible.

Of course, down deep the real native language of the Macintosh is MC68000 assembly language. This chapter describes both the C and Pascal calling conventions at the machine level, and tells how you can incorporate assembly language files into a project using the MDS assembler.

Not all programs you may wish to write for the Macintosh will take the form of applications. Some may be desk accessories, device drivers, or any of a host of code resources such as FKEYs, packages, INIT resources, or custom window, menu, or control definition procedures. LightspeedC makes it easy.

C Calling Conventions

C Function Entry

The arguments are passed on the stack. The first (leftmost) argument appears just above the return address, followed by the remaining arguments in order. Note that the arguments are pushed in reverse order of their appearance in the parameter list.

The stack looks like this on entry (just after the call):

```
            arg-N
            ...
            arg-1
SP ->       return address
```

Thus the first argument can be found at `4(SP)`. If the function begins with a `LINK A6,#...` instruction, the first argument can also be referred to as `8(A6)`.

(When writing an assembly language function, it is up to you whether to use a `LINK A6,#...` instruction. Such an instruction will appear in code generated by the LightspeedC compiler, except for functions with no arguments, no local storage, and no compiler-generated temporaries; in other words, if there is nothing to address off of A6, then A6 is not set up. However, if either the **Macsbug Symbols** option or the **Profile** option is checked, a `LINK A6,#...` instruction is always generated.)

All arguments occupy an even number of bytes on the stack. A byte argument is converted to a word by the caller and may be addressed as a word at the appropriate offset `d(SP)` or as a byte at `d+1(SP)` (the low byte of the word).

C Function Exit

The return result (if any) is returned in D0. The result may be 1, 2, or 4 bytes in length; unused high-order bits may contain garbage.

It is the caller's responsibility to remove the arguments from the stack.

The stack looks like this on exit (just after the return):

```
            arg-N
            ...
SP ->       arg-1
```

An alternate method is used to return `structs`, `unions`, and `doubles`; see "Functions Returning `struct`, `union`, or `double`" further on in this chapter.

C Calling Sequence

The caller pushes the arguments in right-to-left order, then calls the function. Upon return, it must remove the arguments from the stack. Thus the caller's code looks something like this:

```
MOVE    ...,-(SP)       ;  last argument
...
MOVE    ...,-(SP)       ;  first argument
JSR     function
ADD     #...,SP         ;  total size of arguments
```

The function's code looks something like this:

```
LINK    A6,#...         ;  (optional)
...
MOVE    ...,D0          ;  result
UNLK    A6              ;  (optional)
RTS
```

Functions Returning struct, union, or double

An alternate method is used to return a result of type `struct`, `union`, or `double`, since values of these types are in general too large to fit in D0. (Some `structs` or `unions` may be small enough to be returned in D0, but the alternate method is used anyway.)

After pushing the arguments but before issuing the actual call, the caller pushes the address of the location where the return value is to be placed. This address appears at `4(SP)` (or `8(A6)`) and the first argument appears instead at `8(SP)` (or `12(A6)`). The function must obtain this address and store the result at the location pointed to. The address is considered a hidden argument to the function, and it is the caller's responsibility to remove it from the stack.

Because a function returning a `struct`, `union`, or `double` expects its caller to have placed a hidden argument on the stack, it is essential that the caller do so! Therefore, even when you are not interested in the actual return value, always be sure that the function is declared correctly before calling it.

Functions That Accept a Variable Number of Arguments

The C calling conventions are designed to facilitate writing functions with a variable number of arguments. The first argument(s) can always be found in the same place regardless of how many additional arguments are supplied. And the responsibility for removing the arguments from the stack lies with the party that knows how many arguments were actually passed, the caller.

As an elementary example, here is a function that returns the minimum of an arbitrary number of integers. The first argument is the number of additional arguments passed and must be at least 1.

```
int minimum(count, x)
int count, x;
{
    int *xp = &x;     /*  pointer to arg list  */

    while (--count) {
        if (*++xp < x)
            x = *xp;
    }
    return(x);
}
```

The ability to write functions taking a variable number of arguments, without having to resort to assembly language, is so useful that the calling conventions that make it possible have become an unofficial standard for C. To the extent that most C compilers support these conventions, code like the "minimum" function above is portable. Occasionally you may run into a compiler that does not conform; *caveat emptor.*

A function using Pascal calling conventions cannot accept a variable number of arguments, unless it is written in assembly language.

Pascal Calling Conventions

Pascal Function Entry

The arguments are passed on the stack. The last (rightmost) argument appears just above the return address, followed by the remaining arguments in reverse order. If the function returns a result, space for it is reserved above the first argument. If the return value is 1 byte long, 2 bytes are reserved.

The stack looks like this on entry (just after the call):

```
        space for return value (if any)
        arg-1
        ...
        arg-N
SP ->   return address
```

Thus the last argument can be found at `4(SP)`. (If the function begins with a `LINK A6,#...` instruction, the last argument can also be referred to as `8(A6)`.)

(When writing an assembly language function, it is up to you whether to use a `LINK A6,#...` instruction. Such an instruction will appear in code generated by the LightspeedC compiler, except for functions with no arguments, no local storage, and no compiler-generated temporaries; in other words, if there is nothing to address off of A6, then A6 is not set up. However, if either the **Macsbug Symbols** option or the **Profile** option is checked, a `LINK A6,#...` instruction is always generated.)

All arguments occupy 2 or 4 bytes on the stack. A byte argument appears in the high byte of its word and is found at an even offset from SP (or A6).

Pascal Function Exit

The function stores its return result (if any) on the stack in the location reserved by the caller. If the result is 1 byte long, it is placed in the high byte of the word reserved.

It is the function's responsibility to remove the arguments from the stack.

The stack looks like this on exit (just after the return):

```
SP ->  result (if any)
```

Pascal Calling Sequence

The caller pushes the arguments in left-to-right order, then calls the function. Upon return, the result (if any) may be found on the stack. Thus the caller's code looks something like this:

```
SUB     #...,SP          ;  reserve space for result
MOVE    ...,-(SP)        ;  first argument
...
MOVE    ...,-(SP)        ;  last argument
JSR     function
MOVE    (SP)+,...        ;  result
```

The function's code looks something like this:

```
LINK    A6,#...          ;  (optional)
...
UNLK    A6               ;  (optional)
MOVE    (SP)+,A0         ;  return address
ADD     #...,SP          ;  total size of arguments
MOVE    ...,(SP)         ;  store return result
JMP     (A0)
```

Type Considerations

It is not always obvious from a Pascal parameter declaration what the actual argument passed on the stack is supposed to be. You must be careful, when calling a Pascal function, to pass the correct arguments, and when writing a Pascal function, to declare the arguments correctly.

The basic rule is that if an object is larger than 4 bytes, or if it is a VAR parameter, the argument passed is the address of the object; otherwise the object itself is the argument. In Pascal this happens invisibly, but must be managed explicitly when interfacing to Pascal from Lightspeed C. When calling a Pascal function, remember to use the address operator (&) where appropriate; when writing Pascal functions, declare arguments to be pointers where appropriate.

For integer arguments, be sure to pass the correct size. Byte values are not automatically extended to words as they are for C functions. The Pascal `CHAR` type is actually a word type, so pass an `int`, not a `char`. The Pascal `BOOLEAN` type is a byte type, so pass a `char`. (When calling built-in Macintosh routines, LightspeedC handles sizing of integer arguments automatically.)

The Pascal type `PACKED ARRAY[1..4] OF CHAR` is frequently used to specify resource types and file types. Though in Pascal this is an array type it should not be treated like a C array type. Since it is only 4 bytes long it is passed by copying it onto the stack, and should be declared in C as a `long`. Multi-byte character constants may be used for this purpose, e.g. `'STR#'` or `'TEXT'`.

The Quickdraw data type `Point` is only 4 bytes long and so is passed by copying it onto the stack. Don't pass its address instead unless it is a `VAR` parameter. Most `RECORD` (`struct`) types, however, are larger than 4 bytes; an address must be passed even for a non-`VAR` parameter.

When writing a Pascal function, be aware that non-`VAR` parameters are supposed to be passed by value, not by reference, even though the argument actually passed may be the address of the real parameter. The parameter itself must not be modified; if necessary the function should make a local copy of the parameter.

Because the caller must reserve stack space for the return result, it is critical to declare the proper return type when calling a Pascal function. Functions that do not return a value (`PROCEDURE`s in Pascal-ese) should be declared to return `void`. (The built-in Macintosh routines are pre-declared, so you don't need to worry about this.)

The Macintosh Interface

The Macintosh Toolbox and OS interface routines are built-in, and will automatically be called using the Pascal calling conventions without your needing to declare them. What is more, LightspeedC knows the expected number of arguments and the sizes of each argument. If the argument you supply is the wrong size, it will be adjusted to the proper size provided that it is an integer. LightspeedC does not know the expected types of the arguments, only their sizes, and will not complain if you pass the wrong type. It will complain, though, if you pass a non-integral argument that is the wrong size, or if you pass the wrong number of arguments.

Most Macintosh calls are implemented by traps that use the Pascal calling conventions directly; LightspeedC generates these traps inline rather than by issuing a subroutine call. Other calls expect their arguments in machine registers, or are not implemented by traps at all; for these LightspeedC generates calls to library functions. All the calls are made using the Pascal calling conventions.

For the built-in routines LightspeedC knows the sizes of the return values, but it does not know their actual types so it assumes the values are integers, i.e. either `char`, `int`, or `long`. (Of course, for functions that do not return a value the return type is `void`.) The `#include` files supplied with LightspeedC contain preprocessor macros that cast return results to the proper type when necessary. For example,

```
#define GetResource     (Handle) GetResource
```

causes the result from `GetResource` to be correctly considered a `Handle`. Without this definition the result would be considered a `long`.

Making Indirect Calls

Only functions that are declared `pascal`, and Macintosh built-in routines, will be called using Pascal conventions. Functions that are called indirectly through pointers to functions are always called using C calling conventions. If you have a pointer to a Pascal function, you can call the function by using one of the library functions `CallPascal`, `CallPascalB`, `CallPascalW`, or `CallPascalL`.

For example, suppose that `pf` is a pointer to a C function and `pg` is a pointer to a Pascal function; both functions accept two `int` arguments and return `void`. Both pointers may be declared the same way:

```
void (*pf)(), (*pg)();
```

The function pointed to by `pf` can be called, for example:

```
(*pf)(5, 7);
```

But the function pointed to by `pg` must be called:

```
CallPascal(5, 7, pg);
```

Note that the pointer must be the last argument to `CallPascal`.

Interfacing with Assembly Language

Assembly-language source code can be assembled by the MDS assembler, converted to a LightspeedC library, and loaded into a project.

Accessing Global Symbols

Symbols defined in assembly language using the `DS` directive and exported using the `XDEF` or `XREF` directive may be accessed from C as `extern` variables.

Symbols defined in C as global variables may be imported using the `XREF` directive and accessed from assembly language. The `(A5)` addressing base must be supplied explicitly just as it must for symbols defined using the `DS` directive.

Function names defined in assembly language and exported using the `XDEF` or `XREF` directive may be called from C.

Function names defined in C may be imported using the `XREF` directive and called from assembly language. The `(A5)` addressing base must not be supplied.

Register-Saving Conventions

Data registers D0-D2 and address registers A0-A1 are scratch registers. Other registers must be preserved across calls. This applies whether C or Pascal calling conventions are used.

Converting *.Rel* files

The MDS assembler translates assembly-language source files to *.Rel* object files. A *.Rel* file must be converted to a LightspeedC library before it can be added to a project. A utility program, RelConv, is provided for this purpose.

RelConv presents a standard file dialog allowing you to specify a *.Rel* file to be converted. It does this repeatedly, converting each file you specify, until you click the **Cancel** button. Each library created is given the same name as the *.Rel* file, but with the *.Lib* extension replacing the *.Rel* extension.

MDS *.Rel* files do not retain case information present in the assembly-language source for imported and exported symbols. RelConv assumes by default that all symbols should be lower-case. If you want upper-case characters in your symbols, you can supply a "vocabulary" file. If RelConv sees a file with the same name as the file it is converting but with the *.Voc* extension, it is considered to be a vocabulary file containing a list of symbols, one per line, with the desired capitalization. Each symbol found in the *.Rel* file that matches the spelling of a symbol in the vocabulary file will appear with the specified capitalization in the resulting library. Symbols not found in the vocabulary file will appear in lower-case.

To assist you in building a vocabulary file when converting a *.Rel* file for the first time, RelConv, if it finds no vocabulary file, will create one containing all the symbols in the *.Rel* file in lower-case. You can then edit this file to supply the desired capitalizations and run RelConv again.

Assembly language files using the `RESOURCE` directive cannot be converted by RelConv. This directive is usually used to create resources containing code. LightspeedC has its own mechanisms for accomplishing this; see the sections "Desk Accessories and Device Drivers" and "Code Resources" that follow. (You can use the Apple utilities RMaker and ResEdit to create other kinds of resources.)

Desk Accessories and Device Drivers

There are two kinds of Macintosh drivers—desk accessories and device drivers. In this chapter the term *driver* is used to refer to either kind.

Writing a driver is a very different task from writing an application. This section does not attempt to teach you all about writing drivers; refer to the "Device Manager "and "Desk Manager" chapters of *Inside Macintosh* for this information. Assuming you know how to write a driver, this section tells you how to write one in LightspeedC.

The Device Manager expects drivers to be written in assembly language. When a driver written in LightspeedC is called by the Device Manager, a short assembly-language stub receives control and translates the Device Manager's request into a call to the C function `main`. This stub is placed in the driver automatically by LightspeedC.

The stub also performs two additional services designed to make writing a driver easier. It sets up a data area so that your driver can have its own global variables; and it automatically figures out the proper way to return control to the Device Manager. These services are discussed later in this section.

How a Driver is Called

The driver's `main` function is called with three arguments. The function must return an `int`, and it must not be declared `pascal`.

The first argument is a pointer to an I/O parameter block. (This is the value that is passed in address register A0 to the assembly-language entry point of the driver.)

The second argument is a pointer to the driver's device control entry. (This is the value that is passed in address register A1 to the assembly-language entry point of the driver.)

A driver written in assembly language actually has as many as five different entry points; a driver written in LightspeedC only has one. Therefore a third argument is passed to `main`, a small integer indicating which entry point was really called. This selector has the following values:

```
0       Open
1       Prime
2       Control
3       Status
4       Close
```

The body of your `main` function should be a `switch` statement that dispatches based on the value of this selector.

How a Driver Returns

The function `main` should return `0` to indicate successful completion, and a non-zero result code to signal an error condition. You do not have to place a copy of the return value in the `ioResult` field of the I/O parameter block; this is done for you.

A special result code of `1` should be returned from asynchronous calls to `Prime`, `Control`, and `Status` routines if the request could not be completed right away. This result code will be stored in the `ioResult` field of the I/O parameter block, but `0` (no error) will be returned to the Device Manager.

How a Driver Returns, Continued—the jIODone Problem

One of the trickiest aspects of returning from a driver is deciding whether to return directly to the Device Manager (via an `RTS` instruction) or whether to jump to `jIODone`. This is a complex issue and many existing desk accessories do it wrong (though, fortuitously, they manage to work anyway). Following is an explanation of this problem, and a description of what LightspeedC

does about it. If you are not interested in these details, you may skip them; suffice it to say that drivers written in LightspeedC automatically do the right thing without programmer intervention.

First, some background. Associated with each driver is an I/O queue, which is a list of I/O parameter blocks waiting for service from the driver. Calls made to a driver fall into one of two categories: *queued*, meaning that the I/O parameter block passed as an argument to the call is one of the ones on the driver's queue; and *immediate*, meaning that it is not. In the latter case, the queue may even be (and in fact usually is) empty.

All `Open` and `Close` calls are immediate. All `Control` calls made to desk accessories are immediate, with the exception of the "goodbye kiss" (`csCode=-1`) issued to desk accessories who have requested to be notified when the current application exits out from under them. Other calls may be queued or immediate.

Now the rules for returning from a driver are as follows: The driver should return directly to the Device Manager from all immediate calls. It should also return directly to the Device Manager from queued calls requesting asynchronous I/O that could not be completed right away. Finally, it should jump to `jIODone` from queued calls if the driver completed the request (or if there was an error).

It is incorrect to violate these rules; in particular, it is incorrect to jump to `jIODone` to return from an immediate call. `jIODone` will attempt to examine the driver's I/O queue, and since the queue is usually empty it will end up examining low-memory locations beginning at `$0000`. Apparently, these locations somehow look enough like an I/O parameter block to satisfy the Device Manager, but this is clearly an unsafe situation.

Just to make things difficult, when returning from `Prime`, `Control`, and `Status` calls it is `jIODone` that unlocks the driver's code and its device control entry so they won't be islands in the heap between calls to the driver (unless, of course, the driver has requested they remain locked). So the writer of a desk accessory, for instance, has to make a difficult decision—he can return directly to the Device Manager, leaving the driver's code and its device control entry locked and potentially interfering with the host application; or he can violate the rules and jump to `jIODone`. Most desk accessories seem to take the latter route.

LightspeedC avoids this dilemma. When a driver written in LightspeedC returns from `main`, the decision whether to call `jIODone` is made automatically (and correctly). For `Prime`, `Control`, and `Status` calls, if the decision is made to return directly to the Device Manager, and the driver has not requested that its code and device control entry remain locked, they are unlocked.

The Data Area

In LightspeedC, drivers may declare and initialize global (including `static`) variables. (However, initializers that evaluate to addresses are not allowed.) The data is allocated dynamically before `main` is called to implement the `Open` entry, and deallocated after `main` returns from the `Close` call.

A driver must co-exist with the currently running application, and so cannot use address register A5, as the application does, to address its globals. Address register A4 is used instead. It is essential to make sure that any libraries that you include in the project access globals off of A4. For

example, you cannot use the Standard I/O Library in a desk accessory unless you prepare a special desk-accessory version. (Conversely, libraries intended for use with drivers must not be used in applications.)

The MacTraps library does not access any globals and so is compatible with both applications and drivers. It does define some globals, namely the Quickdraw globals, but never accesses them itself. If you access these globals from a driver, you will find that they are not the real Quickdraw globals but simply the driver's own variables that Quickdraw knows nothing about.

(It is possible to reach the real Quickdraw globals from a LightspeedC driver by observing that `0(A5)` holds the address of the last of the Quickdraw globals, `thePort`. The remaining Quickdraw globals may be found at descending addresses from `thePort`; refer to the "Quickdraw" chapter of *Inside Macintosh* for more information. The value of A5 is accessible from C as the low-memory global `CurrentA5`.)

A handle to the dynamically allocated data area is stored in the `dCtlStorage` field of the driver's device control entry. This handle is dereferenced into address register A4 and locked before each call to `main`. It is not unlocked automatically; if you are willing to allow the data area to float in between calls to your driver you can unlock it yourself before returning. If you do this make sure that you do not rely on the address of any data item staying the same between calls; also, make sure that the data area does not contain any objects, such as windows, that the Toolbox assumes will not move.

When a desk accessory is called to implement the `Open` entry, it should check to see whether the allocation of the data area succeeded. If it did not, the `dCtlStorage` field of the device control entry will be `0`. The desk accessory should issue some kind of error message (without using any of its globals!) and close itself.

Fields of the Driver's Header

A driver begins with a header containing several flags and other data items (some of which apply only to desk accessories). When the driver is opened, before the `Open` entry point is called, the Device Manager copies these fields to the device control entry. The device header is not used subsequently; the fields are used only to initialize the corresponding fields in the device control entry.

LightspeedC does not give you any way of pre-setting these fields. They are given reasonable default values, and often you will not feel the need to change them. If the default settings (listed below) are not to your liking, the recommended procedure is to change them on the fly by modifying the device control entry.

For example, suppose that you want your desk accessory to remain locked in between calls, and to be called every 60 ticks (once per second) to perform some periodic action. Simply execute the following code when the driver is called to implement the `Open` entry:

```
dce->dCtlFlags |= dNeedLock|dNeedTime;
dce->dCtlDelay = 60;
```

where `dce` is the second argument to `main`, the pointer to the device control entry.

These are the default settings for desk accessories:

`dCtlFlags`	
`dReadEnable`	0
`dWritEnable`	0
`dCtlEnable`	1
`dStatEnable`	0
`dNeedGoodBye`	0
`dNeedTime`	0
`dNeedLock`	0
`dCtlDelay`	n/a (`dNeedTime=0`)
`dCtlEMask`	`$016A` (mouse-down, key-down, auto-key, update, activate)
`dCtlMenu`	0

These are the default settings for device drivers:

`dCtlFlags`	
`dReadEnable`	1
`dWritEnable`	1
`dCtlEnable`	1
`dStatEnable`	1
`dNeedGoodBye`	0
`dNeedTime`	0
`dNeedLock`	1
`dCtlDelay`	n/a (`dNeedTime=0`)
`dCtlEMask`	n/a (desk accessories only)
`dCtlMenu`	n/a (desk accessories only)

Code Resources

Introduction

LightspeedC can be used to write "pure" code resources. Such resources do not have the complex structure of drivers; they simply contain code to be called at its entry point, `main`. How `main` is called depends on the sort of code resource involved. *Inside Macintosh* documents a variety of resource types, such as `'FKEY'`, `'INIT'`, `'PACK'`, `'WDEF'`, `'MDEF'`, and `'CDEF'`, that are called by the Macintosh Toolbox on designated occasions.

For example, an 'FKEY' resource is called in response to a command-shift-digit key combination; it is called by the Event Manager with no arguments. An FKEY handler in LightspeedC would take the form:

```
main()
{
    ...
}
```

As another example, a 'WDEF' resource is used to provide a custom window definition; it is called by the Window Manager with several arguments (see the "Window Manager" chapter in *Inside Macintosh* for details). A window definition procedure in LightspeedC would take the form:

```
pascal long main(varCode, theWindow, message, param)
int varCode, message;
WindowPtr theWindow;
long param;
{
    ...
}
```

You may define your own code resource types. You might do so, for instance, in order to make a function you've written in LightspeedC available to another program not written in LightspeedC. The "client" program need only load the resource and call it at its beginning. Either C or Pascal calling conventions may be used.

As with drivers, a short assembly-language stub (placed in your code resource automatically by LightspeedC) receives control before `main`. All this stub does is place the address of the resource where you can find it (see "Locking Code Resources", below) and jump to `main`.

Code resources may not have any global (or static) data. You cannot use the MacTraps library because the Quickdraw globals are defined there. However, since the MacTraps library is actually a project, you can make a copy of MacTraps and remove the Quickdraw globals from the copy; the copy can then be used with code resources.

Address register A4 is not treated specially, but address register A5 is reserved as usual.

Packages

A group of related functions may reside in a single code resource. Since there is only a single entry point, `main` must be able to determine from its arguments which function is really being called. This is usually accomplished by pushing a small integer to the stack immediately prior to the call as a selector to indicate the desired function. `main` is written in assembly language and looks something like this:

```
MOVE.L  (SP)+,A1        ; return address
MOVE.W  (SP)+,D0        ; selector
MOVE.L  A1,-(SP)        ; restore return address
...
JMP     ...             ; dispatch to selected function
```

This technique allows each function to have a different number of arguments and a different return type. The client program implements each function with an assembly-language "glue" routine that simply pushes the selector and jumps to the resource; the function then appears to have been called directly from the client program.

This kind of code resource is called a *package*. Macintosh Packages, such as the Standard File Package, are packages, and you can write your own as well.

Locking Code Resources

Code resources should be locked while they are executing and unlocked at other times. The Macintosh Toolbox usually takes care of this for you when calling one of the standard types of code resources (`'PACK'` resources are a notable exception). When defining your own code resource types it is up to you whether the code resource itself or its caller should take the responsiblity for locking it on entry and unlocking it upon return.

If you choose to make a package responsible for locking and unlocking itself, you will find a pointer to the package in address register A0 when `main` is called. Locking the package is accomplished simply by:

```
_RecoverHandle
_HLock
```

Unlocking the handle on exit is a little more tricky, since you have to save the handle someplace while the function is executing. You can't save it on the stack, since the stack is being used to pass arguments to the selected function; and you can't save it in the global data area, since there isn't any. You must save it amidst your code at a location defined using the `DC` directive. For example:

```
handle:
        DC.L    0
main:
        _RecoverHandle
        _HLock
        LEA     handle,A1       ;  (can't do PC-relative store)
        MOVE.L A0,(A1)
        ...
common_exit:                    ;  all functions exit here
        MOVE.L handle,A0
        _HUnlock
        ...
```

If the function `main` is written in C, the pointer to the code resource may be found in the low-memory location `ToolScratch ($9CE)`. Your code might look like this:

```
main()
{
    Handle h = RecoverHandle(* (Ptr *) 0x9CE);

    HLock(h);
    ...
    HUnlock(h);
}
```

A final point: If your code resource can be called reentrantly, it should not unconditionally be unlocked each time it returns. Instead it should be restored to the same state of locked-ness it had on entry.

The Components of a LightspeedC Program

Each source file or library in a project has four object components, known as CODE, DATA, STRS, and JUMP. The CODE component contains all the code generated. The DATA component contains all the global and static variables. String literals and floating-point constants are stored in the STRS component. Finally, the JUMP component contains the jump table.

The sizes in bytes of each of these components may be displayed by the **Get Info** menu command. Totals are also displayed per segment and for the entire project. You will notice that for some components there is some per-segment and/or per-project overhead. The CODE total must be less than 32K per segment, and the DATA and JUMP totals must each be less than 32K for the entire project. If any of these limits are exceeded, the linker will report failure. Note that there is no limit on the size of the STRS component.

For drivers, the DATA and STRS components combined must be less than 32K bytes in size. For code resources, the CODE and STRS components combined must be less than 32K in size, and the DATA component must be empty.

An 8-byte jump table entry is generated for each function that is not declared static as well as for each function that is used in a non-call context (i.e. whose address is taken). No other function gets a jump table entry, since none can be called from outside its file. The jump table built by the **Build Application...** command may be smaller; the entry for each function that is only referenced in intra-segment calls is removed.

When the **Build Application...**, **Build Desk Accessory...**, **Build Device Driver...**, or **Build Code Resource...** command is issued, the project is searched for source files and libraries that are not reachable on a reference chain starting from `main`. Unreachable files may be removed; they are not actually removed from the project, but the object components they contribute will not appear in the application, driver, or code resource being built. The file in which `main` is defined cannot be removed, nor can any file which that file references, nor can any file which any of those files reference, and so on.

If a library is itself a project, the source files and libraries which make it up are individually eligible for removal. Libraries converted from *.Rel* files or built with **Build Library...** are considered atomic units and are either removed or retained in their entirety.

The standard libraries provided with LightspeedC, such as MacTraps and the Standard I/O Library, are distributed as projects so that your application need contain only those portions which it actually uses.

10
Achieving Lightspeed

Introduction

It is sometimes said—not entirely tongue-in-cheek—that the best feature of the C programming language is its portability while its worst feature is its lack of portability!

Since C is already so capable, it is rarely extended in any significant way. It is implemented in a relatively consistent manner on a great variety of computers; hence its reputation for portability. On the other hand, there is no universally accepted language standard, so there are often a number of minor, but annoying, differences among various C implementations that can complicate the task of porting code to another system.

An interesting dilemma confronts a C language implementor. He can try to maximize portability to his compiler by incorporating features and variations found in other systems, or he can try to maximize portability from his compiler by implementing only the lowest common denominator. To make matters worse, these conflicting goals must be balanced against his desire to produce the best possible compiler for users who do not have portability as a primary concern.

We have attempted to include in LightspeedC all sufficiently common language extensions, but to avoid variation from the usual behavior of standard features. As C compilers go, LightspeedC is a very standard one, and you should find a high degree of portability between it and other compilers which also adhere closely to the standard. Unfortunately, "the standard" is merely a de facto one—the answer to "what is standard?" is often simply "what most compilers do". This requires a judgment call, and where our judgment differs from that of other compiler implementors you may encounter portability problems.

This chapter is designed to aid you in your efforts to port your existing code to LightspeedC. As you grumble over this task, console yourself with the knowledge that because LightspeedC is so standard you are likely to be improving the portability of your code—your next porting effort should be much easier!

The list of portability issues presented here is not comprehensive. However, whenever appropriate specific mention is made of other compilers presently available for the Macintosh, specifically Aztec C (from Manx Software), Mac C (from Consulair), and Megamax C (from Megamax). A section is devoted to UNIX compatibility issues; comments made there may also be applicable to compilers designed for other non-Macintosh environments such as the IBM PC.

Sizes of Numbers

The single most common way in which C compilers differ is in the sizes of the three integer types, `short`, `int`, and `long`. There seems to be general—though by no means universal—consensus that:

1. `short` should be the shortest integer type bigger than a `char` that is supported by the hardware;

2. `long` should be the longest integer type supported by the hardware; and

3. `int` should be the "most natural" integer type supported by the hardware.

On the 68000 this means that `short` should be 2 bytes and `long` should be 4 bytes, and most compilers do it this way (Megamax C is a notable exception). The correct size for plain `int` is more problematical. The 68000 has 32-bit registers, so some compilers (including Mac C) implement 4-byte `int`s. But the 68000 has a 16-bit data bus and a 16-bit ALU, so 16-bit operations are considerably more efficient than 32-bit operations. Furthermore, Macintosh applications are often pressed for memory, and 2-byte `int`s use a lot less space than 4-byte `int`s, so some compilers (including Aztec C) implement 2-byte `int`s. LightspeedC implements 2-byte `int`s.

If you are porting from a compiler which has different integer sizes, and you have code which relies on those sizes, you may have some conversion to do. The most common case is code that assumes that `int` and `long` are the same size. Here is an example:

```
foo()
{
     long a, b, result;

     result = baz(a, b);
}

baz(i, j)
{
     return(i + j);
}
```

This code works fine when `int` and `long` are the same size, but it is not portable. It will not work in LightspeedC (or in many other compilers). Either `i` and `j` should be declared long, or `a` and `b` should be cast to `int` before being passed to `baz`.

It is less common for a program to make specific assumptions about the sizes of floating-point numbers, but you should be aware that these also tend to differ between compilers. For maximum accuracy and efficiency, LightspeedC uses the SANE extended-precision type to implement the C type `double`. Other compilers for the Macintosh either don't provide the extended-precision type at all (e.g. Aztec C), or add a new largest type which takes over the role of `double`, changing the C semantics (e.g. Mac C).

Passing a struct as an Argument

In the original *K&R* definition of C, `structs` (and `unions`) could not be passed as arguments to functions. Some compilers, rather than flagging an attempt to do so as an error, implicitly insert an "address-of" (`&`) operator, with the result that a pointer to the `struct` is passed. In effect, the `struct` is passed "by reference"; the called function can modify the `struct` by accessing it indirectly through the pointer. Aztec C, for instance, does this.

Most modern C compilers, including LightspeedC, allow structs to be passed "by value", that is, by copying them onto the stack. The called function can modify its copy but not the original. Here is an example of the kind of problem you can run into when porting code written for an older compiler:

```
typedef struct {
    long a,b,c;
} aStruct;

foo()
{
    aStruct baz;
    ...
    sub1(baz);
}

sub1(p)
aStruct *p;
{
    p->a = 666;
}
```

When `foo` calls `sub1`, the `struct baz` itself—not its address—is passed. `sub1` expects to see a pointer to a `struct` but instead sees `baz.a`. If you are lucky this will immediately lead to an addressing exception; if not, the damage may be more subtle.

Passing a Point as an Argument

A related problem pertains to passing a Quickdraw `Point` as an argument to a Macintosh Toolbox function. Some compilers, because they do not allow passing `structs` as arguments, require the address of the `Point` to be passed instead. Mac C and Megamax C both require, for example,

```
result = PtInRgn(&aPoint, aRgnHandle);
```

even though `PtInRgn` expects the actual `Point` as its first argument. (The address is dereferenced before `PtInRgn` is actually called; otherwise `PtInRgn` would not work!)

In LightspeedC, the above statement would cause `aPoint`'s address to be passed to `PtInRgn`; the correct call is:

```
result = PtInRgn(aPoint, aRgnHandle);
```

Some Toolbox functions expect a `Point` to be passed as a `VAR` parameter; in such cases an address must be passed. For example,

```
SetPt(&aPoint, horizontal, vertical);
```

would be correct in LightspeedC as well as in other compilers.

Identifier Length and Capitalization

One area in which there is a great deal of variation among C compilers is that of maximum identifier length. Many compilers limit the length of identifiers, and ignore characters beyond the maximum. LightspeedC does not, even for identifiers which are visible externally. For example:

```
foo()
{
    int a_very_long_name;

    a_very_long_nme = 1;  /* misspelled */
}
```

Many compilers accept this, because the difference between the two identifiers occurs beyond the spelling "horizon". LightspeedC reports the misspelling as an undeclared identifier. Similarly:

```
/*  file 1  */
int a_very_long_external_name;

---------------

/*  file 2  */
extern int a_very_long_external_nme;  /* misspelled */
```

Many compilers would not notice the misspelling, but LightspeedC reports it as an undefined symbol at link-time.

Compilers which use the MDS linker, such as Mac C, are case-insensitive when it comes to external identifiers. For example:

```
/*  file 1  */
int theFooBar;

---------------

/*  file 2  */
extern int theFoobar;
```

Mac C accepts these as being the same symbol; LightspeedC does not.

In these matters LightspeedC is stricter than other compilers. It will catch spelling and capitalization mistakes that others will not.

Runtime Environment

As you know if you have programmed the Macintosh before, each program must begin with a sequence of initialization calls to various portions of the Macintosh Toolbox. With some compilers, part of this job is done for you before `main` is ever called. For example, Mac C executes:

```
InitGraf(&thePort);
InitFonts();
InitWindows();
```

You will need to insert these lines when porting to LightspeedC. We feel it is more consistent with the C philosophy to leave control up to you. It may be hard for us to imagine a program which wouldn't begin with these calls, but we'd rather not take that decision out of your hands.

(Non-Macintosh—e.g. UNIX-style—programs ported to LightspeedC do not need to make these calls. The Standard I/O Library will take care of things for you.)

LightspeedC takes a similar stand with regard to Megamax C's handling of string arguments to Macintosh Toolbox calls. The Toolbox expects Pascal-style strings which begin with a length byte, not null-terminated C-style strings. Megamax converts string arguments from C-style to Pascal-style on input to Toolbox calls, and converts them back on output. This can be convenient, but the overhead is considerable. Besides, maybe you would find it easier to keep your strings in the Pascal format all the time. We've decided to leave the decision up to you; string constants may be defined Pascal-style using the `"\P..."` convention, and the library functions `CtoPstr` and `PtoCstr` are available to convert strings in place if you like.

The spelling and capitalization of Macintosh Toolbox and Operating System calls differs somewhat between compilers. Megamax C, for example, recognizes lower-case-only names for these calls. LightspeedC conforms exactly to the names given in *Inside Macintosh.*

Use of Assembly Language

Obviously, use of inline assembly does not lead to portable code. LightspeedC does not implement inline assembly, and you will have to remove it from your code when porting from Mac C or Megamax C.

One of the most common uses for inline assembly is to write "glue" routines that mediate between the different calling conventions used in C and Pascal. In LightspeedC you do not need to use inline assembly to achieve this, because you can use the `pascal` keyword. (Of course, your code will still not be portable, but at least LightspeedC does the dirty work.)

When incorporating assembly language routines in a C program, you obviously have to conform to the calling conventions used by your C compiler. There is in general a large degree of conformity between C compilers in this area, but once again this agreement is not universal. Mac C, for instance, implements an unusual set of calling conventions, so your assembly language code will be affected when porting. For more information on LightspeedC's calling conventions, refer to Chapter 9, "Running at Lightspeed".

Compiler Issues

Occasionally you may run into portability issues due to variations in the C language implemented by various compilers. Here are a few we know about.

Mac C is less strict than most other compilers in the syntax it allows. For instance, it allows you to omit a statement-terminating semicolon immediately before a right brace. LightspeedC will report a syntax error.

Consider this program fragment:

```
int input, output;

input = 0;
output = !input;
```

Using Mac C, the value of `output` is now `-1`, but of course it should be `+1`. This will not cause you any problems unless you have code that relies on the non-standard value for canonical true. (Note that it is considered legitimate portable C programming practice—though it may be questionable style!—to rely on canonical true being `+1`.)

Some compilers, including Aztec C, allow initialization of auto aggregates (`structs`, `unions`, and arrays). For example:

```
f()
{
     int x[] = { 2, 3, 5, 7 };

     ...
}
```

This is not implemented in LightspeedC. You could declare the array `static`, use explicit assignment statements, or copy the contents of a duplicate `static` array into `x` each time `f` is called.

Converting from UNIX

The primary issue involved in porting code from a UNIX or UNIX-like environment to LightspeedC is library support for standard UNIX functions. LightspeedC comes with a robust set of UNIX compatibility libraries that provide many of the functions found on the most popular versions of UNIX. Some standard UNIX functions that would have no meaning on the Macintosh have been omitted, and UNIX systems themselves vary in the libraries they support, so you may encounter occasional problems. We've tried to keep them to a minimum. Detailed descriptions of the nearly 200 functions provided in LightspeedC may be found in Chapter 13, "Standard C Libraries".

A useful UNIX feature not available in the standard Macintosh environment, the command line, is supported by LightspeedC's libraries. To use this feature, name your main function `_main` instead of `main`, and add the source file *unix main.c* to your project. This file contains an

implementation of `main` which prompts for a command line when your program begins execution and calls `_main` with UNIX-style `argc` and `argv` parameters. Redirection of the standard input, standard output, and standard error channels is supported using the following conventions:

<	redirect `stdin` to the file name that follows
>	redirect `stdout` to the file name that follows
>>	redirect `stderr` to the file name that follows

Some UNIX implementations use >> to cause the standard output to be appended to the file name that follows. You can easily modify *unix main.c* so that it behaves this way instead. Just change:

```
...
else if (*cp == '>') {
    mode = "w";
    filename = true;
    if (*++cp == '>') {
        file = stderr;
        cp++;
    }
    else
        file = stdout;
}
...
```

to:

```
...
else if (*cp == '>') {
    mode = "w";
    filename = true;
    if (*++cp == '>') {
        mode = "w+";
        cp++;
    }
    file = stdout;
}
...
```

11
Epilogue

A Warp Drive for C Developers

There are probably very few serious system software developers who have not dreamed of creating the ultimate programming environment. At THINK, where such developers are plentiful, these dreams are never in short supply.

Our first attempt to realize one of these dreams was Macintosh Pascal, an interactive programming environment for ANSI Pascal based on a new technology for high level language interpreters developed by one of our founders, Mel Conway. The interactive interpreter made it possible for us to create the illusion of a pure Pascal environment. Source level statement tracing and breakpointing, dynamic observation of variables and expressions, and spontaneous source statement execution in scoped context were all possible during program execution. The integration of program editor, fast compiler/interpreter combination and source level debugging tools into a single relatively modeless programming environment made for an easy-to-use programming tool with fast response.

Because of anticipated limitations on program size and execution speed imposed by memory and the interpreter, we aimed the product at educational and casual users for whom the ease of use and source level debugging would be especially valuable. (Apple desired exactly such an educational product to meet the needs of its University market, believing that serious programming for the Macintosh would always be done on a cross-development system like the Lisa.)

Our own developers soon discovered that the system was wonderful to use until it ran out of steam. Informal observations of programmer productivity suggested that programmers could develop the same program significantly faster when they used Macintosh Pascal than when they used the Lisa Pascal cross-development tools. Not surprisingly, the speed ratio seemed roughly proportional to the time it took to "turn around" a new version of the software under development using each system. The integrated environment and fast compiler (and no linking step) gave Macintosh Pascal a tremendous speed advantage over the Lisa tools. Unfortunately, its limitations made it unsuitable for serious software development. *But people got a taste of speed and convenience and a flock of new dreams was born.* And along the way, a few of us got interested in C.

C seemed to be gaining in popularity with microcomputer software developers. We knew of several conventional C compilers under development for the Mac, but we imagined something a little bit different. Our dream was to build a development system that could turn around a new version of a program so fast that it would seem to be driven by a Star Trek-like warp drive when compared with other systems.

With the arrival of Michael Kahl at THINK these dreams started to head toward reality. Kahl had a few dreams of his own, and more importantly, had the depth of familiarity with C needed to create

a solid and credible product. He felt that no matter how fast our development environment could perform, it would never be acceptable unless it could also generate very high quality native code. Although Kahl was intrigued with MacPascal's source level debugging, he, too, felt that its quick response was the most important adjunct to development. This seemed to be especially true for C programmers who like to see their machine level code exactly as it is so that they can optimize their use of the language. He also envisioned having a central management tool, *the project,* that would make it easy to control everything from segmentation to linking.

Naturally, the language had to be full K&R with modern extensions like enumeration specifiers and structure assignment, IEEE numerics support, full Macintosh Toolbox and OS support, and a serious UNIX compatibility library.

One of the toughest aspects of the project was getting high quality code generated at high speeds. To define a conservative measurement of speed we aimed at lines that were fairly dense on average. Of course, any compiler's speed varies as a function of expression density, which is why compile speed benchmarks can be so variable and misleading. During development our conservative measurement consistently clocked in at approximately 250 lines/second. As a consequence, the finished compiler achieves between 250 and 500 lines/second for typical programs depending on their actual expression density.

In the spirit of C we chose to build a compiler that would generate superior code but leave register optimization to the developer through the use of register variables. Our quality goal was simply that our compiler would almost always generate code that was as good as or better in terms of space and speed than the best compilers of this type on the Mac. The detailed benchmarks supplied in Appendix F address this issue more eloquently than any comments we might make here.

Because the compiler was so fast, and because non-cascading error recovery is so difficult (and rarely achieved in practice), we chose to indicate compile errors by opening the erroneous file in the text editor and putting the user on the offending line.

Once Kahl got fast, high-quality native code compilation working, we soon discovered that link time was now the dominant component in turnaround time. No longer! Probably the single most extraordinary aspect of LightspeedC is that the linker is ultra-fast even for large projects with many files. Because the linker is so fast as to be nearly invisible, we jokingly call it "the missing link". LightspeedC, for example, consists of more than 80 files and nearly a megabyte of source code, but it can link itself in less than one second. For that matter, LightspeedC can compile and link itself from scratch on a RAM disk in a little over two minutes. If you find this hard to believe, take a look at the detailed benchmarks, or try running a few of your own.

Having achieved this exceptional program processing speed, Kahl observed that it might be essential to provide an automatic build facility for users to be able to get the most out of it. There was already the *project* concept to describe the structure of an application, and it seemed obvious to extend this to include an intelligent and highly automated "make" facility, so we did.

The only remaining question was how much more to integrate into the environment. A text editor was built-in from the start, but it was based on the Mac's internal text editing utility, TE, which limited file size to 32K and seriously limited touch typing and screen scrolling performance. So Jon Hueras wrote a program editing utility, PE (using LightspeedC, of course), and Kahl incorporated it into LightspeedC's editor with results you can experience for yourself.

Because we expected that most users would want to create complete applications, and the Mac does not multi-task very cleanly, we decided to clean the slate with a fresh launch to start execution of a

built program. A quick launch and return facility makes it easy to turn around versions under test without having to create a standalone version. By keeping non-code resources in a separate resource file the user can avoid performing a separate RMaker or Resource Editor step at each development turn. When the time comes to build a full standalone version, a simple RMaker process consolidates the two resource files in a single step. For debugging, we assumed the user would use Macsbug, and provided an option to include Macsbug symbols in the object code.

And the final result? A very serious C development environment on the Mac that makes it as fast and easy to work on very large programs as it is to work on small ones. LightspeedC makes it possible to quickly edit a few files of a large multi-file project, and at the touch of a single key, to automatically recompile only as needed, and to relink, build, and launch the project for testing at least 10 times faster than any other Mac-based C development system now on the market.

In other words, a Warp Drive for C Developers.

PART THREE

LANGUAGE REFERENCE MANUAL

12
C Language Reference

This section describes the C language implemented by LightspeedC on the Macintosh. It is meant to be used in conjunction with Appendix A, "C Reference Manual", of Kernighan and Ritchie's *The C Programming Language*. The sections are named and numbered exactly as in that work, and only those aspects of LightspeedC which differ from *K&R* or are potentially ambiguous in *K&R* are described here.

2.2 Identifiers (Names)

K&R specifies that no more than the first eight characters of identifiers are significant, although more can be used. In LightspeedC, there is no limit to the number of characters which are significant in an identifier.

In addition, *K&R* specifies that external identifiers, which are used by various assemblers and loaders, may have additional restrictions. In LightspeedC, no additional restrictions apply to external identifiers.

2.3 Keywords

The following identifiers are reserved in LightspeedC:

```
auto        extern      short
break       float       sizeof
case        for         static
char        goto        struct
continue    if          switch
default     int         typedef
do          long        union
double      pascal      unsigned
else        register    void
enum        return      while
```

The `entry` keyword reserved by *K&R* for future use is not reserved in LightspeedC. In addition, *K&R* mentions that the `fortran` and `asm` keywords are reserved by some implementations of C; they are not reserved in LightspeedC.

The keywords `enum`, `pascal`, and `void` were not reserved in *K&R*. See §§ 4 (`void`), 8.1 (`pascal`) and 8.2 (`enum`) for additional details.

2.4.1 Integer constants

K&R allows the digits 8 and 9 to represent the values 10 and 11 in octal constants. LightspeedC does not allow the digits 8 and 9 in octal constants.

Decimal constants are of type `int`, or `long int` if necessary; hex and octal constants are of type `unsigned int`, or `unsigned long int` if necessary.

A leading '-' is a unary operator, not part of the constant.

2.4.3 Character constants

K&R specifies that certain non-printing characters can be specified using certain escape sequences. The following character escapes are implemented in LightspeedC:

`'\b'`	0x08 (backspace)
`'\f'`	0x0C (form feed)
`'\n'`	0x0A (line feed)
`'\r'`	0x0D (carriage return)
`'\t'`	0x09 (horizontal tab)
`'\v'`	0x0B (vertical tab)

As described in *K&R*, an escape sequence of the form `'\ddd'` where `d` is an octal digit, is also supported.

A single-character character constant has type `int`, and its value is always positive; e.g. `'\377'` is 255, not –1.

Multi-character character constants are allowed. The type of such a constant is `int` if the value is in range, `long int` otherwise. The characters are assigned left-to-right and right-justified. More than 4 characters are not allowed.

2.4.4 Floating constants

Floating constants have type `double`. Data types `float` and `short double` are also supported. See §2.6 for details.

2.5 Strings

In C, each string is terminated with a null byte (`'\0'`) so that programs that scan the string can find its end. This convention is supported in LightspeedC. However, because Macintosh Toolbox and Operating System calls were designed to be called from Pascal, which uses a different string representation, a second type of string constant has been included.

A string beginning `"\p"` or `"\P"` is a Pascal string. It is not terminated with a null byte; instead the first byte is a length byte indicating the number of characters following. Be careful; the length

byte may appear negative if it exceeds 127, and it will not be meaningful at all if the string is longer than 255 characters.

Pascal strings are required when calling certain Macintosh routines; however, if you like, they can be used in other cases as well.

All string constants are aligned on word boundaries.

2.6 Hardware characteristics

Data types have the following hardware characteristics in LightspeedC:

`char`	8 bits
`short int (or short)`	16 bits
`int`	16 bits
`long int (or long)`	32 bits
`float`	32 bits
`short double`	64 bits
`double`	80 bits

Floating-point is IEEE standard, courtesy of Apple's Standard Apple Numeric Environment (SANE) numerics package. The range of `double` values is $\pm 10^{\pm 4932}$. `float` corresponds to SANE `SINGLE`, `short double` corresponds to SANE `DOUBLE`, and `double` corresponds to SANE `EXTENDED`.

4. What's in a name?

Integers of all sizes, including `char`, may be declared `unsigned`. "Plain" `char` is a signed quantity and suffers sign-extension.

Enumerated types are implemented; see §8.2. An enumerated type is not actually a new type, but a synonym for the appropriate size integral type.

A `void` type is available. (This is a recent addition to C, since *K&R*. See Harbison & Steele, section 5.10, for additional details.)

The type "function returning `void`" represents a procedure that does not return a value.

The type "pointer to `void`" (`void *`) is an anonymous pointer type which may be freely converted to any other pointer type without need of a cast.

Finally, an expression may be cast to `void` to indicate that it is being evaluated only for its side effects; its value is discarded.

The `void` type has no other uses; no `void` objects may be declared, and no `void` values may be used. The compiler prevents `void` functions from returning values.

6.1 Characters and integers

K&R points out that a character may be used wherever an integer may be used, but that in all cases, the value is converted to an integer, and the conversion results in sign extension.

In LightspeedC, variables of type `char` are signed and do suffer sign-extension; however, single-character character constants are of type `int` and by contrast with *K&R* always have positive values. For example, the value of `'\377'` is 255, not –1.

6.2 Float and double

All floating-point arithmetic is carried out in extended 80-bit precision. Whenever a `float` or `short double` appears in an expression, it is extended to `double` which corresponds to SANE type `EXTENDED`. Conversion is performed according to Apple's SANE numerics package. `float` corresponds to the SANE type `SINGLE`. `short double` corresponds to the SANE type `DOUBLE`.

6.5 Unsigned

In any conversion, sign-extension is performed according to the old type, not the new type. Thus, when an unsigned integer is converted to a longer signed integer, it is not sign-extended; when a signed integer is converted to a longer unsigned integer, it is sign-extended. (This feature is exactly as defined in *K&R*.)

6.6 Arithmetic conversions

This section describes the "usual arithmetic conversions" that occur when various operators are used. These conversions may be applied to a single operand (as in a unary operator), or jointly to a pair of operands (as in a binary operator).

First, all operands of certain types are converted to a larger type as follows:

type	converted to
`char`	`int`
`unsigned char`	`unsigned int`
`float`	`double`
`short double`	`double`

Then, if both operands have the same type (or if there is only one operand), that is the type of the result. Otherwise, both operands are converted to a common type, and that is the type of the result. The common type is whichever of the two types appears first in the following list:

```
double
unsigned long int
long int
unsigned int
int
```

7.1 Primary expressions

The type of a constant may be `int`, `long int`, `unsigned int`, `unsigned long int`, or `double` depending on its form.

In a function call, actual arguments of integral type are extended to the size of an `int` if necessary, and floating-point values are converted to `double`. Arguments of `struct` or `union` type are allowed, and are passed by value.

If an undeclared identifier is used as the name of the function to be called, it is contextually declared to be a function returning an `int`, exactly as if the declaration `extern int` *identifier* `();` had appeared. Exception: if the identifier names a Macintosh Toolbox or OS call, the definition of that call is entered, somewhat as if the declaration `extern pascal` *type identifier* `();` had appeared, where *type* may be `void`, `char`, `int`, or `long`. However, more information is actually available than given by such a declaration.

If an identifier which was declared `pascal` is used as the name of the function to be called, Pascal calling conventions are used instead of C calling conventions. Integral arguments are not extended to the size of an `int`, and no argument may exceed 4 bytes in size. (See Chapter 9 for additional details.)

Macintosh Toolbox and OS calls are handled similarly to `pascal` functions. In addition, each actual argument is extended or truncated depending on the size of argument expected by the call. If an argument of non-integral type must be resized, or if the wrong number of actual arguments is supplied, the compiler signals an error. (Note that the compiler does not know the expected types of the arguments, only their sizes, so be careful.)

The rules given in *K&R* for `struct` and `union` references are strictly enforced by LightspeedC. The identifier in a `.` or `->` expression must be that of a member of the appropriate `struct` or `union`.

7.2 Unary operators

The `&` operator may be applied to an array, producing a value of type "pointer to array of ...". Without the `&` operator, the array would have been converted to type "pointer to ..." which is quite different, so be careful.

The `&` operator may be applied to a function, but with no effect since the function would have been converted to type "pointer to function returning ..." anyway.

7.3 Multiplicative operators

In a division or remainder operation, an unsigned division is performed if the result is to be an `unsigned` integer of any size. In a signed division, a non-zero remainder has the sign of the dividend.

7.4 Additive operators

The operands must not have type "pointer to `void`". For example:

```
void *generic_pointer;    /*This is legal*/
   generic_pointer ++;    /*This is illegal*/
```

The difference of two pointers has type `long int`.

7.5 Shift operators

The usual arithmetic conversions are performed on the left operand alone, as though a unary operator were being applied. An arithmetic right shift is performed if the left operand is signed.

7.6 Relational operators

A pointer may only be compared to a pointer of the same type or a pointer to `void`.

7.7 Equality operators

A pointer may only be compared to a pointer of the same type, a pointer to `void`, or a constant zero.

Whenever a value is tested to see if it is non-zero, as in the conditional operator and in the `if`, `while`, `do`, and `for` statements, the test is equivalent to `(expression) != 0`. All conversions and type restrictions apply accordingly.

7.13 Conditional operator

In addition to the possibilities given in *K&R*, one of the second and third operands may be a pointer and the other a pointer to `void`; the type of the result is the non-anonymous pointer type.

7.14 Assignment operators

Values of `struct` and `union` type may be assigned; the left and right side must be of the same type. This applies only to simple assignment.

A pointer may be assigned the value of a pointer of the same type, a pointer to `void`, or a constant zero. A pointer to `void` may be assigned the value of a pointer of any type, or a constant zero. Anything else is a compiler error. Use casts to bypass these typing rules.

8.1 Storage class specifiers

LightspeedC implements the additional storage class `pascal`. An identifier declared `pascal` must have type "function returning ...". A `pascal` function is called using Pascal calling conventions; see §7.1. If the `pascal` keyword appears in a function definition, the function will expect to be called using Pascal calling conventions. The `pascal` keyword may appear in conjunction with `extern` and `static`.

A variable of any integral type may be declared `register`; it will be placed in a data register (if possible). A variable of any pointer type may be declared `register`; it will be placed in an address register (if possible). As many as three data registers (`D7, D6, D5`) and two address registers (`A4, A3`) may be active at once; registers allocated to `register` variables become available again at the end of the block in which they are assigned. Excess register declarations are ignored.

8.2 Type specifiers

Additional type specifiers are enumeration specifiers and `void`. The `unsigned` modifier may be applied to `char` and `int` (even in the presence of `short` or `long`). Finally, `short double` refers to the SANE double-precision floating-point type (`long float` is accepted as equivalent to `double`).

Enumeration specifiers have the following syntax:

enumeration-specifier:
enum *identifier*
enum *identifier*$_{opt}$ { *enum-list* }

enum-list:
enum-item
enum-item , *enum-list*

enum-item:
identifier
identifier = *constant-expression*

The tag identifier works just as with `structs` and `unions` to identify the enumeration type.

Each identifier appearing in the *enum-list* is defined as an enumeration constant whose type is the enumeration type. If not explicitly specified by "= *constant-expression*", the value of the constant is one greater than that of the constant preceding it in the *enum-list*, or 0 if it is the first one. Constant expressions appearing in `enum` declarations must have integral type.

An enumeration type is not really a distinct type, but equivalent to `int`, or to `char` if all of its declared constants have values in the range –128 to 127.

8.4 Meaning of declarators

According to *K&R*, "functions may not return arrays, structures, unions or functions, although they can return pointers to such things." In LightspeedC, functions may return structures and unions.

Caution: If a function returns a `struct` or `union`, make sure that declaration of the function is present whenever the function is used, even if the return value is ignored. Otherwise, the consequences may be dire. (See Chapter 9 for details.)

8.5 Structure and union declarations

Members of all types other than `char`, `unsigned char`, and singly and multiply dimensioned arrays of `char` or `unsigned char` are aligned on even addressing boundaries. Similarly, `structs` and `unions` are padded to an even size.

It is legal to specify an array without size as the last member in a `struct`. The array does not contribute to the size of the structure. For example:

```
struct {
    unsigned int    count;
    char            data[];
} CountData; /*  size of CountData is 2  */
```

Bitfields (or fields, as they are called in *K&R*) may be declared to be of any integral type. The size of the declared type determines the word size for that bitfield; thus a word may be 8, 16, or 32 bits in length. Keep this definition of "word" in mind throughout the following discussion.

A sequence of bitfields with the same word size are packed into a word, but a bitfield is placed in the next word if it would otherwise straddle a word boundary. No bitfield may be wider than a word. Fields are assigned beginning with the high-order bit of a word. An unnamed field with a width of 0 "closes out" the current word. A bitfield with a different word size from the preceding bitfield causes this to happen automatically (just as a non-bitfield member does).

The high-order bit of a bitfield is not treated as a sign bit, even if the declared type of the bitfield is signed. Think of it this way: the "real" type of the bitfield is "`unsigned` *n* bits", which gets converted to the declared type whenever the bitfield appears in an expression.

Names of members need only be distinct from each other within a single `struct` or `union` declaration. Each such declaration introduces a unique name space for its members; see §11.1.

8.6 Initialization

Unions may be initialized; the initialization applies to the first member of the union. Structures containing bitfields may be initialized.

9.7 Switch statement

The `switch` expression and `case` constants may be of any integral type, after the usual arithmetic conversions are applied.

9.10 Return statement

A function declared "function returning `void`" may not return a value.

10.1 External function definitions

The `pascal` specifier is allowed in addition to `extern` or `static`; see §8.1.

Formal parameters declared `float` or `short double` have their declaration adjusted to read `double`. Formal parameters may have `struct` or `union` type.

11.1 Lexical scope

LightspeedC follows *K&R* in giving file scope to identifiers declared `extern`, whether explicitly or implicitly. (Many C implementations treat `extern` variables as local to a procedure.)

There are several distinct name spaces. Names in the same space must not conflict with each other, but may be the same as names in other spaces. For example, the same identifier may be used without conflict for a statement label and an enumeration constant. However, macro substitution is performed by the preprocessor without regard for name spaces.

Statement labels form a name space.

`struct`, `union`, and `enum` tags form a name space.

Each `struct` or `union` defines a unique name space for its members.

Variables, functions, `typedef` names, and enumeration constants form a name space.

11.2 Scope of externals

A function which is declared `static` when its definition is given is not exported to other files, even if it was previously declared `extern` (explicitly or implicitly). This allows forward references to private functions.

12. Compiler control lines

Preprocessor lines may begin with any number of spaces or tabs (but not comments). Lines containing only a # (preceded by any number of spaces or tabs) are ignored.

12.3 Conditional compilation

An identifier may optionally appear following `#else` or `#endif`. This identifier must match the corresponding `#ifdef` or `#ifndef`.

The command `#elif` is allowed; it is like `#else` followed by `#if`, but no additional matching `#endif` is required.

Identifiers appearing in an `#if` (or `#elif`) expression evaluate to 0 if they are not macro names.

12.4 Line control

The `#line` command is accepted but ignored in the LightspeedC environment.

13. Implicit declarations

An undeclared identifier appearing in a function-call context is implicitly declared to be of type "function returning `int`", unless it is recognized as the name of a Macintosh Toolbox or OS call; see §7.1.

14.1 Structures and unions

Structures and unions can be assigned, passed as parameters, and returned from functions.

The identifier in a `.` or `->` expression must be that of a member of the appropriate `struct` or `union`.

14.4 Explicit pointer conversions

A `long int` (or `unsigned long int`) is required to hold a pointer without loss of information. Pointers are byte addresses. `chars` (and `unsigned chars`) have no alignment requirements; everything else must have an even address.

15. Constant expressions

The `!` operator may be used in constant expressions; its omission is clearly just an oversight on the part of *K&R*.

16. Portability considerations

In addition to the portability considerations specified in *K&R*, the use of the `pascal` type, of Pascal string conventions, or of the type `short double` will result in code that is not portable to most other C environments.

17. Anachronisms

Obsolete constructions are not supported.

18. Syntax Summary

The constructions described in the next two sections have been added to LightspeedC.

18.2 Declarations

`pascal` is an additional *sc-specifier*.

`void` and *enumeration-specifier* are additional *type-specifiers*.

enumeration-specifier:
`enum` *identifier*
`enum` *identifier*$_{opt}$ `{` *enum-list* `}`

enum-list:
enum-item
enum-item , *enum-list*

enum-item:
identifier
identifier = *constant-expression*

18.5 Preprocessor

`#else` *identifier*$_{opt}$

`#endif` *identifier*$_{opt}$

`#elif` *constant-expression*

13
Standard C Libraries

Introduction

LightspeedC supports all of the functions in the standard C libraries, as well as additional functions in a UNIX compatibility library.

These libraries are supplied both in source code and binary (project) form. As distributed in project form, the libraries are broken up as follows:

stdio	Miscellaneous standard I/O functions.
storage	Memory allocation functions.
strings	String handling functions.
math	Math functions.
unix	Miscellaneous functions provided for UNIX compatibility.
storageu.lib	UNIX memory allocation functions.
unix_strings	UNIX string handling functions.

In order to use the functions in these libraries, you need to include the appropriate header file in your program:

stdio.h	stdio
storage.h	storage, storageu
strings.h	strings, unix_strings
ctype.h	any of the `istype` functions in stdio
math.h	math
unix.h	unix

The appropriate `#include` statement is shown as a part of the syntax for each function. In addition, the library in which the function is contained is shown in parentheses in the heading of the page describing the function.

In some cases, functions are doubly defined--for example, in both `stdio.h` and `unix.h`. Including either of the header files will allow you to access one of these functions.

There are two purposes to the standard libraries. The first is convenience. These libraries contain many essential tools for manipulating files, I/O and reaching the operating system. The second is portability. A standard library separates the code from the underlying operating system. Unfortunately, there is not a universally accepted standard library for C, so

portability cannot be guaranteed. These libraries are a superset of the usual standard I/O library, so you should not have trouble porting code to this C compiler, but you may have to limit yourself to the basic subset when porting from this C compiler.

This section is divided into two parts, one for symbols that are defined as a part of the Standard I/O library, and another for functions in each of the libraries. Within each section, functions are in alphabetical order. In general, functions are one to a page, but where one or more functions are closely related, they may appear on the same page.

There are several cases in which this is not true. There are a large number of character type recognition functions (`isalpha()`, `isalnum()`, etc.), all of which are grouped together on a single page under the name `ctype` (for "character type"). In addition, all of the routines in the math library are listed on a single page, which is alphabetized using the heading `math`.

There is also a function index in the back of the book that will help you to quickly locate a function you need.

There are almost 200 functions and symbols in the standard libraries. However, there is a great deal of overlap between the functions. Consider the functions as a complete wrench set rather than as a few adjustable wrenches. There is a function designed for each specific job. For example, there is one function to get a character from `stdin`, two to get a character from the console, and two to get a character from a file. A complete set has the advantage of efficiency, but the disadvantage of bulk. It is harder to find the right tool at first. The tables below are designed to help you find the right function for the job.

Table 1
INPUT FUNCTIONS

Data Source	get a byte	get a string	formatted scan	get a block of data
stdin	getchar	gets	scanf	----
console	getch,getche	cgets	cscanf	----
memory	----	----	sscanf	----
file	fgetc,getc	fgets	fscanf	fread,read

Table 2
OUTPUT FUNCTIONS

Data Target	put a byte	put a string	formatted print	put a block of data
stdout	putchar	puts	printf,vprintf	----
console	putch	cputs	cprintf,vcprintf	----
memory	----	----	sprintf,vsprintf	----
file	fputc,putc	fputs	fprintf,vfprint	fwrite,write

Table 3
SELECTED FILE MANIPULATION FUNCTIONS

access file by	open file	close file	read from	write to	move place in file
FILE *	fopen	fclose	fread	fwrite	fseek
file number	open	close	read	write	lseek

Table 4
MEMORY ALLOCATION FUNCTIONS

----	zero block	do not zero block
regular block	calloc	malloc, getmem, sbrk
long block	clalloc	mlalloc, getml, lsbrk

Table 5
FREE ALLOCATED MEMORY

if allocated by	free by using
calloc	cfree, free
clalloc	free, cfree
malloc	cfree, free
mlalloc	free, cfree
getmem	rlsmem
getml	rlsml
sbrk	rbrk
lsbrk	rbrk

Defined Symbols

The following symbols are defined by including `stdio.h`:

`BUFSIZ`

`BUFSIZ`, the size of an input/output buffer, is defined as 512. Buffers are used in calls to functions like `read()` and `write()`.

Buffers less than `BUFSIZ` are less efficient, buffers larger than `BUFSIZ` are treated as sets of `BUFSIZ` blocks. `BUFSIZ` is measured in characters (or bytes), so a buffer will hold proportionately fewer items of a larger data type.

`BUFSIZE` is an alternate symbol with the same definition.

`_console`

`_console` contains the file pointers for I/O to the console. The console is the keyboard and the screen. `stdin`, `stdout`, and `stderr` initially point to the console as well, but can be redirected to point to a file.

A family of functions beginning with the letter c (`cgets()`, `cprintf()`, etc.) are dedicated to performing I/O to the console.

`EOF`

`EOF`, the value used to represent the end of a file, is defined as `-1`. Write code using `EOF` rather than `-1`. This will allow code to be ported to a system which does not use `-1` to represent the end of a file (some systems use `0` to represent `EOF`). `EOF` is returned by the appropriate library functions when they are at the end of a file.

For example:

```
while ((ch = getchar()) != EOF)
      {
      /* DO SOMETHING */
      }
```

`errno`

`errno` is a global variable, located in `stdio`, holding the last error message number from any `stdio` I/O routine.

ERROR MESSAGES

`noErr`	0	All is well - defined in MacTypes.h

Queuing Errors

`qErr`	-1	Queue element not found during deletion
`vTypErr`	-2	Invalid Queue element

Trap Call Errors

corErr	-3	Trap ("core routine") number out of range
unimpErr	-4	Unimplemented trap

Device Manager Errors

controlErr	-17	Driver error during Control operation
statusErr	-18	Driver error during Status operation
readErr	-19	Driver error during Read control
writErr	-20	Driver error during Write control
badUnitErr	-21	Bad unit number
unitEmptyErr	-22	No such entry in unit table
openErr	-23	Driver error during Open operation
closeErr	-24	Driver error during Close operation
dRemovErr	-25	Attempt to remove an open driver
dInstErr	-26	Attempt to install nonexistent driver
abortErr	-27	Driver operation aborted
notOpenErr	-28	Driver not open

File Manager Errors

dirFulErr	-33	directory full
dskFulErr	-34	disk full
nsvErr	-35	no such volume
ioErr	-36	i/o error
bdNamErr	-37	bad name error
fnOpnErr	-38	file not open
eofErr	-39	end of file
posErr	-40	tried to position before start of file
mFulErr	-41	memory full
tmfoErr	-42	too many files open
fnfErr	-43	file not found
wPrErr	-44	write protected disk
fLckdErr	-45	file is locked
vLckdErr	-46	volume is locked
fBsyErr	-47	file is busy
dupFNErr	-48	duplicate filename
opWrErr	-49	file already open with write permission
paramErr	-50	error in parameter list
rfNumErr	-51	refnum error
gfpErr	-52	get file position error
volOffLinErr	-53	volume not online (was ejected)
PermErr	-54	permission error (on file open)
volOnLinErr	-55	drive volume already online
nsDrvErr	-56	no such drive
noMacDskErr	-57	not a Mac diskette

extFSErr	-58	volume belongs to external fs
fsRnErr	-59	file system rename error
badMDBErr	-60	bad master directory block
wrPermErr	-61	write permission error

Low-Level Disk Errors

noDriveErr	-64	Drive isn't connected
offLinErr	-65	No disk in drive
noNybErr	-66	Disk is probably blank
noAdrMkErr	-67	Can't find an address mark
dataVerErr	-68	Read-verify failed
badCksmErr	-69	Bad address mark
badBtSlpErr	-70	Bad address mark
noDtaMkErr	-71	Can't find a data mark
badDCksum	-72	Bad data mark
badDBtSlp	-73	Bad data mark
wrUnderrun	-74	Write underrun occured
cantStepErr	-75	Drive error
tk0BadErr	-76	Can't find track 0
initIWMErr	-77	Can't initialize disk controller chip
twoSideErr	-78	Tried to read side 2 of a disk in a single read
spdAdjErr	-79	Can't correctly adjust disk speed
seekErr	-80	Drive Error
sectNFErr	-81	Can't find sector

Clock Chip Errors

clkRdErr	-85	Unable to read clock
clkWrErr	-86	Time written did not verify
prWrErr	-87	Parameter RAM written did not verify
prInitErr	-88	Validity status is not OxAB

AppleTalk Manager Errors

ddpSktErr	-91	DDP socket error
ddpLenErr	-92	DDP datagram too long
noBridgeErr	-93	No bridge found
lapProtErr	-94	ALAP error
excessCollsns	-95	ALAP no CTS receive after 32 tries
portInUse	-97	Driver open error, port already in use
portNotCf	-98	Driver open error, port not configured for use

Scrap Manager Errors

noScrapErr	-100	Desk scrap isn't initialized
noTypeErr	-102	No data of requested type

Memory Manager Errors

memFullErr	-108	Not enough room in heap zone
nilHandleErr	-109	Master Pointer was NIL in HandleZone
memWZErr	-111	WhichZone failed
memPurErr	-112	Trying to purge a locked/non-purgeable block
memLockedErr	-117	for MoveHHi

Resource Manager Errors

resNotFound	-192	Resource not found
resFNotFound	-193	Resource file not found
addResFailed	-194	Add resource failed
addRefFailed	-195	Add resource file failed
rmvResFailed	-196	Remove resource failed
rmvRefFailed	-197	Remove resource file failed

Additional AppleTalk Manager Errors

nbpBuffOvr	-1024
nbpNoConfirm	-1025
nbpConfDiff	-1026
nbpDuplicate	-1027
nbpNotFound	-1028
nbpNISErr	-1029
reqFailed	-1096
tooManyReqs	-1097
tooManySkts	-1098
badATPSkt	-1099
badBuffNum	-1100
noRelErr	-1101
cbNotFound	-1102
noSendResp	-1103
noDataArea	-1104
reqAborted	-1105
buf2SmallErr	-3101
noMPPError	-3102
cksumErr	-3103
extractErr	-3104
readQErr	-3105
atpLenErr	-3106

```
atpBadRsp       -3107
recNotFnd       -3108
sktClosedErr    -3109
```

FILE

FILE, the structure which holds the file descriptor, is defined below. This type definition creates a type called FILE which can be used anywhere you could use one of C's built in types (int, char, long, ...). Because of the library functions, you will probably never have to access the internals of a FILE directly. The fields are included here just in case.

Note: This manual will use "file descriptor" to refer to the C structure which holds the file information. Kernighan and Ritchie use "file descriptor" to refer to the number assigned to each open file. This manual will call that number the "file number". A "file pointer" is a pointer to a file descriptor (FILE *).

```
typedef   struct {
   int    refnum;          /* OS Reference number          */
   int    last_error;      /* holds last error on file     */
   int    fileno;          /* fnum for level 1 I/O         */
   unsigned user_buf:1,    /* 1 if user defined buffer     */
          InUse:1,         /* 0 if free                    */
          StdStream:1,     /* 0 = file, 1 = stdio          */
          rd:1,wr:1,       /* flag for read and write      */
          look_full:1,     /* flag for look_ahead          */
          mod:1,           /* file modified flag           */
          binary:1;        /* flag for binary file         */
   char   look_ahead;      /* char for look_ahead          */
   Ptr    filebuf;         /* Pointer to buffer            */
   int    fpos;            /* pos of file in buffer        */
   int    inbuf;           /* # of chars in buffer         */
   } FILE;
```

See also: _file, _console, stdin, stdout, stderr

_file

_file is an array of FILE (_file[_NFILE]) which is used to hold the file descriptors of all the files which are open. _file is used implicitly by all the file access functions. When you use open() to open a file, a FILE in _file will be dedicated to that file, and the index will be returned as a file number. If you use fopen() to open a file, the file pointer (FILE *) to the FILE in _file is returned instead of the file number. File I/O functions require either a file number or a file pointer as an argument. fileno() returns the corresponding file when given a file pointer. To go the other way, use _file[fn], the file pointer corresponding to file number fn.

See also: FILE, fopen(), open(), fileno()

_NFILE — _NFILE, the number of files that can be open at one time, is defined as 15.

NULL — NULL, the value of a pointer that is pointing to nothing, is defined as 0L. NULL is returned by many library functions (eg: `calloc()` or `sbrk()`). It is typically used to indicate an error by functions which return pointers. On every system, NULL will point to nothing, although its value will not always be 0L.

`stdin` — file pointer to the standard input
`stdout` — file pointer to the standard output
`stderr` — file pointer to the standard error

The three standard I/O file pointers may be used anywhere that a function takes a `FILE *` as an argument. For example, `getc(stdin)` gets a single character from `stdin`; `fputs("Hello", stdout)` puts the string Hello on the standard output. Some functions use the standard I/O automatically, most notably: `getchar()` and `putchar()`, `gets()` and `puts()`, `scanf()` and `printf()`. Library functions send their error messages to `stderr` automatically; your procedures can use it too or can send error messages into other files. The tables of input and output functions in the introduction to this section show which functions to use for `stdin` and `stdout`.

I/O from/to `stdin` and `stdout` can be redirected so that a file is used instead of the console. This allows programs to be written and compiled using `stdin` and `stdout`, and then connected to a file at run time.

Experienced C programmers from other systems should take note of the `cxxxxx` family of functions (`cputs`, `cprint`, `cgets`, `cscanf`, as well as `putch`, `getch`, `getche`) which send their output to the console automatically. I/O to/from the console using these functions cannot be redirected. The console I/O functions are included in the tables in the introduction.

abort (unix)

exit and call system debugger

SYNTAX

```
#include <unix.h>
void abort();
```

DESCRIPTION

Exits and calls the system debugger if it is loaded. Use `exit()` if you do not want the system debugger to be called. This function does not return or close files.

SEE ALSO

`exit()`, `_exit()`

SYNTAX

```
#include <storage.h>
int allmem();
```

DESCRIPTION

`allmem()` is equivalent to the call: `bldmem(0)`.

NOTE

The UNIX library procedures `getmem()`, `getml()`, `rlsmem()`, `rlsml()`, `allmem()`, `bldmem()`, `rbrk()`, `rstmem()`, `lsbrk()`, and `sbrk()` are not recommended for casual use. These procedures attempt to provide low-overhead UNIX-like memory allocation, and are provided for compatibility purposes.

RETURN VALUE

0 if success; -1 if failure.

SEE ALSO

`bldmem()`, `rbrk()`, `rstmem()`

atoi
atol
atof
(unix)

convert string to a number

SYNTAX

```
#include <unix.h>
int atoi(s)
char *s;

long atol(s)
char *s;

double atof(s)
char *s;
```

DESCRIPTION

These functions skip over leading white space (`isspace(*s)` is true) and convert the string `s` to the target type.

`atoi(s)` Converts `s` to an integer.

`atol(s)` Converts `s` to a long.

`atof(s)` Converts `s` to a floating point number.

A leading minus sign (-) indicates a negative number.

RETURN VALUE

The converted value of the string.

SEE ALSO

`stc_d()`, `stc_i()`, `stcd_i()`, `stch_i()`, `stci_d()`, `stcu_d()`

bldmem (storageu)

SYNTAX

```
#include <storage.h>
int bldmem(n)
int n;
```

DESCRIPTION

`bldmem()` allocates a pool of `n` K bytes of memory for subsequent `getmem()` and `getml()` calls.

Typically, you would make some initial calls to `lsbrk()` or `sbrk()` to get "level 1" memory. Then you would make a call to `allmem()` or `bldmem()` to set up a level 2 memory pool.

Memory allocated by `getmem()` and `getml()` calls after a call to `allmem()` or `bldmem()` can all be released by a call to `rstmem()`.

NOTE

The UNIX library procedures `getmem()`, `getml()`, `rlsmem()`, `rlsml()`, `allmem()`, `bldmem()`, `rbrk()`, `rstmem()`, `lsbrk()`, and `sbrk()` are not recommended for casual use. These procedures attempt to provide low-overhead UNIX-like memory allocation, and are provided for compatibility purposes.

RETURN VALUE

0 if success; -1 if failure.

SEE ALSO

`allmem()`, `rbrk()`, `rstmem()`

calloc (storage)

allocate and clear a block of memory

SYNTAX

```
#include <storage.h>
char *calloc(n,s)
unsigned n,s;
```

DESCRIPTION

`calloc()` dynamically allocates a block of `n*s` bytes of memory, and clears (zeroes) the block. `n` is the number of items desired, and `s` is the size of each item.

This routine uses the standard Macintosh memory manager to perform its function.

There are a half dozen different functions to dynamically allocate memory. See the tables in the introduction to this section for a comparison of the functions.

Use the `sizeof` operator to find the size of the item to assure portability (see the example). Although the block will be properly aligned, you should cast the pointer returned by `calloc()` so that it is guaranteed to be formatted as a pointer to the desired type of item.

`cfree()` or `free()` deallocates a block of memory that was allocated by `calloc()`.

EXAMPLES

```
typedef struct
{
     int day, month, year;
     int hours[24];
} date;
char *calloc();
date  *newmonth;
if ( (newmonth = (data *) calloc(31, sizeof(date)))
     != NULL )
   {/* do work */ }
```

RETURN VALUE A pointer to the newly allocated block; `NULL` if insufficient memory is available.

SEE ALSO `calloc()`, `malloc()`, `mlalloc()`, `cfree()`

CallPascal
CallPascalB
CallPascalW
CallPascalL
(MacTraps)

Call a Pascal Function Indirectly

SYNTAX

```
#include <pascal.h>
pascal void CallPascal();

pascal char CallPascalB();

pascal int CallPascalW();

pascal long CallPascalL();
```

DESCRIPTION

LightspeedC's `pascal` keyword allows you to define functions that will be called with Pascal calling conventions rather than C calling conventions.

However, functions that are called indirectly through pointers to functions are always called using C calling conventions. If you have a pointer to a Pascal function, you can call it by using one of these four `CallPascal()` routines.

Pass the same arguments as you would to the function you're trying to call, with a pointer to the function you're trying to call as an additional last argument.

free memory allocated by a memory allocation function

cfree (storage)

SYNTAX

```
#include <storage.h>
int cfree(cp)
char *cp;
```

DESCRIPTION

`cfree()` releases a block of memory which was dynamically allocated by `calloc()` or `malloc()`

The argument, `cp`, is a pointer to an allocated block. `cp` must have been allocated by `calloc()` or `malloc()`. If `cp` was not so allocated, anything can happen. There are several other library functions which free dynamically allocated memory. A table in the introduction to this section matches these deallocating functions to the corresponding allocating function(s).

RETURN VALUE

`0` if successful; `-1` if an error occurs.

SEE ALSO

`calloc()`, `malloc()`, `free()`, `rlsmem()`, `rlsml()`

cgetpid (unix)

concatenate string with process id number

SYNTAX

```
#include <unix.h>
char *cgetpid(str)
char *str;
```

DESCRIPTION

`cgetpid()` returns the user's string (`str`) concatenated with the process id number expressed as a five digit number with leading zeros. For example, if the process id = 1, then

```
printf("%s",cgetpid("Process"))
```

will print `Process00001`.

RETURN VALUE

A pointer to a string containing `str` concatenated with the process id as returned by `getpid()`.

SEE ALSO

`getpid()`, `setpid()`

get a string from the console

cgets
(stdio)

SYNTAX

```
#include <stdio.h>
char *cgets(s)
char *s;
```

DESCRIPTION

`cgets()` gets a string from the console (the keyboard) and puts it into the character string pointed to by `s`. The string is terminated when a carriage return (`0x0D`, `'\r'`) is entered from the keyboard. The input is always echoed to the screen.

(Note that this is not the usual C definition of a string. The carriage return is stripped off by `cgets`, and the string is stored internally in the standard C fashion - as a character array terminated by a null character (`'\0'`)).

Be sure that `s` is a character array long enough to handle any anticipated input, or else the excess input will be written into the memory that follows `s`, destroying whatever was there before.

`cgets()` echoes input to the screen.

RETURN VALUE

A pointer to string `s`; `NULL` if error.

SEE ALSO

`cputs()`

circle
(unix)

SYNTAX

```
#include <unix.h>
void circle (x, y, r)
int x, y, r;
```

DESCRIPTION

`circle()` draws a circle with center `x,y` and a radius `r`.

The Macintosh coordinate system is defined with 0,0 at the top left of the screen, and 512,342 at the bottom right. Coordinates are given in pixels. The radius is measured in pixels, as are the center coordinates.

SEE ALSO

`gotoxy()`, `label()`, `line()`, `move()`, `point()`

clear and allocate a long block of memory

clalloc (storage)

SYNTAX

```
#include <storage.h>
char *clalloc(n,s)
unsigned long n,s;
```

DESCRIPTION

`clalloc()` is yet another function to dynamically allocate memory. `clalloc()` is the same as `calloc()` except that its arguments are `long`s rather than integers. `clalloc()` differs from `malalloc()` in that `clalloc()` clears the block of memory.

In fact, there are a half dozen different functions to dynamically allocate memory. See the tables in the introduction to this section for a comparison of the functions. Be sure that both arguments in a call to `clalloc()` are `long`s; this usually requires casting the result of the `sizeof` operator, as illustrated in the example. `free()` and `cfree()` free memory allocated by `clalloc()`.

EXAMPLES

```
long block_ct;
char c, *c_ptr;
 ...
c_ptr = clalloc(block_ct, (long) (sizeof(c)));
```

RETURN VALUE

A pointer to the newly allocated block; `NULL` if there is insufficient memory.

SEE ALSO

`calloc()`, `malloc()`, `malalloc()`, `cfree()`, `free()`

clearerr
clrerr
(stdio)

clear last error flag of a stream

SYNTAX

```
#include <stdio.h>
void clearerr(stream)
FILE *stream;

void clrerr(stream)
FILE *stream;
```

DESCRIPTION

`clearerr()` resets the `last_error` flag of the file descriptor of the given `stream`. The `last_error` flag could be accessed directly by the syntax:

```
stream -> last_error
```

`stream` can be any pointer to a `FILE`.

`clrerr()` is a synonym for `clearerr()`. It too clears the last error flag of a given `stream`.

SEE ALSO

`FILE`

close (unix)

SYNTAX

```
#include <unix.h>
int close(fn)
int fn;
```

DESCRIPTION

Closes the file whose file descriptor number is fn. The file descriptor number is the number returned by `open()` when the file was last opened.

`exit()` automatically closes all open files. `fclose()` closes a file when given a file descriptor number.

RETURN VALUE

`0` if successful; `-1` if an error occurs.

SEE ALSO

`open()`, `fclose()`, `_closeall()`

_closeall (stdio)

close all open files

SYNTAX

```
#include <stdio.h>
int _closeall();
```

DESCRIPTION

`_closeall()` closes all open files that were opened via a call to `open()` or `fopen()`.

`_closeall()` is called automatically upon successful program termination.

Call `close()` or `fclose()` to close a single file.

RETURN VALUE

The number of files that could not be closed.

SEE ALSO

`close()`, `fclose()`, `open()`, `fopen()`

draw line from cursor to x,y

cont (unix)

SYNTAX

```
#include <unix.h>
void cont(x,y)
int x,y;
```

DESCRIPTION

`cont()` draws a line from the current screen position to position x,y. The Macintosh coordinate system is defined with 0,0 at the top left of the screen, and 512,342 at the bottom right.

SEE ALSO

`circle()`, `gotoxy()`, `label()`, `line()`, `move()`, `point()`

cprintf (stdio)

formatted print to the console screen

SYNTAX

```
#include <stdio.h>
int cprintf(format {,args})
char *format;
```

DESCRIPTION

`cprintf()` does a `printf` to the console screen. `printf` stands for formatted print, and is the most general way to output text and variables, whether to the console (`cprintf()`), a file (`fprintf()`), `stdout` (`printf()`) or to a string in memory (`sprintf()`). See `printf()` for a complete description of the format rules.

RETURN VALUE

The number of characters sent to the output `stream` if successful; `EOF` if an error occurs.

SEE ALSO

`printf()`, `sprintf()`, `fprintf()`

SYNTAX

```
#include <stdio.h>
void cputs(s)
char *s;
```

DESCRIPTION

`cputs()` puts a null terminated (`'\0'`) string on the user's console screen. Unlike `puts()`, `cputs` does NOT automatically send a carriage return (`'\r'`) or a newline (`'\n'`) when it reaches the end of the string. However, the string may explicitly include `'\r'` or `'\n'` characters. The string can be a variable or a constant, as shown in the example. The null character is not written to the console.

EXAMPLES

```
#include <stdio.h>
char *s, buf[20]
s = gets(buf);
cputs(s);
cputs("\r\nDONE\r\n");
```

SEE ALSO

`cgets()`, `puts()`, `fputs()`

creat (unix)

create a new file called filename

SYNTAX

```
#include <unix.h>
int creat(filename, mode)
char *filename;
int mode;
```

DESCRIPTION

`creat()` creates a new file named `filename`. That file is automatically opened in the specified mode. The mode argument is described in full under `open()`. The meaningful values for mode when using `creat()` are:

Opening flags:

`O_RDONLY`	read only
`O_WRONLY`	write only
`O_RDWR`	read/write
`O_APPEND`	append

Creation flags:

`O_CREAT`	create if file doesn't already exist
`O_TRUNC`	reset length to 0 (truncate) if file exists
`O_EXCL`	don't create if file exists

File type flags:

`O_TEXT`	text file
`O_BINARY`	binary file

OR a mode from "Opening Flags" to a mode from "File Type Flags" to create and open a specific file type in a specific way. If the file already exists (and `O_EXCL` is not set), `creat()` will delete and recreate it.

`creat()` returns a file number which is an index into `_file`, the array of file descriptors, and is used in calls to such functions as `read()`, `write()`, `close()`, and `lseek()`.

RETURN VALUE

The file number of the newly created and opened file if successful; `EOF` if error.

SEE ALSO

`open()`, `fopen()`, `FILE`, `_file`

SYNTAX

```
#include <stdio.h>
int cscanf(format {,args})
char *format;
```

DESCRIPTION

`cscanf()` does a `scanf()` from the keyboard of the console. `scanf()` stands for formatted scan, and is the most flexible way to read text in and convert it into a mixture of strings and numeric values.

Note that `cscanf()` takes a variable number of arguments -- it can read values into a variable number of variables. `format` is a string telling what sort of variables to expect, in what order, and with what text intermixed.

NOTE

`args` are POINTERS to the variables to read into, NOT the variables themselves. Remember to use the `&` operator to get the address of a variable. If you don't, you will overwrite random memory.

Refer to the description of `scanf()` for complete information about the `format` and `args` arguments.

`scanf()` and `printf()` are conceptually inverse functions (and therefore, `cscanf()` is to `scanf()` as `cprintf()` is to `printf()`).

RETURN VALUE

Number of items successfully input.

SEE ALSO

`scanf()`, `fscanf()`, `sscanf()`, `printf()`

ctime
(unix)

display time

SYNTAX

```
#include <unix.h>
char *ctime (clock)
unsigned long *clock;
```

DESCRIPTION

`ctime ()` returns a pointer to a string representing the time. The output is a 26 character string in the following form:

Sun Sep 16 01:03:52 1973\n\0

If `clock` is `NULL`, the time is taken from the Macintosh's time buffer; otherwise the time passed in is used.

EXAMPLES

```
#include <unix.h>
unsigned long a;
time (&a);  printf("%s", ctime(&a));
printf("%s", ctime(NULL));
```

RETURN VALUE

A pointer to a string containing the time.

SEE ALSO

`time(), localtime()`

Convert a C String to a Pascal String

CtoPstr (MacTraps)

SYNTAX

```
#include <pascal.h>
char *CtoPstr(s)
char *s;
```

DESCRIPTION

`CtoPstr()` converts a C string (terminated with a null character) to a Pascal string (preceded with the escape sequence `"\p"` and a count of the length of the string).

The string `s` is converted in place. However, this function also returns a pointer to the converted string.

There is a limit of 255 characters on the length of a Pascal string.

RETURN VALUE

A pointer to the converted string.

SEE ALSO

`PtoCstr()`

ctype (stdio)

tests whether character c has characteristic type

SYNTAX

```
#include <stdio.h>
int istype(c);
char c;
```

DESCRIPTION

There is a family of functions to make various tests on a character. All take a single character as an argument and return an `int` value. The return value is non-zero if the condition is true, 0 if it is false. The `istype` functions (with all numeric values in decimal) are:

`isalnum(c)`	True if `c` is a letter (`a-z`, `A-Z`) or a digit (`0-9`)
`isalpha(c)`	True if `c` is a letter (`a-z`, `A-Z`)
`isascii(c)`	True if `c` is an ASCII character (ASCII 32-128)
`iscntrl(c)`	True if `c` is a control character (ASCII 0-31)
`iscsym(c)`	True if `c` is a character that can appear in a valid C identifier (a letter, digit, or underscore)
`iscsymf(c)`	True if `c` is a character that can appear as the first character of a valid C identifier (a letter or an underscore)
`isdigit(c)`	True if `c` is a digit (`0-9`)
`isgraph(c)`	True if `c` is any printing character other than a space (ASCII `041` through `0176`)
`islower(c)`	True if `c` is a lower case letter (`a-z`)
`isodigit(c)`	True if `c` is an octal digit (`0-7`)
`isprint(c)`	True if `c` is a printable character (ASCII 32-255; but not 127)
`ispunct(c)`	True if `c` is a punctuation character. A punctuation character is one of: `!#$%&*()"'` `` ` ~ :;,<>/?|[]{}-_=+.`
`isspace(c)`	True if `c` is a space character. A space character is one of: space, `\t`, `\v`, `\n`, `\f` or `\r`.
`isupper(c)`	True if `c` is an upper case letter (`A-Z`)
`isxdigit(c)`	True if `c` is a hex digit (`0-9`, `A-F`, `a-f`).

The functions all use a predefined table called `__ctype` which contains a mask bit for each type of character, as listed below:

alpha	1	_ascii_	16
digit	2	_cntrl_	32
hex	4	_punct_	64
octal	8	_space_	178

With the exception of `isupper()` and `islower()`, which are true functions, these routines are macros that test the mask settings for the respective character.

RETURN VALUE non-zero if the condition is true; `0` if the condition is false.

SEE ALSO `toascii()`, `toupper()`, `tolower()`

eraseplot
(unix)

SYNTAX

```
#include <unix.h>
void eraseplot();
```

DESCRIPTION

`eraseplot()` erases (whites out) the screen. The cursor is left at 0,0 (the upper left corner of the screen).

exit
_exit
(unix)

SYNTAX

```
#include <unix.h>
void exit();
void _exit();
```

DESCRIPTION

`exit()` calls `fclose()` for each open file and then calls `_exit()` to leave the program and return to the shell. Closing the files includes flushing the I/O buffers.

`_exit()` assumes that the files are closed and exits the program immediately.

`exit()` is used more often than `_exit()` because `exit()` takes cares of pending I/O in the buffers while closing the files.

SEE ALSO

`abort()`, `fclose()`

fclose
(stdio)

close a file

SYNTAX

```
#include stdio.h
int fclose(stream)
FILE *stream;
```

DESCRIPTION

`fclose()` closes the given `stream`, flushing the I/O buffer if necessary. The file descriptor in `_file` used by `stream` is then available for another file.

The difference between `fclose()` and `close()` is that `fclose()` takes a file pointer as its argument while `close()` takes the file descriptor number.

RETURN VALUE

0 if success; `EOF` if error.

SEE ALSO

`open()`, `close()`, `_closeall()`

test whether at end of stream

feof (stdio)

SYNTAX

```
#include <stdio.h>
int feof(stream)
FILE *stream;
```

DESCRIPTION

`feof()` is used to test whether a previous I/O call to the given `stream` caused an EOF error (such as trying to get a character after having reached the end of the file.)

Naturally, if you use `clrerr()` or `clearerr()` to clear the error flag of that `stream`, `feof()` will no longer report an EOF error. Similarly, a later call with another type of error will also erase the EOF error flag.

RETURN VALUE

non-zero if `last_error` flag in the `stream` structure is an EOF error; 0 if `last_error` is not an EOF error.

SEE ALSO

`FILE`, `clrerr()`, `clearerr()`, `ferror()`

ferror (stdio)

test whether I/O error flag is set for stream

SYNTAX

```
#include <stdio.h>
int ferror(stream)
FILE *stream;
```

DESCRIPTION

`ferror()` tests whether any sort of I/O error flag is set for the given `stream`. `clrerr()` and `clearerr()` will reset the error flag.

RETURN VALUE

non-zero if any I/O error had occurred; `0` if the `last_error` flag in the `stream` structure is not set.

SEE ALSO

`feof()`, `clrerr()`, `clearerr()`

flush I/O buffer of stream

fflush (stdio)

SYNTAX

```
#include <stdio.h>
int fflush(stream)
FILE *stream;
```

DESCRIPTION

`fflush()` flushes the I/O buffer of the specified `stream`. This is often done if an I/O error has occurred and you want to start anew. Buffers should also be flushed before closing files.

Note that `fflush()` returns `0` if successful.

RETURN VALUE

`0` if success; `EOF` if an error occured while trying to flush the buffer.

SEE ALSO

`fopen()`

fgetc
(stdio)

get a character from stream

SYNTAX

```
#include <stdio.h>
int fgetc(stream)
FILE *stream;
```

DESCRIPTION

`fgetc()` reads in the next character from the given `stream`. Since `fgetc()` returns the integer value of the character, it can be used to get bytes from a binary file.

Use `fread()` or `fgets()` to read in multiple characters more easily and efficiently, and use `fscanf()` to read text and data directly into variables.

RETURN VALUE

`EOF` if error or at the EOF of the stream; the integer value of the character otherwise.

SEE ALSO

`fread()`, `fgets()`, `fscanf()`, `getc()`, `gets()`, `read()`, `scanf()`

get a string from stream and put in array

fgets (stdio)

SYNTAX

```
#include <stdio.h>
char *fgets(s, n, stream)
char *s;
int n;
FILE *stream;
```

DESCRIPTION

`fgets()` tries to read the next `n` characters from `stream` and store them in the character array `s`. It adds the null character (`'\0'`) to the character array after the `n`th character, and returns a pointer to `s`. Therefore, `s` must be declared as a character array large enough to hold `n+1` characters.

Two events will stop `fgets()` before it has read in `n` characters. First, if it reads in a newline (`'\n'`), it adds the `'\n'` to the string, adds the `'\0'`, and returns `s`. Second, if `fgets()` reads in an EOF, it considers it an error, sets the EOF error flag, stops trying to read in characters from the file, and returns a `NULL` pointer.

Use `ferror` to find out which error occurred last.

RETURN VALUE

`NULL` if error or if EOF; `s` (the first argument) otherwise.

SEE ALSO

`fgetc()`, `fread()`, `fscanf()`, `ferror()`

fileno (unix)

get file number when given file descriptor

SYNTAX

```
#include <unix.h>
int fileno(stream)
FILE *stream;
```

DESCRIPTION

`fileno()` returns the file number of `stream`, where `stream` is a pointer to a file descriptor. The file number is simply the number of the file descriptor in the array of open file descriptors (see `_file` and `FILE`). Some library routines (`close()`, `read()`, `write()`, `lseek()`) access a file by using the file number, others by using the pointer to the actual file descriptor.

Files can be opened using either `fopen()` or `open()`; `fopen()` returns the pointer to the file descriptor, while `open()` returns the file number. Therefore, you can open a file with `fopen()` (returning the file descriptor pointer) and later use `fileno()` to get the file number and use the library functions which use the file number. There is no library function to go the other way. Instead use `_file[fn]` which is the file pointer of file number `fn`.

RETURN VALUE

The file number of `stream`, if success; `EOF` if error.

SEE ALSO

`_file`, `FILE`, `fopen()`, `open()`

fopen (stdio)

SYNTAX

```
#include <stdio.h>
FILE *fopen(pathname, type)
char *pathname;
char *type;
```

DESCRIPTION

`fopen()` opens a stream to `pathname` by creating a file descriptor as well as doing the necessary operating system work. `fopen()` returns the pointer to the newly created file descriptor. The file is opened according to the `type` (the values for `type` are described below). The effect of `type` depends on whether the given `pathname` can be found: a new file is created if `pathname` does not exist, `pathname` is opened if it already exists.

Files can also be opened with `open()`, which returns the file number in the array of file descriptors rather than the pointer to the file descriptor. Use `open()`, `close()`, etc. instead of the standard C library file calls `fopen()`, `fclose()`, etc. when you are concerned with UNIX compatibility.

`fopen()` and `open()` are not synonymous. Not only do they interpret their arguments differently, their arguments are of different types. Note that some library functions access a file by its file pointer, others by its file number. Therefore, your choice of `fopen()` vs. `open()` will dictate your choice of other library routines.

The values for `type` are strings and are:

`"r"`	open an existing file for reading.
`"w"`	create a file for writing if it does not exist; delete and then recreate a file for writing if it already exists.
`"a"`	create a file for writing if it does not exist; open a file and place pointer at the EOF for appending if it already exists.
`"r+"`	open an existing file for reading and writing.
`"w+"`	create a file for reading and writing it if does not exist; delete and recreate a file for reading and writing if it already exists.

"a+" create a file for reading and writing if it does not exist; open a fplace pointer at the EOF for appending and reading if it already exists.

b binary file: '\r' not converted to '\n' on reads, '\n' not converted to '\r' on writes.

b is not in quotes because it is added to one of the other values for type. For example, "rb", "wb", "ab", "r+b", "w+b", "a+b". "b" alone is meaningless and is an error.

Note that the type argument is a character pointer, not a single character. Therefore, newfile = fopen("foo", "r") is legal while newfile = fopen("foo", 'r') will not operate as intended, and will have unpredictable results.

RETURN VALUE NULL if error; the pointer to the file descriptor if success.

SEE ALSO open(), _file, FILE, fileno(), fclose(), fflush(), fread(), fwrite()

format and print to a file

fprintf (stdio)

SYNTAX

```
#include <stdio.h>
int fprintf(stream, format, {, args})
FILE *stream;
char *format;
```

DESCRIPTION

`fprintf()` formats and prints output to the specified `stream`. It is used to print a combination of text and data to the file pointed to by `stream`. In all other ways, it is identical to `printf()` which does a formatted print to the standard output. See `printf()` for a description of the `format` and optional `args` arguments.

`fputc()`, `fputs()`, and `fwrite()` put a character, a string, and a block of text in a file, respectively: they are simpler and more efficient, but are not as general and cannot deal with numeric data.

`fscanf()` is the inverse of `fprintf()`: it does a formatted read of text and data from a file.

RETURN VALUE

The number of characters sent to the output `stream` if successful; `EOF` if an error occurs.

SEE ALSO

`printf()`, `cprintf()`, `sprintf()`, `fputc()`, `fputs()`, `fwrite()`, `fscanf()`

fputc (stdio)

put a character into a file

SYNTAX

```
#include <stdio.h>
int fputc(c, stream)
char c;
FILE *stream;
```

DESCRIPTION

`fputc()` adds a single character `c` to `stream`. See `fputs()`, `fwrite()`, and `fprintf()` for ways to add strings, blocks of text, or combinations of text and data to a file. `putc()` puts a single character to the standard input.

RETURN VALUE

`EOF` if error (such as a read only file); `c`, the integer value of the character, if success.

SEE ALSO

`putc()`, `fputs()`, `fwrite()`, `fprintf()`

put a string into a file

fputs (stdio)

SYNTAX

```
#include <stdio.h>
int fputs(s, stream)
char *s;
FILE *stream;
```

DESCRIPTION `fputs()` puts a null terminated (`'\0'`) string to `stream`. The `'\0'` is not written to the file.

RETURN VALUE `0` if successful; otherwise `EOF`.

SEE ALSO `cputs()`, `puts()`, `fprintf()`, `fwrite()`, `fputc()`

fread (stdio)

read a block of characters from a file

SYNTAX

```
#include <stdio.h>
int fread(ptr, size_of_ptr, count, stream)
char *ptr;
unsigned size_of_ptr;
int count;
FILE *stream;
```

DESCRIPTION

`fread()` tries to read `count` items from `stream` and store them in the block pointed to by `ptr`. The `size_of_ptr` field should be an unsigned integer that is the size of the object pointed to by the pointer. If `fread` reaches the end of the file before it has read `count` bytes, it sets the EOF error flag and returns the number of bytes it was able to read.

`read()` is a similar function to read a block of data from a file. `read()` accesses the file by its file number rather than its file pointer. `fscanf()`, `fgets()`, `fgetc()` and `getc()` are other functions to get data from a file.

EXAMPLES

```
#include <stdio.h>
struct FOO {
            int a;
            char b;
           } foo[3];
fread (&foo[0], sizeof(struct FOO), 3, stdin);

/* reads 3 items of sizeof FOO into buffer
   pointed to by foo from stdin */
```

RETURN VALUE

0 on immediate end of file or error (use `ferror()` and `feof()` to tell); otherwise the number of items transferred.

SEE ALSO

`read()`, `fscanf()`, `fgetc()`, `fgets()`, `getc()`

release whole block of memory

free (storage)

SYNTAX

```
#include <storage.h>
int free(cp)
char *cp;
```

DESCRIPTION

`free()` releases a block of memory which was dynamically allocated by `calloc()`, `malloc()`, `clalloc()` or `mlalloc()`. `free()` automatically knows how much memory to free. It just needs `cp`, the pointer to the start of the block. `cp` must have been a pointer that was returned by `calloc()`, `malloc()`, `clalloc()` or `mlalloc()`. The whole block is freed.

RETURN VALUE

`0` if successful; `-1` if an error occurs.

SEE ALSO

`calloc()`, `malloc()`, `cfree()`

freopen (stdio)

redirect output to different file

SYNTAX

```
#include <stdio.h>
FILE *freopen(filename, type, stream)
char *filename, *type;
FILE *stream;
```

DESCRIPTION

`freopen()` closes the file whose file descriptor is `stream` and calls `fopen()`, passing the `filename` and `type`. The file will be opened using the same string that was used to close it.

This routine is used to redirect a `stream`'s output to a different file. For example, after the call `freopen("abc", "w", stdout)`, all output to `stdout` will go to the file `"abc"`.

RETURN VALUE

`NULL` if error (sets an error code in `errno`); otherwise `stream`.

SEE ALSO

`fopen()`, `fclose()`

SYNTAX

```
#include <stdio.h>
int fscanf(stream, format {, args})
FILE *stream;
char *format;
```

DESCRIPTION

`fscanf()` does a formatted scan taking input from `stream`. It breaks the input up into characters, strings, integers, etc. according to format, and stores that formatted data in the variables pointed to by `args`. There can be a variable number of `args`. The `args` must be pointers to variables, not the variables themselves. See `scanf()` for a description of the format argument.

CAUTION

The most common mistake when using `fscanf()` is to use the variables themselves rather than the pointers to the variables. `fscanf()` needs to know the address of the variables it will use to store data, not their present values. Failing to use addresses to store the values will overwrite random memory. See the examples in `scanf()`. If `fscanf()` runs out of arguments before reaching the end of s, arbitrary values from the stack are used as the address of the arguments and overwrite random memory.

Similar functions are `scanf()` (from `stdio`), `cscanf()` (from the console), and `sscanf()` (from a string). Other functions to get data from a file include `getc()`, `fgetc()`, `fgets()`, `read()`, and `fread()`. `fprintf()` does a formatted print of data to a file.

RETURN VALUE

Number of items successfully input.

SEE ALSO

`scanf()`, `cscanf()`, `sscanf()`

fseek (stdio)

move to a different point within a file

SYNTAX

```
#include <stdio.h>
int fseek(stream, offset, type)
FILE *stream;
long offset;
int type;
```

DESCRIPTION

`fseek()` allows for non-sequential I/O to a `stream`. When a file is opened for reading or writing, a "logical cursor" is placed at the beginning of the file. That logical cursor is advanced through the file with each read or write. `fseek()` and its cousin `lseek()` move that logical cursor. (`lseek()` differs in two main ways: it takes a file number rather than a pointer to the file descriptor and it returns a `long` rather than an `int`).

The type argument determines where the seek begins:

0	offset relative to the beginning of the file
1	offset relative to current file position
2	offset relative to the end of file

`offset` is simply the count in bytes from the starting point specified by the type argument. Note that `offset` is a `long`, and can be either positive or negative.

EXAMPLES

```
FILE *fd;
fseek(fd, 0L, 0);   /* move to start of file */
fseek(fd, 20L, 1);  /* move forward in file 20 bytes */
fseek(fd, 0L, 2);   /* move to end of file */
```

RETURN VALUE

Zero if success, non-zero if error.

SEE ALSO

`lseek()`

SYNTAX

```
#include <stdio.h>
long ftell(stream)
FILE *stream;
```

DESCRIPTION

`ftell()` returns the current position within the file. `ftell()` returns `-1L` if an error has occured.

RETURN VALUE

Current position within file if successful; `-1L` if error.

SEE ALSO

`tell()`, `fseek()`

fwrite (stdio)

write a block of characters to a file

SYNTAX

```
#include <stdio.h>
int fwrite(ptr, size_of_ptr, count, stream)
char *ptr;
unsigned size_of_ptr;
int count;
FILE *stream;
```

DESCRIPTION

`fwrite()` writes a block of data to `stream`. `count` is the number of items to be written, and `ptr` is a pointer to the start of the block. `size_of_ptr` is the size of the item to write.

`write()` also writes a block of data to a file. `write()` differs in two ways: it accesses the file by its file number and it simply writes a specified number of bytes to the file.

`fprintf()`, `fputs()`, `fputc()`, and `putc()` are other functions which add data to a file. `fread()` is the inverse of `fwrite()`.

RETURN VALUE

`0` if error (check with `ferror()` and `feof()`); otherwise number of items transferred.

SEE ALSO

`fread()`, `write()`, `fprintf()`, `fputs()`, `fputc()`, `putc()`

SYNTAX

```
#include <stdio.h>
int getc(stream)
FILE *stream;
```

This is a macro calling `fgetc()`

DESCRIPTION

`getc()` gets the next character from `stream`. `getc()` returns that character as an `int` so that it can be used to get bytes from a binary file.

`fgetc()` also gets a character from a file, while `fscanf()`, `fread()`, `fgets()` and `read()` get larger groups of data. `putc()` is the inverse of `getc()` and adds a character to a file.

RETURN VALUE

The integer value of the next character from `stream`; `EOF` on end of file or error.

SEE ALSO

`fgetc()`, `fgets()`, `fscanf()`, `fread()`, `read()`, `putc()`

getch
(stdio)

get next character from keyboard, don't echo

SYNTAX

```
#include <stdio.h>
int getch();
```

DESCRIPTION

`getch()` gets the next character from the keyboard. The character is not echoed. (`getche()` is the same as `getch()` except the character is always echoed to the screen).

Note that the return value is an `int`, not a `char`. This is so that `getch()` can return all possible characters (0-255) and `EOF` (-1).

RETURN VALUE

The integer value of the character; or `EOF` on end of file or error.

SEE ALSO

`getchar()`, `getche()`

get next character from stdin

getchar (stdio)

SYNTAX

```
#include <stdio.h>
int getchar()
```

`getchar()` is a macro calling `fgetc(stdin)`

DESCRIPTION

`getchar()` gets the next character from `stdin`. The character is not echoed to `stdout`. Note that the return value is an `int`, not a `char`. This is so that `getchar()` can return all possible characters (0-255) and `EOF` (-1).

RETURN VALUE

The integer value of the character; or `EOF` on end of file or on error.

SEE ALSO

`getc()`, `getch()`, `getchar()`, `getche()`, `fgetc()`

getche (stdio)

get next character from keyboard, echo to screen

SYNTAX

```
#include <stdio.h>
int getche();
```

DESCRIPTION

`getche()` is the same as `getch()` except that it echoes the character to the screen. See the description of `getch()`.

RETURN VALUE

The integer value of the character from the keyboard; or `EOF` on end of file.

SEE ALSO

`getc()`, `getch()`, `getchar()`, `getche()`, `fgetc()`

dynamically allocate n bytes of memory

getmem (storageu)

SYNTAX

```
#include <storage.h>
char *getmem(n)
unsigned n;
```

DESCRIPTION

`getmem()` gets `n` bytes of memory from the free memory pool allocated by `bldmem()`. The memory is not automatically zeroed.

`getml()` is the same as `getmem()` except that it can allocate a block of memory that can be greater than 65535 bytes. There are actually a half dozen different functions which dynamically allocate memory. They are described together in the introduction to this section as well as on the separate pages describing each function.

Use `rlsmem()` to free memory allocated by `getmem()`.

NOTE

The UNIX library procedures `getmem()`, `getml()`, `rlsmem()`, `rlsml()`, `allmem()`, `bldmem()`, `rbrk()`, `rstmem()`, `lsbrk()`, and `sbrk()` are not recommended for casual use. These procedures attempt to provide low-overhead UNIX-like memory allocation, and are provided for compatibility purposes.

RETURN VALUE

A pointer to the newly allocated block; or `NULL` if error (such as insufficient memory).

SEE ALSO

`getml()`, `rlsmem()`, `rlsml()`

getml (storageu)

dynamically allocate n bytes of memory

SYNTAX

```
#include <storage.h>
char *getml(n)
unsigned long n;
```

DESCRIPTION

`getml()` gets `n` bytes of memory from the free memory pool allocated by `bldmem()`. The memory is not automatically zeroed. `getmem()` is the same as `getml()` except that it can allocate only an integer length block of memory.

There are actually a half dozen different functions which dynamically allocate memory. They are described together in the introduction to this section as well as separately by name.

Use `rlsml()` to free memory allocated by `getml()`.

NOTE

The UNIX library procedures `getmem()`, `getml()`, `rlsmem()`, `rlsml()`, `allmem()`, `bldmem()`, `rbrk()`, `rstmem()`, `lsbrk()`, and `sbrk()` are not recommended for casual use. These procedures attempt to provide low-overhead UNIX-like memory allocation, and are provided for compatibility purposes.

RETURN VALUE

A pointer to the newly allocated block; or `NULL` if error (such as insufficient memory).

SEE ALSO

`getmem()`, `rlsml()`

return process id number

getpid (unix)

SYNTAX

```
#include <unix.h>
int getpid();
```

DESCRIPTION

`getpid()` returns the process id number. This number is initially 1, but can be changed using `setpid()`.

This routine is provided for UNIX compatibility. It has no real function.

RETURN VALUE

The process id number.

SEE ALSO

`setpid()`

gets (stdio)

get characters from stdin till newline and store in s

SYNTAX

```
#include <stdio.h>
char *gets(s)
char *s;
```

DESCRIPTION

`gets()` reads characters in from `stdin` to the character array `s`. When it reads in a newline (`'\n'`), the `'\n'` is discarded and a null character (`'\0'`) is added to `s` to terminate the string. Therefore, `s` must be large enough to hold one more character than is anticipated.

`fgets()` and `cgets()` get strings from a file and from the console respectively. They have some differences so check their descriptions.

`getchar()` and `scanf()` also read in data from `stdin`.

RETURN VALUE

`s`, the first argument, is returned if success; `NULL` if EOF or error (use `ferror()` and `feof()` to determine cause).

SEE ALSO

`cgets()`, `fgets()`, `getchar()`, `scanf()`

return user id number

getuid (unix)

SYNTAX

```
#include <unix.h>
int getuid();
```

DESCRIPTION

`getuid()` returns the user id number. This number is initially 1, but can be changed using `setuid()`.

This routine is provided for UNIX compatibility. It has no real function.

RETURN VALUE

The user's id number.

SEE ALSO

`setuid()`

getw (unix)

return the next two bytes of stream as integer

SYNTAX

```
#include <unix.h>
int getw(stream)
FILE *stream;
```

DESCRIPTION

`getw()` returns the next two bytes of `stream` as an integer. The bytes are assumed to be in memory image order (that is, according to the 68000 convention, with the MSB in the first byte).

EXAMPLE

```
int i,j;
i=0x1234;
 .
 .
 .
fputc(i>>8 & 0x00ff, fp);  /* MSB */
fputc(i & 0x00ff, fp);     /* lsb */
fseek(fp, -2L, 1);         /* move back 2 bytes */
j=getw(fp);                /* i==j */
```

RETURN VALUE

EOF is returned if an error occurs.

SEE ALSO

`putw()`

move cursor on the screen

gotoxy (stdio)

SYNTAX

```
#include <stdio.h>
void gotoxy(x,y);
int x,y;
```

DESCRIPTION

`gotoxy()` simply moves the cursor on the screen to the given `(x,y)` character position (as in standard I/O).

`0,0` is in the top left corner of the screen. The range of possible coordinates will depend on the point size. For the default font (25 lines by 80 characters), the bottom right corner coordinates will be `80,25`. Character positions are calculated for a monospaced font.

Note that this is different from `move()`, which uses a pixel-based coordinate system.

SEE ALSO

`circle()`, `label()`, `line()`, `move()`, `point()`

kbhit
(stdio)

test whether keyboard was hit

SYNTAX

```
#include <stdio.h>
int kbhit();
```

DESCRIPTION

`kbhit()` tests whether there is a character available from the keyboard without getting the character. `kbhit()` is provided so that you can write a loop to watch until the keyboard is hit, doing other work or with a timeout. `getch()` and similar functions have no timeout, and so take control for an indefinite period of time. Once the keyboard is hit, you can use `getch()` and know that it will return.

RETURN VALUE

`1` if a keyboard character is available; `0` if no keyboard character is available.

SEE ALSO

`getch()`, `getche()`

SYNTAX

```
#include <unix.h>
void label(s)
char *s;
```

DESCRIPTION

`label()` writes the null-terminated string `s` at the current cursor position.

The Macintosh coordinate system is defined with 0,0 at the top left of the screen, and 512,342 at the bottom right. Coordinates are given in pixels.

SEE ALSO

`circle()`, `gotoxy()`, `line()`, `move()`, `point()`

line
(unix)

plot line from x1,y1 to x2,y2

SYNTAX

```
#include <unix.h>
void line(x1,y1,x2,y2)
int x1,y1,x2,y2;
```

DESCRIPTION

`line()` draws a line from pixel `(x1,y1)` to pixel `(x2,y2)` using Macintosh QuickDraw calls.

The Macintosh coordinate system is defined with 0,0 at the top left of the screen, and 512,342 at the bottom right. Coordinates are given in pixels.

SEE ALSO

`circle()`, `gotoxy()`, `label()`, `move()`, `point()`

return pointer to time record

localtime (unix)

SYNTAX

```
#include <unix.h>
tm *localtime(clock)
unsigned long clock;
```

DESCRIPTION

`localtime()` returns a pointer to a UNIX-compatible time record called `tm`.

The record is of the following form (defined in `unix.h`):

```
typedef struct{
     int tm_sec;      /* seconds 0-59 */
     int tm_min;      /* minutes 0-59 */
     int tm_hour;     /* hours 0-23 */
     int tm_mday;     /* day of month (1-31) */
     int tm_mon;      /* month of year (0-11) */
     int tm_year;     /* year -1900 */
     int tm_wday;     /* day of week (sunday=0) */
     int tm_yday;     /* day of year (0-365) */
     int tm_isdst;    /* daylight savings time
                         - NOT SUPPORTED ! */
} tm;
```

RETURN VALUE

A pointer to `tm`.

SEE ALSO

`time()`, `ctime()`

SYNTAX

```
#include <unix.h>
char *locv(hi,lo)
int hi,lo;
```

DESCRIPTION

`locv()` converts a signed double-precision integer (long), passed as two int arguments, to an ASCII string containing a decimal representation of that number. The value of the signed double-precision integer is (hi<<16) | lo.

This function is provided for UNIX compatibility.

RETURN VALUE

A pointer to a buffer containing the converted null terminated decimal string.

NOTE

Since `locv` returns a pointer to a static buffer it cannot be used more than once in an expression because the second call will overwrite the contents of the buffer.

SYNTAX

```
#include <storage.h>
char *lsbrk(n)
unsigned long n;
```

DESCRIPTION

`lsbrk()` allocates a long block of bytes. The block is not zeroed. This is a low-level UNIX call for getting memory. There are half a dozen different ways to allocate memory. They are described together in the introduction to this section as well as individually by function name.

NOTE

The UNIX library procedures `getmem()`, `getml()`, `rlsmem()`, `rlsml()`, `allmem()`, `bldmem()`, `rbrk()`, `rstmem()`, `lsbrk()`, and `sbrk()` are not recommended for casual use. These procedures attempt to provide low-overhead UNIX-like memory allocation, and are provided for compatibility purposes.

RETURN VALUE

`NULL` if error; the pointer to the newly allocated block if success.

SEE ALSO

`rstmem()`, `sbrk()`, `getml()`, `getmem()`

lseek (unix)

move to a different point inside a file

SYNTAX

```
#include <unix.h>
long lseek(fn, offset, type)
int fn;
long offset;
int type;
```

DESCRIPTION

`lseek()` allows for non-sequential I/O to a stream. When a file is opened for reading or writing, a "logical cursor" is placed at the beginning of the file. That logical cursor is marched forward through the file with each read or write. `lseek()` and its cousin `fseek()` move that logical cursor. `fseek()` is a standard C library function whereas `lseek()` is the UNIX version. (`fseek()` differs in two main ways: it takes a pointer to a file descriptor rather than a file number and it returns an `int` rather than a `long`.)

The type argument determines where the seek begins:

0 offset relative to the beginning of the file
1 offset relative to current file position
2 offset relative to the end of file

`offset` is simply the count in bytes from the starting point chosen by the `type` argument.

CAUTION

`offset` is a `long`, and can be either positive or negative.

RETURN VALUE

`EOF` if error; position in file if success.

SEE ALSO

`fseek()`

SYNTAX

```
#include <storage.h>
char *malloc (n)
unsigned n;
```

DESCRIPTION

`malloc()` dynamically allocates n bytes of memory. Unlike `calloc()` it does not clear the block of memory. `malloc()` returns a pointer to the block of memory it has allocated. `free()` and `cfree()` free memory allocated by `malloc()`.

There are actually several different functions to allocate memory. They are described together in the introduction to this section as well as separately by name.

RETURN VALUE

A pointer to the block if success; or NULL if failure.

SEE ALSO

`free()`, `cfree()`, `calloc()`, `clalloc()`, `mlalloc()`

The various mathematical functions in the math library are described in the tables below.

CAUTION

It is very important to make sure that functions returning double are declared to be return double. Otherwise, random memory may be trashed. (See Chapter 9 for details.)

Trigonometric Functions

Definition	Value	Notes
`double acos(x)` `double x;`	Arc cosine of `x`	Return value is measured in radians. If `x < -1` or `x > 1`, 0 is returned to indicate an error, and the error code EDOM is stored in the external variable `errno`.
`double asin(x)` `double x;`	Arc sine of `x`	Return value is measured in radians. If `x < -1` or `x > 1`, 0 is returned to indicate an error, and the error code EDOM is stored in the external variable `errno`.
`double atan(x)` `double x;`	Arc tangent of `x`	Return value is measured in radians.
`double atan2(y,x)` `double y,x;`	Arc tangent of `y/x`	Return value is measured in radians. If both `x` and `y` are 0, 0 is returned to indicate an error, and the error code EDOM is stored in the external variable `errno`.
`double cos(x)` `double x;`	Cosine of `x`	`x` is measured in radians.
`double sin(x)` `double x;`	Sine of `x`	`x` is measured in radians.
`double tan(x)` `double x;`	Tangent of `x`	`x` is measured in radians.

Algebraic Functions

Definition	Value	Notes
`double exp(x)` `double x;`	Exponential function: e^x	If the argument is so large that the result cannot be represented, then the largest possible floating point number is returned, and the error code `ERANGE` is stored into the external variable `errno`.
`double fmod(x,y)` `double x,y;`	x modulus y	See also `modf()`.

`double frexp(x,nptr)`
`double x;`
`int *nptr;`
	Split a floating point number into a fraction and an exponent.
	The fraction is returned, and the exponent is stored into the location pointed to by `nptr`.
		`ldexp` performs the inverse function.

`double ldexp(x,n)`
`double x;`
`int n;`
	$x * 2^n$	`frexp` performs the inverse function.

`double log(x)` `double x;`	Natural (base e) logarithm of x	If $x \leq 0$, the negative of the largest possible floating point number is returned, and the error code `ERANGE` is stored into the external variable `errno`.
`double log10(x)` `double x;`	Base 10 logarithm of x	If $x \leq 0$, the negative of the largest possible floating point number is returned, and the error code `ERANGE` is stored into the external variable `errno`.

`double modf(x,nptr)`
`double x;`
`int *nptr;`
	x modulus `*nptr`	See also `fmod()`.

`double pow(x,y)` `double x,y;`	x^y	Power function
`double sqrt(x)` `double x;`	Square root of x	The positive square root is returned. If $x < 0$, 0.0 is returned to indicate an error, and the error code `EDOM` is stored into the external variable `errno`. See also `pow()`.

Hyperbolic Functions

Definition	Value	Notes
`double cosh(x)` `double x;`	Hyperbolic cosine of `x`	
`double sinh(x)` `double x;`	Hyperbolic sine of `x`	
`double tanh(x)` `double x;`	Hyperbolic tangent of `x`	

Random Numbers

Definition	Value	Notes
`rand()`	A random integer	See also `srand()`, which seeds random number generator.
`srand(seed)` `unsigned int seed;`	Seeds the rand() function; returns a random integer based on seed	See also `rand()`, which returns a random number when called.

Miscellaneous

Definition	Value	Notes
`int abs(x)` `int x;`	Absolute value of `x`	See also `fabs()`, `labs()`.

`double ceil(x)`
`double x;`
	Ceiling of `x`	For example: `ceil(5.2)=6.0`
		`ceil(4.0)=4.0`
		`ceil(-3.7)=-3.0`
		See also `floor()`.

`double fabs(x)` `double x;`	Absolute value of `x`	See also `abs()`, `labs()`.

`double floor(x)`
`double x;`
	Floor of `x`	For example: `floor(5.2)=5.0`
		`floor(4.0)=4.0`
		`floor(-3.7)=-4.0`
		See also `ceil()`.

`long int labs(x)` `long int x;`	Absolute value of `x`	See also `abs()`, `fabs()`.

allocate a long block of memory

mlalloc (storage)

SYNTAX

```
#include <storage.h>
char *mlalloc(n);
unsigned long n;
```

DESCRIPTION

`mlalloc()` dynamically allocates `n` bytes of memory. `n` is a `long`, so `mlalloc()` can allocate a long block with one call. Unlike `clalloc()`, `mlalloc` does not clear that block of memory. `mlalloc()` returns a pointer to the block of memory it has allocated. `cfree()` and `free()` deallocate as they do with `malloc()`.

There are actually several different functions to allocate memory. They are described together in the introduction to this section as well as separately by name.

RETURN VALUE

A pointer to the block if success; `NULL` if failure.

SEE ALSO

`calloc()`, `clalloc()`, `malloc()`, `cfree()`, `free()`

move (unix)

move cursor to x,y

SYNTAX

```
#include <unix.h>
void move(x,y)
int x,y;
```

DESCRIPTION

`move()` moves the cursor to Macintosh pixel position `x,y`.

The Macintosh coordinate system is defined with 0,0 at the top left of the screen, and 512,342 at the bottom right. Coordinates are given in pixels.

SEE ALSO

`circle()`, `gotoxy()`, `label()`, `line()`, `point()`

move n bytes from one position to another

movmem (unix)

SYNTAX

```
#include <unix.h>
void movmem(s,d,n)
char *s,*d;
unsigned n;
```

DESCRIPTION

movmem() copies n bytes from location s in memory to location d. This is a block copy operation. The routine checks the relative positions of the source and destination to ensure that data is not overwritten during the move.

EXAMPLES

```
char s[100];
char t[200];
movmem(&t[100], s, sizeof(s));
     /* copies last 100 elements of t into s */
```

SEE ALSO

repmem(), setmem()

onexit
(stdio)

call a Procedure before program exit

SYNTAX

```
#include <stdio.h>
typedef void (*Proc)();
Proc *onexit (proc)
Proc (proc);
```

DESCRIPTION

Arrange for C function `Proc` to be called just before program exit. `Proc` is called with no arguments. Up to 32 `Proc` routines may be specified. They are called in the reverse order in which they were supplied to `onexit()`.

RETURN VALUE

`NULL` if `onexit()` was unsuccessful; or `Proc` if successful.

SEE ALSO

`exit()`, `_exit()`

open (unix)

SYNTAX

```
#include <unix.h>
int open(filename, mode)
char *filename;
int mode;
```

DESCRIPTION

`open()` opens `filename` for reading or writing, according to `mode`. `open()` returns the file number of the open file. That file number is used by the library functions `read()`, `write()`, `lseek()`, and `close()`. `mode` is made by adding together one flag from each of the three groups below.

Opening flags:

`O_RDONLY`	read only
`O_WRONLY`	write only
`O_RDWR`	read/write
`O_APPEND`	append

Creation flags:

`O_CREAT`	create if file doesn't already exist
`O_TRUNC`	reset length to 0 (truncate) if file exists
`O_EXCL`	don't create if file exists

File type flags:

`O_TEXT`	text file
`O_BINARY`	binary file

`fopen()` also opens a file, but uses different arguments and returns a file pointer rather than a file number.

`creat()` uses the opening flags and file type flags described here when creating a file. The creation flags are defaulted by `creat()` to `O_CREAT` and `O_TRUNC`.

RETURN VALUE

`n`, where `n` is the file number of the file (0-14), if success; `EOF` if error.

SEE ALSO

`close()`, `creat()`, `read()`, `write()`, `lseek()`, `fopen()`

perror (unix)

print user's string and error number

SYNTAX

```
#include <unix.h>
void perror(str)
char *str;
```

DESCRIPTION

`perror()` prints out the user's string (`str`) on `stdout` and then "`Error number n`" where `n` is the last error number that occurred.

RETURN VALUE

`void`

SEE ALSO

`exit()`

place a point at location x,y

point (unix)

SYNTAX

```
#include <unix.h>
void point(x,y)
int x,y;
```

DESCRIPTION

`point(x,y)` sets `pixel(x,y)` on the Macintosh screen to black. The Macintosh coordinate system is defined with 0,0 at the top left of the screen, and 512,342 at the bottom right.

Note: do not confuse this function with the QuickDraw type `Point`.

SEE ALSO

`circle()`, `gotoxy()`, `label()`, `line()`, `move()`

printf (stdio)

print and format to stdout

SYNTAX

```
#include <stdio.h>
int printf(format {, args})
char *format;
```

DESCRIPTION

`printf()` is the easiest way to output a mixture of text and values of variables (character, numeric, and string) to `stdio`. Other versions of `printf()`, such as `fprintf()`, `sprintf()`, `cprintf()`, `vprintf()`, `vfprintf()`, `vsprintf()`, and `vcprintf()`, use the same argument conventions except as noted in their own descriptions. The versions differ in where the output is sent.

`printf()` takes a variable number of arguments. The first argument, `format`, is required. It is a string containing text and format specifiers. The format specifiers tell how to interpret and format the variable number of `args` that follow.

The `args` argument(s), written in braces above to indicate that it is optional and may be repeated, is a list of all the variables that this `printf()` statement will take data from. Make sure that there is one variable argument of the correct size for each format specifier in `format` requiring data. Arguments must also be of the right type. For example, if the format specifier specifies a string, the argument must be a `char*`. If the number of arguments, or the argument types, do not match the number and type of format specifiers, expect unpredictable results.

Format specifiers begin with the character `%` and include zero or more of the following conversion specification elements:

```
% [option flags] [field size] [.precision] [size] conversion
```

Some of these elements are optional, but if present, they must be specified in the order in which they are described below.

Option Flags (optional):

- Left adjust output in field, pad on right (default is to right justify).

0	Use zero (0) rather than space for the pad character.
+	Always produce a sign, either + or —.
space	Always produce either the — sign or a space.
#	Use a variant of the main conversion operation. (See the description of conversion operations below for details.)

Field Size Specification (optional):

The minimal field width, expressed as a decimal integer. `arg` will be printed in a field at least this wide. If `arg` is shorter than field, field will be padded on the side determined by the field adjusting flags. The pad character is normally a space. The pad character is set to zero if the field width is given with a leading 0. For example:

```
printf("%03 d", 2);
```

prints:

```
002
```

with width=3, zero padding. (The zero padding does not imply octal values.)

Precision Specification (optional):

The precision specification identifies the maximum number of characters to be printed from a string or the number of characters to be printed to the right of the decimal point for a long or double.

It consists of a period (.) followed by an optional decimal integer. If the integer is omitted, a precision of 0 is assumed. This is not the same as omitting the precision specification entirely.

An asterisk (*) can also follow the period; in this case, the appropriate `arg` (selected by its corresponding position in the argument list) is used to specify the precision. This argument must be an `int`.

Argument Size Specifications (optional):

`l`	argument is a `long`
`h`	argument is a halfword (a `short`)

Conversion Characters (required):

c	argument is a single character
d	argument printed in signed decimal
e, E	argument is printed in signed decimal floating point format ([-]d.dddeÂ±dd). One digit appears before the decimal point, and the precision specifies the number of digits to be printed after the decimal point. The only difference between e and E, is that in the latter case, the exponent is introduced by the letter E rather than the letter e.
f	argument is printed in signed decimal fixed point format ([-]ddd.dddd). The precision specifies the number of digits to be printed after the decimal point. If precision is missing, 6 digits are output; if precision is 0, no decimal point appears.
g, G	Use the f or e format, whichever is smaller. G will cause E to be used instead of e.
o	argument is printed in unsigned octal.
s	argument is a string. Print until null character or have filled field specified by precision.
u	argument printed in unsigned decimal
x	argument printed in unsigned hexadecimal, using the digits 0123456789abcdef
X	argument printed in unsigned hexadecimal, using the digits 0123456789ABCDEF
%	print a %, no argument used

EXAMPLES

```
/* print text string followed by a newline.
   It contains no format specifiers, so
   requires no arguments. */
printf("simple text\n");

/* print n = 3;  %d is the format specifier
   to print a decimal number  */
int n;
n = 3;
printf("n = %d", n);
```

```
/* print a date and time in the form:
   weekday, month, day, hour:minute */
char *weekday, *month;
int day, hour, min;
printf("%s, %s, %d, %.2d:%.2d", weekday, month,
   day, hour, min);
```

RETURN VALUE The number of characters printed.

SEE ALSO scanf(), cprintf(), sprintf(), vprintf()

PtoCstr (MacTraps)

Convert a Pascal String to a C String

SYNTAX

```
#include <pascal.h>
char *PtoCstr(s)
char *s;
```

DESCRIPTION

`PtoCstr()` converts a Pascal string (preceded with the escape sequence `"\p"` and a count of the length of the string) to a C string (terminated with a null character).

The string `s` is converted in place (that is, the original string is modified). However, this function also returns a pointer to the converted string.

There is a limit of 255 characters on the length of a Pascal string.

RETURN VALUE

A pointer to the converted string.

SEE ALSO

`CtoPstr()`

SYNTAX

```
#include <stdio.h>
int putc(c, stream)
char c;
FILE *stream;
```

DESCRIPTION

`putc()` adds `c` to `stream`. `putc()` is a macro definition for `fputc()`.

Other functions which put characters into a file are: `fputs()`, `fprintf()`, `vfprintf()`, `write()`, and `fwrite()`.

`putch()` puts a character onto the screen and `putchar()` puts a character onto `stdout`.

RETURN VALUE

The character argument if success; `EOF` if failure.

SEE ALSO

`fputc`, `fputs()`, `fprintf()`, `vfprintf()`, `write()`, `fwrite()`, `putchar`

putch
(stdio)

put a character on screen

SYNTAX

```
#include <stdio.h>
void putch(c)
char c;
```

DESCRIPTION

`putch()` puts a single character to the screen. Unlike `putc()` and `fputc()`, which put a character to a file, or `putchar()`, which puts a character into `stdout`, it is not possible for `putch()` to fail because the screen will always be there.

`putch()` returns nothing.

`cputs()`, `cprintf()`, and `vcprintf()` also send characters to the screen.

SEE ALSO

`putc()`, `fputc()`, `putchar()`, `cputs()`, `cprintf()`, `vcprintf()`

put a character to stdout

putchar (stdio)

SYNTAX

```
#include <stdio.h>
int putchar(c)
char c;
```

DESCRIPTION

`putchar()` is a macro which expands to `fputc(c, stdout)`. `putchar()` sends a single character to `stdout`. The other functions which send data to `stdout` are `puts()` and `printf()`.

RETURN VALUE

The character argument if success; `EOF` if error.

SEE ALSO

`cputc()`, `putch()`, `fputc()`, `puts()`, `printf()`

puts
(stdio)

put a string to stdout

SYNTAX

```
#include <stdio.h>
int puts(s)
char *s;
```

DESCRIPTION

`puts()` puts the characters from string `s` to `stdout` until it reaches a null byte (`'\0'`) signalling the end of the string. The `'\0'` is discarded, and a carriage return (`'\r'`) is written to `stdout`. There are some differences between `puts()` and `cputs()` (put string to console) and `fputs()` (put string to file) so check the descriptions before using them. `printf()` and `putchar()` also send output to `stdout`.

RETURN VALUE

0 if success; `EOF` if failure.

SEE ALSO

`fputs()`, `cputs()`, `printf()`, `putchar()`

write word as 2 byte number to stream

putw (unix)

SYNTAX

```
#include <unix.h>
int putw(word,stream)
int word;
FILE *stream;
```

DESCRIPTION

`putw()` writes `word` as high byte, then low byte to the given output `stream`. Words written to a file are in memory image format.

RETURN VALUE

The `word` if success; `EOF` if error.

SEE ALSO

`getw()`

qksort (unix)

Sort a table in place

SYNTAX

```
#include <unix.h>
void qksort (nel, compare, swap)
int nel;
int (*compare)();
void (*swap)();
```

DESCRIPTION

`qksort` is an implementation of the quicksort algorithm. It sorts a table of data.

`nel` is the number of elements in the table.

`compare` is the name of a user-supplied comparison function.

`swap` is the name of a user-supplied swap function.

```
int compare(i,j)
int i, j;
```

The `compare` function is passed two indices, starting at zero, into the table, the location of which is assumed to be known by `compare`. `compare` should return an integer less than, equal to, or greater than zero, depending on whether the first indexed item is less than, equal to, or greater than the second.

```
void swap(i,j)
int i, j;
```

The `swap` function is passed two indices, starting at zero, into the table, the location of which is assumed to be known by `swap`. `swap` should interchange the two table items.

SEE ALSO

`qsort()`

Sort a table in place

qsort (unix)

SYNTAX

```
#include <unix.h>
int qsort (base, nel, size, compare)
char *base;
int nel;
int size;
int (*compare)();
```

DESCRIPTION

`qsort` is an implementation of the quicksort algorithm. It sorts a table of data.

`base` points to the first byte of the table.

`nel` is the number of elements in the table.

`size` is the size of each element in the table. NOTE: to provide an efficient implementation for the internal swap procedure, it is assumed that if size is 2 or 4, then the table pointed to by `base` is aligned on a word boundary. If it is not, qsort will cause an *address error.*

`compare` is the name of a user-supplied comparison function.

```
int compare(p1, p2)
type  *p1, *p2;
```

The `compare` function is passed two pointers to items of *`type`*. `compare` should return an integer less than, equal to, or greater than zero, depending on whether the first item is less than, equal to, or greater than the second.

RETURN VALUE

0 if `qsort()` completes successfully, -1 if an error occurred. An error occurs if qsort cannot obtain `size` bytes for its internal swap function temporary buffer when `size` is a value other than 1, 2, or 4.

SEE ALSO

`qksort()`

rbrk
(storageu)

reset memory break point

SYNTAX

```
#include <storage.h>
void rbrk();
```

DESCRIPTION

`rbrk()` resets the memory break point to the starting position.

NOTE

The UNIX library procedures `getmem()`, `getml()`, `rlsmem()`, `rlsml()`, `allmem()`, `bldmem()`, `rbrk()`, `rstmem()`, `lsbrk()`, and `sbrk()` are not recommended for casual use. These procedures attempt to provide low-overhead UNIX-like memory allocation, and are provided for compatibility purposes.

SEE ALSO

`allmem()`, `bldmem()`, `rstmem()`, `lsbrk()`, `sbrk()`, `getmem()`, `rlsmem()`, `getml()`, `rlsml()`, `rstmem()`

SYNTAX

```
#include <unix.h>
int read(fn, buffer, count)
int fn;
char *buffer;
unsigned int count;
```

DESCRIPTION

`read()` tries to read `count` bytes from file number `fn` into the array pointed to by `buffer`. If `read()` reaches the EOF before it has read `count` bytes, it stops and returns the number of bytes it did read. Make sure the buffer is large enough to hold the data. An efficient size for reading is BUFSIZ, the size of the I/O buffers. `fread()` also reads data in from a file but uses different arguments, including accessing the file by its file pointer.

RETURN VALUE

The number of bytes read; `EOF` if error.

SEE ALSO

`fread()`, `write()`, `open()`

realloc (storageu)

change the size of an allocated region of memory

SYNTAX

```
#include <storage.h>
char *realloc(ptr, size)
char *ptr;
unsigned int size;
```

DESCRIPTION

`realloc()` changes the size of the memory region pointed to by `ptr` while preserving its contents.

If the new size is larger than the old, new space is added at the end. If necessary, the contents are copied to another region of memory. If the new size is smaller than the old, space is taken away at the end of the old region and the contents of that portion are lost.

`relalloc()` can be used if you need a larger `size` specification (`long` rather than `int`).

RETURN VALUE

A pointer to the (new) region in memory if success; `NULL` if error.

SEE ALSO

`relalloc()`

change the size of an allocated region of memory

relalloc (storageu)

SYNTAX

```
#include <storage.h>
char *relalloc(ptr, size)
char *ptr;
unsigned long size;
```

DESCRIPTION

`relalloc` changes the size of the memory region pointed to by `ptr` while preserving its contents. If necessary, the contents are copied to another region of memory.

This function is identical to `realloc` except that it allows for a larger `size` specification (`long` rather than `int`).

RETURN VALUE

A pointer to the (new) region in memory if success; `NULL` if error.

SEE ALSO

`realloc()`

remove (unix)

remove filename from directory

SYNTAX

```
#include <unix.h>
int remove(filename)
char *filename;
```

DESCRIPTION

`remove()` deletes the file from the directory in its entirety. It performs the same function as `unlink` in this implementation.

RETURN VALUE

0 if success; `-1` if failure.

SEE ALSO

`unlink()`, `rename()`, `creat()`

rename a file

rename (unix)

SYNTAX

```
#include <unix.h>
int rename(old, new)
char *old, *new;
```

DESCRIPTION

`rename()` changes the name of a file in the directory from `old` to `new`. `old` and `new` are C-style null terminated strings.

RETURN VALUE

`0` if success; `-1` if failure.

SEE ALSO

`creat()`, `remove()`, `rename()`

repmem (unix)

copy pattern into memory

SYNTAX

```
#include <unix.h>
void repmem(s,v,lv,nv)
char *s,*v;
int lv,nv;
```

DESCRIPTION

repmem copies the pattern pointed to by v into memory at s, nv times. lv is the length of the pattern string v.

This means that nv * lv bytes are moved into v.

SEE ALSO

allmem(), bldmem(), getmem(), movmem(), rlsmem(), rstmem(), setmem(),

move I/O position to start of file

rewind (stdio)

SYNTAX

```
#include <stdio.h>
void rewind(stream)
FILE *stream;
```

DESCRIPTION

`rewind()` sets the I/O position back to the start of the file. The I/O position is moved automatically during I/O, as well as explicitly by `lseek()` and `fseek()`.

A call to `fseek(fd, 0L, 0)` is the same as a call to `rewind(fd)`.

EXAMPLE

```
rewind(fd);
fseek(fd, 0L, 0);     /* same as rewind */
```

SEE ALSO

`fseek()`, `lseek()`

rlsmem (storageu)

release memory allocated by getmem()

SYNTAX

```
#include <storage.h>
int rlsmem(cp, n)
char *cp;
unsigned n;
```

DESCRIPTION

`rlsmem()` releases a block of memory allocated by `getmem()`. `cp` is a pointer to the first character in the block, and `n` is an unsigned length of the block.

`rlsmem()` is analogous to `free()`, which releases a block of memory allocated by `calloc()`. Note the difference between `rlsmem()` and `free()`: `rlsmem()` needs to be told how many bytes to release, while `free()` releases the whole block automatically. For that reason, `rlsmem()` is both more flexible (can release part of a block) and more difficult to use than `free()`.

`rlsml()` releases memory allocated by `getml()` or `getmem()`.

The introduction to this section contains a list matching allocating functions with deallocating functions.

NOTE

The UNIX library procedures `getmem()`, `getml()`, `rlsmem()`, `rlsml()`, `allmem()`, `bldmem()`, `rbrk()`, `rstmem()`, `lsbrk()`, and `sbrk()` are not recommended for casual use. These procedures attempt to provide low-overhead UNIX-like memory allocation, and are provided for compatibility purposes.

RETURN VALUE

0 if success; -1 if failure.

SEE ALSO

`getmem()`, `rlsml()`, `getml()`

SYNTAX

```
#include <storage.h>
int rlsml(cp, n)
char *cp;
unsigned long n;
```

DESCRIPTION

`rlsml()` releases a block of memory got by `getml()`. `cp` is a pointer to the first character in the block, and `n` is an unsigned length of the block.

`rlsml()` is analogous to `free()`, which releases a block of memory allocated by `calloc()`. Note the difference between `rlsml()` and `free()`: `rlsml()` needs to be told how many bytes to release, while `free()` releases the whole block automatically. For that reason, `rlsml()` is both more flexible (can release part of a block) and more difficult to use than `free()`.

`rlsmem()` releases memory allocated by `getmem()`.

The introduction to this section contains a list matching allocating functions with deallocating functions.

NOTE

The UNIX library procedures `getmem()`, `getml()`, `rlsmem()`, `rlsml()`, `allmem()`, `bldmem()`, `rbrk()`, `rstmem()`, `lsbrk()`, and `sbrk()` are not recommended for casual use. These procedures attempt to provide low-overhead UNIX-like memory allocation, and are provided for compatibility purposes.

RETURN VALUE

`0` if success; `-1` if failure.

SEE ALSO

`getml()`, `rlsmem()`

rstmem (storageu)

reset memory allocated by getmem()

SYNTAX

```
#include <storage.h>
void rstmem();
```

DESCRIPTION

`rstmem()` releases the memory allocated by `getmem()` calls made after calls to `allmem()` or `bldmem()`.

NOTE

The UNIX library procedures `getmem()`, `getml()`, `rlsmem()`, `rlsml()`, `allmem()`, `bldmem()`, `rbrk()`, `rstmem()`, `lsbrk()`, and `sbrk()` are not recommended for casual use. These procedures attempt to provide low-overhead UNIX-like memory allocation, and are provided for compatibility purposes.

SEE ALSO

`allmem()`, `bldmem()`, `rbrk()`, `sbrk()`, `lsbrk()`, `getmem()`, `getml()`, `rlsmem()`, `rlsml()`

SYNTAX

```
#include <storage.h>
char *sbrk(n)
unsigned n;
```

DESCRIPTION

`sbrk()` dynamically allocates a block of `n` bytes. The block is not zeroed. A similar function, `calloc()`, allocates a block and zeroes it before returning.

`rbrk()` is used to release memory allocated by `sbrk()`, `lsbrk()`, `getmem()` or `getml()`.

NOTE

The UNIX library procedures `getmem()`, `getml()`, `rlsmem()`, `rlsml()`, `allmem()`, `bldmem()`, `rbrk()`, `rstmem()`, `lsbrk()`, and `sbrk()` are not recommended for casual use. These procedures attempt to provide low-overhead UNIX-like memory allocation, and are provided for compatibility purposes.

RETURN VALUE

A pointer to the newly allocated block; `NULL` if error.

SEE ALSO

`lsbrk()`, `getmem()`, `getml()`

scanf (stdio)

scan and format input from stdin

SYNTAX

```
#include <stdio.h>
int scanf(format {, args})
char *format;
```

DESCRIPTION

`scanf()` reads characters in from `stdin` and formats them into data. That data is stored in variables.

For example, if `scanf()` is expecting an integer and reads in the characters `"371"`, it converts it from a character string into the integer value 371, and stores it in a variable pointed to by one of `args`.

`scanf()` can read in different kinds of data in a single call, so it provides an extrememly useful and flexible way to get input.

`cscanf()`, `fscanf()`, and `sscanf()` use the same argument conventions as `scanf()` except when specified otherwise in their own descriptions. This description of `format` and `args` applies to all `scanf()` functions.

The first argument to `scanf()` is `format`, a character string telling `scanf` how to format the data it reads. `format` consists of text and conversion specifications, just as in `printf()`. The conversion specifications tell `scanf()` how to format the data that is being read in. Each conversion specification is matched to a field of input. Depending on the conversion specification, that field of input is either fixed length or goes until the next white space character in the input string.

Ordinary text in `format` is a place-holder which is matched against the input text between input fields. If the input text does not match the text in `format`, the `scanf()` is aborted. Format syntax and the conversion specifications are described below.

The remaining arguments to `scanf` are `args`, a variable number of pointers to variables. There should be one `arg` for each conversion specification in `format` (except for the conversion specifications `%*` and `%%`, which are explained below).

CAUTION

The most common mistake when using `scanf()` is to use the variables themselves rather than the pointers to the variables. `scanf()` needs to know the address of the variables it will use to store data, not their present values. Failing to use addresses to store the values will overwrite random memory. See the examples. If `scanf()` runs out of arguments before reaching the end of s, arbitrary values from the stack are used as the address of the arguments and overwrite random memory.

White space characters in format are ignored. White space characters are spaces, tabs, or newlines. Include them to make the format string more readable. White space characters in the input stream separate input fields.

Non-white space characters (except for `%`) are matched to the input characters. They must match perfectly or else the `scanf()` is aborted.

`%` indicates the start of a conversion specification.

Conversion specifications begin with `%` and contain in order:

```
% [*] [width] [size] type
```

Assignment Suppression (optional):

If the first character in the conversion specification is an asterisk (*), `scanf()` reads the input field until the next white space character. It does not assign the input field to any variable.

Field Width (optional):

`n` where n is an integer. `scanf` reads up to `n` characters for this field. If the field is longer than `n` characters, it skips the rest of the field, and continues with the next part of `format` at the next white space in the input.

Size Specifications (optional):

`l` `arg` variable for this field is a `long`.

`h` `arg` variable for this field is a half-word (a `short`).

Conversion Characters (terminate conversion specification):

`c`	interpret input as a single character; `arg` should be a `char*`; white space characters are not skipped in this case. `%1s` will read in the next non-white space character.
`d`	interpret input as a decimal integer; `arg` should be an `int*`.
`e,E,f, g,G`	interpret input as a signed decimal floating point number. These operations are all identical.
`o`	interpret input as an octal integer; `arg` should be an `int*`; input need not start with `0`.
`s`	interpret argument as a character string; `arg` should be a `char*`; a `'\0'` will be added to `arg` after the string is read in.
`u`	interpret input as unsigned decimal; `arg` should be an `int*`.
`x,X`	interpret input as a hexadecimal value; `arg` should be an `int*`; input need not start with `0x`. These two specifiers are identical.
`[`	scan the input for the set of characters in the format string following the `[` up to a terminating `]`. You can think of the characters enclosed in the brackets as a "scanset." `scanf()` will take as input the maximum string of characters that match the scanset. When a circumflex (`^`) appears as the first character in the scanset (e.g. `[^0123456789]`), it acts as a complement operator, and causes all characters not in the set to be matched. A range of consecutive characters can be represented by separating the first and last members of the range with a hyphen (for example `[A-Za-z]` will match any alphabetic characters, or `[0-9]` will match the set of digits. If the first character in a range is not lexically smaller than the last, then the characters in the scanset stand for themselves. For example, `[9-1]` will match the three characters "`9-1`." To include a right bracket (`]`) in the scanset, it must be the first character in the set. For example, `[]]` will match a single right bracket.

`arg` should be a `char[]*` large enough to hold the data and the terminating `'\0'`, which will be added automatically.

`%` input should be the character `%`; input not stored in a variable in `arg`. No `arg` is used in the process.

EXAMPLES

```
int n;
scanf("%d ", &n); /* correct: &n is a pointer to n */
scanf("%d ", n);  /* incorrect: n is the variable itself */
                  /* this will trash random memory */
```

RETURN VALUE Number of items successfully input.

SEE ALSO `cscanf()`, `fscanf()`, `sscanf()`, `getchar()`, `gets()`

setbuf (unix)

attach private buffer to file

SYNTAX

```
#include <unix.h>
void setbuf(fp,buf)
FILE *fp;
char *buf;
```

DESCRIPTION

`setbuf()` attaches a private buffer to the file whose file pointer is `fp`. The size of the buffer is `BUFSIZ`. If `buf` is `NULL` then this is the same as calling `setnbuf()`.

SEE ALSO

`setnbuf()`

set console screen to echo keyboard input

Set_Echo (stdio)

SYNTAX

```
#include <stdio.h>
void Set_Echo(state);
int state;
```

DESCRIPTION

`Set_Echo()` sets the console to echo input from the keyboard to the screen. If `state` is nonzero, then echo is enabled; if `state` is zero, echo is disabled.

Echo is enabled by default.

SEE ALSO

`Set_Tab()`, `getch()`, `getche()`

setjmp
longjmp
(unix)

perform non-local goto

SYNTAX

```
#include <unix.h>
int setjmp(env)
jmp_buf env;

void longjmp (env, val);
jmp_buf env;
int val;
```

DESCRIPTION

`setjmp()` is used with `longjmp()` to provide a non-local goto function, useful for dealing with errors and interrupts in low-level subroutines.

`setjmp()` saves the stack environment in `env` and returns the value 0. `longjmp` restores the environment saved by the last call to `setjmp()`. Program execution continues as if `setjmp` had just returned `val`.

RETURN VALUE

`setjmp()` returns 0 unless `longjmp()` is called with `val=0`, in which case `setjmp()` returns 1.

initialize a block of memory

setmem (storageu)

SYNTAX

```
#include <unix.h>
void setmem(addr, n, c)
char *addr;
unsigned n;
char c;
```

DESCRIPTION

`setmem()` initializes a block of memory starting at `addr`, for `n` bytes, to the value `c`.

SEE ALSO

`movmem()`, `repmem()`

setnbuf (unix)

set a file to be unbuffered

SYNTAX

```
#include <unix.h>
void setnbuf(fp)
FILE *fp;
```

DESCRIPTION

`setnbuf` sets a file as unbuffered; that is, it undoes the effect of a previous call to `setbuf()`.

SEE ALSO

`setbuf()`

setpid (unix)

SYNTAX

```
#include <unix.h>
int setpid(val)
int val;
```

DESCRIPTION

`setpid()` sets the process id number to `val`.

This call is provided solely for UNIX compatibility. It has no real function.

RETURN VALUE

`val`, the argument passed in.

SEE ALSO

`getpid()`

Set_Tab (stdio)

set tab width of console screen

SYNTAX

```
#include <stdio.h>
void Set_Tab(tab_width)
int tab_width;
```

DESCRIPTION

`Set_Tab()` sets the space between tabs on the console screen to `tab_width`. `Set_Tab()` does not change the internal representation of a tab, only the effect of outputting a tab on the console.

If `tab_width` is less than 1, it is set to 1.

SEE ALSO

`Set_Echo()`

set user's id number to val

setuid (unix)

SYNTAX

```
#include <unix.h>
int setuid(val)
int val;
```

DESCRIPTION

`setuid` sets the user's id number to `val`.

This routine is provided for UNIX compatibility. It has no real function.

RETURN VALUE

`val`, the argument passed in.

SEE ALSO

`getuid()`

sleep (unix)

pause seconds, then continue

SYNTAX

```
#include <unix.h>
void sleep(secs)
int secs;
```

DESCRIPTION

`sleep()` waits `secs` and then continues.

format and make output into a string

sprintf (stdio)

SYNTAX

```
#include <stdio.h>
int sprintf(s, format {, args})
char *s, *format;
```

DESCRIPTION

`sprintf()` is yet another function based on `printf()`. Just as `printf()` formats text and data into a string of characters and then prints it on `stdout`, `sprintf()` formats text and data into a character array. That character array is left containing a string of output. Therefore, the primary action of `sprintf()` is a side effect: changing the contents of string `s`.

The first argument, `s`, is a pointer to a character array large enough to hold the anticipated result. `format` and the optional `args` are interpreted in the usual way by `printf()` (see `printf()` for details) with the result put in `s`.

`sscanf()` is the inverse of `sprintf()`. It scans and formats input from a string.

Care must be taken to ensure that the buffer pointed to by `s` is large enough to hold the converted string. Also note that the string will be null-terminated automatically by `sprintf`.

RETURN VALUE

The number of characters put into the buffer pointed to by `s`, including null terminator, if successful; `EOF` if an error occurs.

SEE ALSO

`printf()`, `sscanf()`

sscanf (stdio)

scan and format input from a string

SYNTAX

```
#include <stdio.h>
int sscanf(s, format [, args])
char *s, *format;
```

DESCRIPTION

`sscanf()`, like the other `scanf()` functions (`scanf()`, `cscanf()`, and `fscanf()`), does a scan and format of a series of characters. The special feature of `sscanf()` is that it takes that series of characters from its first argument, a string `s`. `s` should terminate in a null (`'\0'`) character. `format` and `args` are interpreted as usual; see `scanf()` for details. Similarly, `sprintf()` does a format and print to a character array in memory.

CAUTION

The most common mistake when using `sscanf()` is to use the variables themselves rather than the pointers to the variables. `sscanf()` needs to know the address of the variables it will use to store data, not their present values. Failing to use addresses to store the values will overwrite random memory. See the examples in `scanf()`. If `sscanf()` runs out of arguments before reaching the end of `s`, arbitrary values from the stack are used as the address of the arguments and overwrite random memory.

RETURN VALUE

Number of items successfully input.

SEE ALSO

`scanf()`, `fscanf()`, `cscanf()`, `sprintf()`

count number of characters in string s that are in set

stcarg (unix_strings)

SYNTAX

```
#include <strings.h>
int stcarg(s, set)
char *s, *set;
```

DESCRIPTION

`stcarg()` counts the number of characters in string `s` that are in `set`.

`stcarg()` will ignore strings enclosed in double quotes or literals enclosed in quotes inside `s` (see example). A backslash character in `s` is also ignored.

Consequently, a single quote, a double quote, or a backslash contained in `*set` will never match a character in `s`.

`stcarg()` returns the count of matched characters.

EXAMPLES

```
stcarg("abcde","12345")    /* returns 0 */
stcarg("aabcd","aeiou")    /* returns 2 */
stcarg("aeiou","aabcd")    /* returns 1 */
stcarg("abc\\d","abc\\d") /* returns 4 (the backslash
                              character is ignored) */
stcarg("'s'tp","stp")      /* returns 2 (the s is
                              ignored because it
                              is quoted) */
```

RETURN VALUE

The number of characters in `s` that are in `set` with the restrictions described above.

stccpy (strings)

copy first n characters from s2 to s1

SYNTAX

```
#include <strings.h>
int stccpy(s1, s2, n)
char *s1, *s2;
int n;
```

DESCRIPTION

`stccpy()` copies `n` characters from `s2` to `s1`. `stccpy()` will also stop if it copies a null character (`'\0'`) from `s2` to `s1`. The area pointed to by `s1` must be large enough to hold `n` characters. A `'\0'` is appended to `s1` only if less than `n` characters could be copied from `s2` before reaching the end of `s2`.

The character pointers are incremented by `stccpy()`. Therefore, the copy will overlap what was copied before if (`s2 < s1 < (s2 + n)`).

If `n` is negative or 0, `stccpy()` does not copy any characters and returns 0.

`strcpy()` and `strncpy()` also copy one string to another, but with some small differences.

The destination string is always null terminated.

EXAMPLES

```
A = "ABCD" ;  B = "EFGH IJ" ;
stccpy(A,B);
     /* A = "EFGH"  */
```

RETURN VALUE

The number of characters copied if `n` is positive (including null terminator). 0 if `n` is negative or zero.

SEE ALSO

`strcpy()`, `strncpy()`

SYNTAX

```
#include <unix.h>
int stcd_i(s,r)
char *s;
int *r;
```

DESCRIPTION

`stcd_i()` converts an ASCII string `s` to an integer. The result is placed in `r` and the number of characters scanned is returned. Scanning stops when an invalid character is present. Valid characters are `0-9`, `−`, and `+`.

`stcd_i()` is similar to `atoi()`.

RETURN VALUE

The number of characters scanned.

SEE ALSO

`stch_i()`, `stci_d()`, `stcu_d()`, `atoi()`

stch_i (unix)

convert an ASCII hex value to an integer

SYNTAX

```
#include <unix.h>
int stch_i(s,r)
char *s;
int *r;
```

DESCRIPTION

`stch_i()` converts an ASCII hex string `s` to an integer. The result is placed in `r` and the number of characters scanned is returned. Scanning stops when an invalid character is present. Valid characters are `0-9`, `a-f`, `A-F`, `-`, and `+`.

RETURN VALUE

The number of characters scanned.

SEE ALSO

`stcd_i()`, `stci_d()`, `stcu_d()`

convert integer to ASCII string, place string in buffer

stci_d (unix)

SYNTAX

```
#include <unix.h>
int stci_d(out,in,outlen)
char *out;
int in;
int outlen;
```

DESCRIPTION

`stci_d()` converts the integer `in` to an ASCII string and places the string in the buffer pointed to by `out`. A maximum of `outlen` characters are placed into the buffer.

RETURN VALUE

Number of characters placed in buffer `out`, not including null terminator.

SEE ALSO

`stch_i()`, `stcd_i()` `stcu_d()`

stcis (strings)

return position of first character in s not in set

SYNTAX

```
#include <strings.h>
int stcis(s, set)
char *s, *set;
```

DESCRIPTION

`stcis()` returns the position of the first character in string `s` that is not in `set`. The position of the first character in `s` is 0, the second character is position 1, etc.

If all characters in `s` are in `set`, then `stcis()` returns the position of the null character which terminates string `s`. In effect, this is the length of the string, not counting the null at the end.

`set` is implemented as a string, but it is logically used as a group of characters with order unimportant.

`strspn()` is a synonym for `stcis()`.

RETURN VALUE

The position of the first character in `s` that is not in `set`.

SEE ALSO

`strspn()`, `stcisn()`, `strcspn()`, `strpbrk()`, `strrpbrk()`

SYNTAX

```
#include <strings.h>
int stcisn(s, set)
char *s, *set;
```

DESCRIPTION

`stcisn()` finds the first character in string `s` which is a member of string `set`. `stcisn()` returns an `int` which is the place of the first letter in `s` to be in the set (see examples). The first character in the string `s` is in place 0, the second is in place 1, etc.

If no character in `s` is a member of the set, then `stcisn()` returns the place of the null character which terminates string `s`. This is also the length of the string as returned by `strlen()`.

The second argument is called a set for semantic reasons; however, it is implemented simply as a string.

`strcspn()` is a synonym for `stcisn()`.

EXAMPLE

```
stcisn("word","aeiou")    /* returns 1 (the place
                             of the o in word) */
stcisn("igloo","aeiou")   /* returns 0 (the place
                             of the i in igloo) */
stcisn("GREEN","aeiou")   /* returns 5 (the place
                             of the '\0' in GREEN) */
```

RETURN VALUE

The place of the first character in `s` which is in `set`, or the place of the null which terminates `s` if no character in `s` is in `set`.

SEE ALSO

`stcis()`, `strcspn()`, `stpbrk()`, `strpbrk()`

stclen (strings)

return the length of string s

SYNTAX

```
#include <strings.h>
int stclen(s)
char *s;
```

DESCRIPTION

`stclen()` simply returns the number of non-null characters in string s.

`strlen()` is a synonym for `stclen()`.

EXAMPLE

```
stclen("the cat")  /* returns 7 */
stclen("")         /* returns 0 */
```

RETURN VALUE

The number of non-null characters in s.

SEE ALSO

`strlen()`

search for pattern anywhere in string

stcpm (unix_strings)

SYNTAX

```
#include <strings.h>
int stcpm(string, pattern, q)
char *string, *pattern, *q;
```

DESCRIPTION

`stcpm()` calls `stcpma()` repetitively to search for `pattern` anywhere in `string`.

`stcpm()` allows wildcard matches in `pattern`, using the following pattern-matching rules:

`?` matches any single character.

`c*` (where `c` can be replaced by any character) matches any number of consecutive `c` characters (including 0).

`c+` (where `c` can be replaced by any character) matches one or more consecutive `c` characters (but not 0 characters).

To match a wildcard character literally, precede it by a backslash (`\`) character. For example, `\*` in `pattern` will only match a single `*` in `string`, `\?` will only match a `?`, and `\+` will only match a `+`. (When you are writing the backslash character in a string for the compiler, you have to precede the backslash character itself with a backslash. `\\` is turned by the compiler into a single backslash character. In input from the keyboard or a file, a single backslash is sufficient.)

Because of wildcards and backslashes, the number of characters matched is not necessarily the same in `pattern` as in `string`. (For example `pattern = "ab+cd"`, `string = "abbbbbcd"` is a perfect match even though `pattern` has 5 characters and `string` has 8.)

NOTE

`stcpm()` returns two values, one directly and one by side effect. Its return value is an `int` which is the length of the matched pattern if a match is made, and 0 if no match is made.

*q, the third argument, is used to return a pointer to the first occurrence of pattern in string (or is left unchanged if no match is made).

EXAMPLES

```
char *substring;
stcpm("bxde", "a*b+?d", &substring)

/* returns 3 and sets substring to the address of the start
   of the source string. Note that a* matches zero or more
   instances of "a".      */

stcpm ("yybxde", "a*b+?d", &substring)

/* returns 3 and sets substring to the address
   of 'b' in the source string.    */
```

RETURN VALUE

0 if pattern is not in string; n where n is the length of pattern if pattern is in string; by side effect, the third argument *q is set to p where p is a pointer to the first pattern in string. *q is left unchanged if no match occurs.

SEE ALSO

stcpma()

stcpma (unix_strings)

SYNTAX

```
#include <strings.h>
int stcpma(string, pattern)
char *string, *pattern;
```

DESCRIPTION

`stcpma()` compares `pattern` to the beginning of `string`. If they match exactly, `stcpma()` returns the number of characters in the match. If they partially match, `stcpma()` returns the number of characters until they differ. If they don't match at all, `stcpma()` returns 0.

`stcpma()` supports the following wildcard matching syntax in `pattern`:

`?` matches any single character.

`c*` (where `c` can be replaced by any character) matches any number of consecutive `c` characters (including 0).

`c+` (where `c` can be replaced by any character) matches one or more consecutive `c` characters (but not 0 characters).

To match a wildcard character literally, precede it by a backslash (`\`) character. For example, `\*` in `pattern` will only match a single `*` in `string`, `\?` will only match a `?`, and `\+` will only match a `+`. (When you are writing the backslash character in a string for the compiler, you have to precede the backslash character itself with a backslash. `\\` is turned by the compiler into a single backslash character. In input from the keyboard or a file, a single backslash is sufficient.)

Because of wildcards and backslashes, the number of characters matched is not necessarily the same in `pattern` as in `string`. (For example `pattern = "ab+cd"`, `string = "abbbbbcd"` is a perfect match even though `pattern` has 5 characters and `string` has 8.) The number returned is the number of characters in `string` that were matched.

EXAMPLES

```
stcpma("bxde", "a*b+?d");      /* returns 3 */
stcpma("yybxde", "a*b+?d");    /* returns 0 */
```

RETURN VALUE

The number of characters at the beginning of `string` that match `pattern`; `0` if no characters match.

SEE ALSO

`stcpm()`

SYNTAX

```
#include <unix.h>
int stcu_d(out,in,outlen)
char *out;
unsigned in;
int outlen;
```

DESCRIPTION

`stcu_d()` converts the unsigned integer `in` to an ASCII string and places the string in the buffer pointed to by `out`. A maximum of `outlen` characters are placed into the buffer.

RETURN VALUE

The number of characters placed in buffer `out`, not including null terminator.

SEE ALSO

`stcd_i()`, `stch_i()`, `stci_d()`

Stdio_config
(stdio)

SYNTAX

```
#include <stdio.h>
void Stdio_config(font, size, style, mode)
int font;
int size;
int style;
int mode;
```

DESCRIPTION

`Stdio_config()` configures standard I/O to handle Macintosh font information. If you are want to change the default I/O configuration, you must call this function before doing any screen I/O.

`font` is the number of a Macintosh font. `size` is the point size. `style` is the font style (e.g. bold, underscore, etc.) `mode` is the transfer mode (`srcCopy`, `srcOR`, etc.).

See `FontMgr.h` for symbolic definitions of font names. See `QuickDraw.h` for symbolic definitions of style and transfer mode.

The default values if `Stdio_config()` is not called are `monaco, 9, 0, 1`, which corresponds to Monaco font, 9 point, normal style, source bits ORed.

return pointer to first blank space character in ptr

SYNTAX

```
#include <strings.h>
char *stpblk(ptr)
char *ptr;
```

DESCRIPTION

`stpblk()` finds the first white space character in `ptr`.

`stpblk()` does not stop at a `'\0'` character marking the end of the string, but will continue on through memory until a white space character is found.

RETURN VALUE

`p` where `p` is a `char*` to the first whitespace character in `ptr`.

SEE ALSO

`isspace()`

stpbrk (strings)

return pointer to first character in s that is in set

SYNTAX

```
#include <strings.h>
char *stpbrk(s, set)
char *s, *set;
```

DESCRIPTION

`stpbrk()` finds the first character in string `s` that is in `set`. `stpbrk()` returns a pointer to that first character in `s` (not to the matched character in `set`). It returns `NULL` if no characters match.

Note that `set` is implemented as a `char*`, but it is used logically as a set of characters.

`strpbrk()` is a synonym for `stpbrk()`.

`strcspn()` also finds the first character of a string `s` in a set, but it returns an `int` which is the position of the character rather than a pointer to the actual character. `strrpbrk()` returns a pointer to the last character in `s` which appears in `set`.

RETURN VALUE

A pointer to the first character in `s` that is in `set`; `NULL` if no characters in `s` are in `set`.

SEE ALSO

`strpbrk()`, `strcspn()`, `strrpbrk()`

SYNTAX

```
#include <strings.h>
char *stpchr(s, c)
char *s, c;
```

DESCRIPTION

`stpchr()` returns a character pointer to the first occurrence of character `c` in string `s`. If `c` is not in `s`, `stpchr()` returns `NULL`. `stpchr()` will not search past the null character (`'\0'`) which ends string `s`. If `c` is equal to `'\0'`, it is considered to match the `'\0'` which terminates string `s` and a pointer to that position is returned.

`strchr()` is a synonym for `stpchr()`.

`strpos()` also finds a character `c` in a string `s`, but `strpos()` returns an `int` which is the position of `c` in `s` rather than a pointer to that character.

`strrpos()` and `strrchr()` return the position of the last `c` in `s` and a pointer to the last `c` in `s`, respectively.

RETURN VALUE

`NULL` if `c` is not in `s`; otherwise, a pointer to the first occurrence of `c` in `s`.

SEE ALSO

`strpos()`, `strrpos()`, `strrchr()`, `strchr()`

stpcpy (strings)

copy s2 into s1

SYNTAX

```
#include <strings.h>
char *stpcpy(s1, s2)
char *s1, *s2;
```

DESCRIPTION

`stpcpy()` copies the entire string `s2` into `s1`. `stpcpy()` uses `(strlen(s2) + 1)` as the third argument to `strncpy()` to ensure that `s2` is copied in its entirety.

See `strncpy()` and `strlen()` for details of their functions.

Be sure that the character array pointed to by `s1` is sufficiently large.

`strcpy()` is synonymous with `stpcpy()`.

RETURN VALUE

A pointer to `s1`.

SEE ALSO

`strncpy()`, `stccpy()`, `strcpy()`

SYNTAX

```
#include <strings.h>
char *stpsym(input, output, symlen)
char *input, *output;
int symlen;
```

DESCRIPTION

`stpsym()` finds the first symbol in `input`. A symbol begins with a letter (upper or lower case) and ends at the first character which is not a letter or digit. `stpsym()` copies up to the first `(symlen - 1)` characters of the first symbol from `input` to `output`.

The `symlen` argument should be the length of output available, as it keeps `stpsym()` from copying too much into `output` and destroying the memory that follows. A null character (`'\0'`) is added to `output` after the last character from the symbol.

`strpsym()` also returns a pointer to the first character in `input` after the symbol. If the first character in `input` does not begin a symbol, `stpsym()` does not copy into `output` and returns a pointer to the first character in `input`.

Note that if the first symbol in `input` is larger than `(strlen - 1)` characters, `stpsym()` returns a pointer to the next character in that symbol as the beginning of the next symbol. Thus, if the character at the return pointer is a letter or digit, you know that the preceding symbol has been broken in two.

The key thing to remember is that `stpsym()` returns two things: a copy of the symbol in `output` and a pointer to the character after the symbol in `input`.

EXAMPLES

```
stpsym("  abc_1 ", sym, sizeof(sym))
/* sets sym = "abc" */
```

RETURN VALUE

Directly: `p` where `p` is a pointer into `string` to the character after the symbol. `p` points to the beginning of `s` if it does not begin a symbol.

By side effect, `output` has a copy of the first symbol from `string` (up to `(strlen - 1)` characters) terminated with a `'\0'`.

SEE ALSO

`stptok()`

SYNTAX

```
#include <strings.h>
char *stptok(input, output, toklen, brkset)
char *input, *output, *brkset;
int toklen;
```

DESCRIPTION

`stptok()` parses the next token in `input`. A token is defined as a sequence of characters not containing any characters from `brkset`, the fourth argument to `stptok()`.

`stptok()` has two effects. First, `stptok()` copies up to the first `(toklen - 1)` characters from the first token in `input` to `output` and appends a null character (`'\0'`) to `output`. Second, `stptok()` returns a pointer to the first character after the token.

There are two special cases to be aware of. If the first character in `input` is not part of a token (and therefore a member of the `brkset`), `stptok()` puts a `'\0'` at the start of `output` and returns a pointer to that first character in `input`. If the first token in `input` is longer than `(toklen - 1)` characters, the pointer returned is `&input[toklen]`, even though that character might seem to be logically part of the first token.

`stptok()` is parallel to `stpsym()` in every way. `stpsym()` parses the first symbol (letters and digits) of an input stream.

EXAMPLES

```
stptok("abc to 2", sym, sizeof (sym), "5")

/* returns a pointer to the first character
   in the source and sets sym to "abc" */
```

RETURN VALUE

A pointer to the next character in `input` after the first token is parsed; by side effect, `output` contains a copy of the first `(toklen - 1)` characters of the first token in `input`. `output` is always terminated with a `'\0'`.

SEE ALSO

`stpsym()`

strcat
(strings)

append s2 to s1

SYNTAX

```
#include <strings.h>
char *strcat(s1, s2)
char *s1, *s2;
```

DESCRIPTION

`strcat()` calls `strncat()` to append the entire string `s2` to the string pointed to by `s1`.

`strcat()` uses `(strlen(s2) + 1)` as the third argument to `strncat()` to ensure that `s2` is appended in its entirety. Be sure that there is enough space in the character array after the end of the string `s1` to allow `s2` to be appended.

The null character which ended string `s1` is overwritten by the first character from `s2`.

RETURN VALUE

`s1` with `s2` appended to it.

SEE ALSO

`strlen()`, `strncat()`

SYNTAX

```
#include <strings.h>
char *strchr(s, c)
char *s, c;
```

DESCRIPTION

`strchr()` returns a character pointer to the first occurrence of character `c` in string `s`. If `c` is not in `s`, `strchr()` returns `NULL`. `strchr()` will not search past the null character (`'\0'`) which ends string `s`.

If `c` is equal to `'\0'`, it is considered to match the `'\0'` which terminates string `s` and a pointer to that position is returned.

`stpchr()` is a synonym for `strchr()`.

`strpos()` also finds a character `c` in a string `s`, but `strpos()` returns an `int` which is the position of `c` in `s` rather than a pointer to that character.

`strrpos()` and `strrchr()` return the position of the `last c in s` and a pointer to the `last c in s respectively`.

RETURN VALUE

`NULL` if `c` is not in `s`; otherwise, a pointer to first occurrence of `c` in `s`.

SEE ALSO

`strpos()`, `strrpos()`, `strrchr()`, `stpchr()`

strcmp (strings)

lexically compare s1 and s2

SYNTAX

```
#include <strings.h>
int strcmp(s1, s2)
char *s1, *s2;
```

DESCRIPTION

`strcmp()` calls `strncmp()` to compare the length of `s1` to the length of `s2`. `strcmp()` uses `(strlen(s1) + 1)` as the third argument to `strncmp()` to ensure that `s1` is compared to `s2` in its entirety.

Note that if `s1` is a substring of `s2`, then `strcmp()` will return a number greater than 0 because the last characters compared will be the terminating null (`'\0'`) character of `s1` against some character in `s2`.

See `strncmp()` and `strlen()` for details of their functions.

`stscmp()` is a synonym for `strcmp()`.

RETURN VALUE

0 if string `s1` is the same as string `s2`; positive if `s1 > s2`; negative if `s1 < s2`.

SEE ALSO

`strncmp()`, `strlen()`, `stscmp()`

copy s2 into s1

strcpy (strings)

SYNTAX

```
#include <strings.h>
char *strcpy(s1, s2)
char *s1, *s2;
```

DESCRIPTION

`strcpy()` calls `strncpy()` to copy the entire string `s2` into `s1`. `strcpy()` uses `(strlen(s2) + 1)` as the third argument to `strncpy()` to ensure that `s2` is copied in its entirety.

See `strncpy()` and `strlen()` for details of their functions.

Be sure that the character array pointed to by `s1` is large enough to hold the whole of `s2`.

`stpcpy()` is a synonym for `strcpy()`.

RETURN VALUE

A pointer to `s1`.

SEE ALSO

`strncpy()`, `stccpy()`, `stpcpy()`

strcspn (strings)

return position of first character in s that is in set

SYNTAX

```
#include <strings.h>
int strcspn(s, set)
char *s, *set;
```

DESCRIPTION

`strcspn()` finds the first character in string `s` which is a member of string `set`. `strcspn()` returns an `int` which is the place of the first letter in `s` to be in the set (see examples). The first character in the string `s` is in place 0, the second is in place 1, etc.

If no character in `s` is a member of the set, then `strcspn()` returns the place of the null character which terminates string `s`. This is also the length of the string as returned by `strlen()`.

The second argument is called `set` for semantic reasons, however it is implemented simply as a string.

`stcisn()` is a synonym for `strcspn()`.

EXAMPLES

```
strcspn("word", "aeiou")  /* returns 1 (the place
                             of the o in word) */
strcspn("igloo", "aeiou") /* returns 0 (the place
                             of the i in igloo) */
strcspn("GREEN", "aeiou") /* returns 5 (the place
                             of the '\0' in GREEN) */
```

RETURN VALUE

The place of the first character in `s` which is in `set`. If no character in `s` is in `set`, then the return value is the position of the null that terminates `s`.

SEE ALSO

`stcis()`, `stcisn()`, `strspn()`, `stpbrk()`, `strpbrk()`

return the length of string s

strlen (strings)

SYNTAX

```
#include <strings.h>
int strlen(s)
char *s;
```

DESCRIPTION

`strlen()` simply returns the number of non-null characters in string s.

`stclen()` is a synonym for `strlen()`.

RETURN VALUE

The number of non-null characters in s.

EXAMPLES

```
strlen("the cat")  /* returns 7 */
strlen("")         /* returns 0 */
```

SEE ALSO

`stclen()`

strncat (strings)

append up to n characters from s2 to s1

SYNTAX

```
#include <strings.h>
char *strncat(s1, s2, n)
char *s1, *s2;
int n;
```

DESCRIPTION

`strncat()` appends `s2` to `s1` until it has appended `n` characters or it has reached the end of `s2`. The characters are added to `s1` starting at the first null character (`'\0'`) of `s1`. That `'\0'` is replaced by the first character of `s2`.

If `n` characters are appended without reaching the end of `s2`, `strncat()` adds a terminating null character (`'\0'`).

If `n` is negative or zero, `strncat()` returns `s1` unchanged.

RETURN VALUE

`s1` where `s1` is the first argument with the contents of `s2` appended to it.

SEE ALSO

`strcat()`

compare up to first n characters of s1 and s2

strncmp (strings)

SYNTAX

```
#include <strings.h>
int strncmp(s1, s2, n)
char *s1, *s2;
int n;
```

DESCRIPTION

`strncmp()` compares `s1` and `s2` up to a limit of `n`, and returns an integer indicating whether `s1` is equal to, less than, or greater than `s2`.

`strcmp()` follows the same rules except that it compares until it reaches the end of one or both strings.

RETURN VALUE

`0` if string `s1` is the same as string `s2` in the first `n` characters; positive if `s1 > s2` in the first `n` characters; negative if `s2 > s1` in the first `n` characters.

SEE ALSO

`strcmp(), stscmp()`

strncpy (strings)

copy up to n characters from s2 to s1

SYNTAX

```
#include <strings.h>
char *strncpy(s1, s2, n)
char *s1, *s2;
int n;
```

DESCRIPTION

`strncpy()` copies characters from `s2` to `s1` until either `n` characters have been copied or it has reached the null character (`'\0'`) which marks the end of `s2`. If the end of `s2` is reached first, (`n - strlen(s2)`) null characters are appended as padding. No null characters are appended if `n` characters from `s2` are added without reaching a `'\0'` in `s2`.

Thus, to reiterate, the first `n` characters of `s1` will be written over, either by the first `n` characters from `s2`, or if `s2` ends early, `s2` plus enough null character padding.

If `n` is negative or zero, `s1` is returned unchanged.

`stccpy()` also copies `n` characters from `s2` to `s1`.

`stccpy()` handles null padding differently, and returns the number of characters copied instead of a pointer to `s1`.

`strcpy()` copies `s2`, and nothing but `s2`, to `s1`.

RETURN VALUE

A pointer to `s1`.

SEE ALSO

`stccpy()`, `strcpy()`

return pointer to first character in s that is in set

strpbrk (strings)

SYNTAX

```
#include <strings.h>
char *strpbrk(s, set)
char *s, *set;
```

DESCRIPTION

`strpbrk()` finds the first character in string `s` that is in `set`. `strpbrk()` returns a pointer to that first character in `s` (not to the matched character in `set`). It returns `NULL` if no characters match.

`stpbrk()` is a synonym for `strpbrk()`.

`strcspn()` also finds the first character of a string `s` in a set, but it returns an `int` which is the position of the character rather than a pointer to the actual character. `strrpbrk()` returns a pointer to the last character in `s` which appears in `set`.

RETURN VALUE

A pointer to the first character in `s` that is in `set`. `NULL` if no characters in `s` are in `set`.

SEE ALSO

`stpbrk()`, `strcspn()`, `strrpbrk()`

strpos (strings)

return position of the first character c in string s

SYNTAX

```
#include <strings.h>
int strpos(s, c)
char *s, c;
```

DESCRIPTION

`strpos()` searches `s` from beginning to end for the character `c`. `strpos()` returns the first position of `c` in `s`, with the first character in `s` being at position 0. `strpos()` returns -1 if `c` is not in `s`.

`strchr()` also finds a character `c` in a string `s`, but returns a pointer to the first such character.

`strrpos()` and `strrchr()` return the position of the last `c` and a pointer to the last `c` in `s` respectively.

RETURN VALUE

The first position of `c` in `s`; `-1` if `c` is not in `s` or if `c` is `'\0'`.

SEE ALSO

`strchr()`, `strrchr()`, `strrpos()`

return pointer to last occurrence of c in s

strrchr (strings)

SYNTAX

```
#include <strings.h>
char *strrchr(s, c)
char *s, c;
```

DESCRIPTION

`strrchr()` searches string `s` from beginning to end for character `c`. `strrchr()` returns a pointer to the last occurrence of `c` that it finds.

If `c` is the null character (`'\0'`), `strrchr()` returns a pointer to the null character which terminates `s`. `strrchr()` returns `NULL` if `c` is not in `s`. `strrpos()` also finds the last occurrence of a character `c` in a string `s`, but returns its position as an integer rather than a pointer to it.

`strpos()` and `strchr()` return the position of the first `c` and a pointer to the first `c` in `s` respectively.

RETURN VALUE

A pointer to the last occurrence of `c` in `s`; `NULL` if `c` is not in `s`.

SEE ALSO

`strrchr()`, `strpos()`, `strchr()`

strrpbrk (strings)

return pointer to last character in string s that is in set

SYNTAX

```
#include <strings.h>
char *strrpbrk(s, set)
char *s, *set;
```

DESCRIPTION

`strrpbrk()` finds the last character in `s` that is in `set`, and returns a pointer to that character. The pointer is to the character in `s`, not the character it matches in `set`.

Note that while `set` is implemented as a character string, the order of characters in it does not matter. A lower case character does not match its equivalent upper case character.

`strpbrk()` returns the pointer to the first character in `s` that is in `set`.

RETURN VALUE

A character pointer to the last character in `s` that is in `set`; `NULL` if no character in `s` is in `set`.

SEE ALSO

`strpbrk()`, `strspn()`, `stcis()`, `stcisn()`

return position of the last character c in string s

strrpos (strings)

SYNTAX

```
#include <strings.h>
int strrpos(s, c)
char *s, c;
```

DESCRIPTION

`strrpos()` searches string `s` from beginning to end for character `c`. `strrpos()` returns the last position of `c`, with the first character of `s` being at position 0. `strrpos()` returns -1 if `c` is not in `s`.

If `c` is the null character (`'\0'`), `strrpos()` returns the position of the `'\0'` which terminates string `s`.

`strrchr()` also finds a character `c` within a string `s`, but returns a pointer to the last such character.

`strpos()` and `strchr()` return the position of the first `c` in `s` and a `pointer` to the `first c in s` respectively.

RETURN VALUE

The position of the last `c` in `s`; `-1` if `c` is not in `s`.

SEE ALSO

`strrchr()`, `strpos()`, `strchr()`

strspn (strings)

return position of first character in s not in set

SYNTAX

```
#include <strings.h>
int strspn(s, set)
char *s, *set;
```

DESCRIPTION

`strspn()` returns the position of the first character in string `s` that is not in `set`. The position of the first character in `s` is 0, the second character is position 1, etc.

If all characters in `s` are in `set`, then `strspn()` returns the position of the null character which terminates string `s`. `set` is implemented as a string, but it is logically used as a set of characters with order unimportant.

`stcis()` is a synonym for `strspn()`.

RETURN VALUE

The position of the first character in `s` that is not in `set`.

SEE ALSO

`stcis()`, `stcisn()`, `strcspn()`, `strpbrk()`, `strrpbrk()`

lexically compare s1 and s2

stscmp (string)

SYNTAX

```
#include <strings.h>
int stscmp(s1, s2)
char *s1, *s2;
```

DESCRIPTION

`stscmp()` calls `strncmp()` to compare the entire length of `s1` to the length of `s2`. `stscmp()` uses `(strlen(s1) + 1)` as the third argument to `strncmp()` to ensure that `s1` is compared to `s2` in its entirity.

Note that if `s1` is a substring of `s2`, then `stscmp()` will return a number greater than 0 because the last characters compared will be the terminating null character (`'\0'`) of `s1` against some character in `s2`.

See `strncmp()` and `strlen()` for details of their functions.

`strcmp()` is a synonym for `stscmp()`.

RETURN VALUE

0 if string `s1` is the same as string `s2`; positive if `s1 > s2`; negative if `s1 < s2`.

SEE ALSO

`strncmp()`, `strlen()`, `strcmp()`

stspfp (unix)

parse filename pattern

SYNTAX

```
#include <unix.h>
int stspfp(p,n)
char *p;
int n[16];
```

DESCRIPTION

`stspfp` parses a filename pattern which consists of node names separated by colons. Each colon is replaced by a null byte and the beginning index of that node is placed in the index array.

For example, `:abc:de:fgh` has 3 nodes, and their indices are 1 for `abc`, 5 for `de` and 8 for `fgh`. The last entry in the array index is -1.

RETURN VALUE

0 if successful; -1 if too many nodes or other error.

tell (unix)

SYNTAX

```
#include <unix.h>
long tell(fildes)
int fildes;
```

DESCRIPTION

`tell` returns the position within the file. This is the UNIX equivalent to `ftell`, except you pass the file's number instead of the file descriptor. If an error occurs then EOF is returned.

RETURN VALUE

A `long` representing the position within file if success; `EOF` if error.

SEE ALSO

`ftell()`

time
(unix)

return the time in seconds

SYNTAX

```
#include <unix.h>
unsigned long time(tloc)
unsigned long *tloc;
```

DESCRIPTION

`time()` returns the time in seconds since Jan 1, 1970 00:00:00. If `tloc` is not NULL, then the `time` is placed in `tloc` also.

RETURN VALUE

The time in seconds since Jan 1, 1970, 00:00:00.

SEE ALSO

`ctime()`, `localtime()`

SYNTAX

```
#include <ctype.h>
char toascii(i)
char i;
```

DESCRIPTION

`toascii()` converts `i` into a character by returning its lower seven bits. The highest order bit becomes a 0. Thus, `toascii()` returns the character equivalent of integers between 0 and 255. This is not the same as the familiar `itoa()` function, which converts an integer into an ASCII string (eg: the integer 65 is converted into the character `'A'` by `toascii()`, but into the string `"65"` by `itoa()`).

WARNING: `toascii()` and `toint()` are not inverse functions.

RETURN VALUE

The character equivalent (low 7 bits) of the integer argument.

SEE ALSO

`toint()`, `itoa()`

toint
(stdio)

convert a hexadecimal character into an integer

SYNTAX

```
#include <stdio.h>
int toint(c)
char c;
```

DESCRIPTION

`toint()` returns the integer corresponding to the hexadecimal value of a character `c`. For example, `toint('2')` returns the `int` 2, `toint('a')` returns the `int` 10, `toint('F')` returns the `int` 15, `toint('r')` returns -1 to indicate an error. `toint()` accepts upper or lower case. Note that `toint()` is not the same as `atoi()` which converts a whole string into an `int` (as opposed to a character), nor is it the inverse of `toascii()`.

RETURN VALUE

-1 if error; the `int` that corresponds to the argument if success.

SEE ALSO

`toascii()`, `atoi()`

convert a character to lower case

tolower
_tolower
(stdio)

SYNTAX

```
#include <stdio.h>
int tolower(c)
char c;

int _tolower(c)
char c;
```

DESCRIPTION

`tolower()` returns the lower case equivalent of an upper case letter `c`. If `c` is not an upper case letter, then `c` is returned unchanged.

`_tolower()` is faster. However it returns a wrong value if its argument is not an upper case letter.

RETURN VALUE

The lower case equivalent of upper case character `C`, or if `c` is not an upper case letter, `c` itself.

SEE ALSO

`toupper()`, `_toupper()`

toupper
_toupper
(stdio)

convert a character to upper case

SYNTAX

```
#include <stdio.h>
int toupper(c)
char c;

int _toupper(c)
char c;
```

DESCRIPTION

`toupper()` returns the upper case equivalent of a lower case letter `c`. If `c` is not a lower case letter, `toupper()` returns it unchanged.

`_toupper()` is faster. However, it returns a wrong value if its argument is not a lower case letter.

RETURN VALUE

The upper case equivalent of lower case letter `c`; if `c` is not a lower case letter, `c` itself.

SEE ALSO

`tolower()`

return terminal id number

ttyn (unix)

SYNTAX

```
#include <unix.h>
int ttyn()
```

DESCRIPTION

`ttyn()` returns the terminal id number, which is 1. This routine is provided for UNIX compatibility, and performs no real function.

RETURN VALUE

The terminal id number, which is 1.

SEE ALSO

`setpid()`, `setuid()`, `getuid()`, `getpid()`

ungetc (stdio)

return character c to input stream buffer

SYNTAX

```
#include <stdio.h>
int ungetc(c, stream)
int c;
FILE *stream;
```

DESCRIPTION

`ungetc()` pushes a single character `c` back to the I/O stream. The next read from that stream will get that character first.

`ungetc(s)`, where `s` is an input stream from the keyboard, works by manipulating the `look_full` and `look_ahead` fields of the file descriptor (see `FILE`). When `s` is a file, the character is placed back into the buffer. If the character is different from the last character read and the buffer has been modified by a write, the new character will be written to the file.

RETURN VALUE

`c` if successful; `EOF` if not. Pushing back EOF places the value of EOF on the `stream` and returns the value of `EOF`.

SEE ALSO

`getc()`

push character back to keyboard input stream

ungetch (stdio)

SYNTAX

```
#include <stdio.h>
char ungetch(c)
char c;
```

DESCRIPTION

`ungetch()` pushes the character `c` back to the keyboard input stream.

RETURN VALUE

The character `c`, if the character could be pushed back; otherwise, `EOF`.

SEE ALSO

`getch()`, `ungetc()`

unlink (unix)

unlink a file from the directory

SYNTAX

```
#include <unix.h>
int unlink(filename)
char *filename;
```

DESCRIPTION

`unlink()` removes filename from the directory and deletes the file. The library function `remove()` also removes a file from the directory.

RETURN VALUE

`0` if success; `-1` if failure.

SEE ALSO

`creat()`, `remove()`, `rename()`

variable format and print to stdout

vprintf
vcprintf
vfprintf
vsprintf
(stdio)

SYNTAX

```
#include <stdio.h>
int vprintf(format, array)
char *format;
char *array;
int vcprintf(format, array)
char *format;
char *array;
int vfprintf(format, array)
char *format;
char *array;
int vsprintf(format, array)
char *format;
char *array;
```

DESCRIPTION

The `vprintf` family of routines is entirely analogous in function to the similarly named routines lacking the `v` prefix. That is, `vprintf` performs the same function as `printf`, `vfprintf` as `fprintf`, and so on.

The difference between each pair of routines is that instead of an argument list corresponding to the format specifiers in `format`, the `vprintf` functions take a single array of arguments.

In building this array, you must be sure that each item in it corresponds to the appropriate format specifier in `format`.

RETURN VALUE

The number of characters written, not counting the terminating null character (`'\0'`).

SEE ALSO

`printf()`, `cprintf()`, `fprintf()`, `sprintf()`

write (unix)

write a block of bytes to a file

SYNTAX

```
#include <unix.h>
int write(fn, buffer, nbytes)
int fn;
char *buffer;
unsigned nbytes;
```

DESCRIPTION

`write()` copies `nbytes` from `buffer` to the file with file descriptor number `fn`. The most efficient choice for `nbytes` is usually `BUFSIZ`.

`fwrite()` is another library function to write a block of data to a file, with some differences. `write()` uses the file number (returned if `open()` is used to open the file) while `fwrite()` uses the file descriptor pointer (returned if `fopen()` is used to open the file.) In addition, `write()` has three arguments, while `fwrite()` has four. Therefore, `write()` and `fwrite()` are NOT synonymous.

RETURN VALUE

The number of bytes copied, if success; `-1` if error.

SEE ALSO

`fwrite()`

14
C Calling Sequences for Macintosh Toolbox and OS Routines

This section lists the C calling sequences for the Macintosh Toolbox and Operating System Routines.

Routine groupings correspond to those in *Inside Macintosh* from which they are taken. Sections are in the following order:

Binary-Decimal Conversion Package	14-2
Control Manager	14-3
Desk Manager	14-6
Device Manager	14-7
Dialog Manager	14-8
Disk Driver	14-11
Disk Initialization Package	14-12
OS Event Manager	14-13
Toolbox Event Manager	14-14
File Manager	14-15
Font Manager	14-21
International Utilities Package	14-22
Memory Manager	14-23
Menu Manager	14-26
Package Manager	14-29
QuickDraw	14-30
Resource Manager	14-41
Scrap Manager	14-44
Segment Loader	14-45
Serial Driver	14-46
Sound Driver	14-47
Standard File Package	14-48
System Error Handler	14-49
TextEdit	14-50
Operating System Utilities	14-53
Toolbox Utilities	14-55
Vertical Retrace Manager	14-58
Window Manager	14-59

Binary-Decimal Conversion Package

```
void StringToNum(theString, theNum)
Str255 theString;
long *theNum;

void NumToString(theNum, theString)
long theNum;
Str255  theString;
```

```
ControlHandle NewControl(curWindow, boundsRect, title, visible, value, min,
                         max, contrlProc, refCon)
WindowPtr curWindow;
Rect *boundsRect;
Str255 title;
Boolean visible;
int value;
int min;
int max;
int contrlProc;
long refCon;

void DisposeControl(theControl)
ControlHandle theControl;

void KillControls(theWindow)
WindowPtr theWindow;

void MoveControl(theControl, h, v)
ControlHandle theControl;
int h, v;

void SizeControl(theControl, w, h)
ControlHandle theControl;
int w, h;

void DragControl(theControl, startPt, bounds, slopRect, axis)
ControlHandle theControl;
Point startPt;
Rect *bounds;
Rect *slopRect;
int axis;

void ShowControl(theControl)
ControlHandle theControl;

void HideControl(theControl)
ControlHandle theControl;

void SetCTitle(theControl, title)
ControlHandle theControl;
Str255 title;

void GetCTitle(theControl, title)
ControlHandle theControl;
Str255 *title;
```

```
void HiliteControl(theControl, hiliteState)
ControlHandle theControl;
int hiliteState;

void SetCRefCon(theControl, data)
ControlHandle theControl;
long data;

long GetCRefCon(theControl)
ControlHandle theControl;

void SetCtlValue(theControl, theValue)
ControlHandle theControl;
int theValue;

int GetCtlValue(theControl)
ControlHandle theControl;

int GetCtlMin(theControl)
ControlHandle theControl;

int GetCtlMax(theControl)
ControlHandle theControl;

void SetCtlMin(theControl, theValue)
ControlHandle theControl;
int theValue;

void SetCtlMax(theControl, theValue)
ControlHandle theControl;
int theValue;

ProcPtr GetCtlAction(theControl)
ControlHandle theControl;

void SetCtlAction(theControl, newProc)
ControlHandle theControl;
ProcPtr newProc;

int TestControl(theControl, thePt)
ControlHandle theControl;
Point thePt;

int TrackControl(theControl, thePt, actionProc)
ControlHandle theControl;
Point thePt;
ProcPtr actionProc;

int FindControl(thePoint, theWindow, theControl)
Point thePoint;
WindowPtr theWindow;
```

```
ControlHandle *theControl;

void DrawControls(theWindow)
WindowPtr theWindow;

ControlHandle GetNewControl(controlID, owner)
int controlID;
WindowPtr owner;
```

Desk Manager

```
Boolean SystemEvent(myEvent)
EventRecord *myEvent;

void SystemClick(theEvent, theWindow)
EventRecord *theEvent;
WindowPtr theWindow;

void SystemTask()

void SystemMenu(menuResult)
long menuResult;

Boolean SystemEdit(editCode)
int editCode;

int OpenDeskAcc(theAcc)
Str255 theAcc;

void CloseDeskAcc(refNum)
int refNum;
```

Device Manager

```
OsErr OpenDriver(name, drvrRefNum)
Str255 name;
int *drvrRefNum;

OsErr CloseDriver(refNum)
int refNum;

DCtlHandle GetDCtlEntry(refNum)
int refNum;
```

```
void InitDialogs(resumeProc)
ProcPtr resumeProc;

DialogPtr GetNewDialog(dialogID, wStorage, behind)
int dialogID;
Ptr wStorage;
WindowPtr behind;

DialogPtr NewDialog(wStorage, boundsRect, title, visible, theProc, behind,
                    goAwayFlag, refCon, itmLstHndl)
Ptr wStorage;
Rect *boundsRect;
Str255 title;
Boolean visible;
int theProc;
WindowPtr behind;
Boolean goAwayFlag;
long refCon;
Handle itmLstHndl;

Boolean IsDialogEvent(event)
EventRecord *event;

Boolean DialogSelect(event, theDialog, itemHit)
EventRecord *event;
DialogPtr *theDialog;
int *itemHit;

void ModalDialog(filterProc, itemHit)
ProcPtr filterProc;
int *itemHit;

void DrawDialog(dialog)
DialogPtr dialog;

void CloseDialog(dialog)
DialogPtr dialog;

void DisposDialog(dialog)
DialogPtr dialog;

int Alert(alertID, filterProc)
int alertID;
ProcPtr filterProc;

int StopAlert(alertID, filterProc)
int alertID;
```

```
ProcPtr filterProc;

int NoteAlert(alertID, filterProc)
int alertID;
ProcPtr filterProc;

int CautionAlert(alertID, filterProc)
int alertID;
ProcPtr filterProc;

void CouldAlert(alertID)
int alertID;

void FreeAlert(alertID)
int alertID;

void CouldDialog(DlgID)
int DlgID;

void FreeDialog(DlgID)
int DlgID;

void ParamText(cite0, cite1, cite2, cite3)
Str255 cite0, cite1, cite2, cite3;

void ErrorSound(sound)
ProcPtr sound;

void GetDItem(dialog, itemNo, kind, item, box)
DialogPtr dialog;
int itemNo;
int *kind;
Handle *item;
Rect *box;

void SetDItem(dialog, itemNo, kind, item, box)
DialogPtr dialog;
int itemNo;
int kind;
Handle item;
Rect *box;

void SetIText(item, text)
Handle item;
Str255 text;

void GetIText(item, text)
Handle item;
Str255  text;

void SelIText(dialog, itemNo, startSel, endSel)
```

```
DialogPtr dialog;
int itemNo;
int startSel, endSel;

/* routines designed only for use in Pascal (or C!) */
int GetAlrtStage()

/* returns aCount */
void ResetAlrtStage()

void DlgCut(dialog)
DialogPtr dialog;

void DlgPaste(dialog)
DialogPtr dialog;

void DlgCopy(dialog)
DialogPtr dialog;

void DlgDelete(dialog)
DialogPtr dialog;

void SetDAFont(fontNum)
int fontNum;
```

Disk Driver

```
OSErr DiskEject(drvnum)
int drvnum;

OSErr SetTagBuffer(buffPtr)
Ptr buffPtr;

OSErr DriveStatus(drvNum, status)
int drvNum;
DrvSts *status;
```

Disk Initialization Package

```
void DILoad()

void DIUnload()

int DIBadMount(where, evtMessage)
Point where;
long evtMessage;

OsErr DIFormat(drvNum)
int drvNum;

OsErr DIVerify(drvNum)
int drvNum;

OsErr DIZero(drvNum, volName)
int drvNum;
Str255 volName;
```

OS Event Manager

```
OsErr PostEvent(eventNum, eventMsg)
int eventNum;
long eventMsg;

void FlushEvents(whichMask, stopMask)
int whichMask, stopMask;

void SetEventMask(theMask)
int theMask;

Boolean OSEventAvail(mask, theEvent)
int mask;
EventRecord *theEvent;

Boolean GetOSEvent(mask, theEvent)
int mask;
EventRecord *theEvent;

QHdrPtr GetEvQHdr()
```

```
Boolean EventAvail(mask, theEvent)
int mask;
EventRecord *theEvent;

Boolean GetNextEvent(mask, theEvent)
int mask;
EventRecord *theEvent;

Boolean StillDown()

Boolean WaitMouseUp()

void GetMouse(pt)
Point *pt;

long TickCount()

Boolean Button()

void GetKeys(k)
keyMap *k;

long GetDblTime()

long GetCaretTime()
```

```
OsErr PBOpen(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBClose(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBRead(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBWrite(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBControl(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBStatus(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBKillIO(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBGetVInfo(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBGetVol(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBSetVol(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBFlushVol(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBCreate(paramBlock, aSync)
```

```
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBDelete(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBOpenRF(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBRename(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBGetFInfo(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBSetFInfo(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBSetFLock(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBRstFLock(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBSetFVers(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBAllocate(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBGetEOF(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBSetEOF(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBGetFPos(paramBlock, aSync)
ParmBlkPtr paramBlock;
```

```
Boolean aSync;

OsErr PBSetFPos(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBFlushFile(paramBlock, aSync)
ParmBlkPtr paramBlock;
Boolean aSync;

OsErr PBMountVol(paramBlock)
ParmBlkPtr paramBlock;

OsErr PBUnmountVol(paramBlock)
ParmBlkPtr paramBlock;

OsErr PBEject(paramBlock)
ParmBlkPtr paramBlock;

OsErr PBOffLine(paramBlock)
ParmBlkPtr paramBlock;

void AddDrive(drvrRefNum, drvNum, QEl)
int drvrRefNum;
int drvNum;
DrvQElPtr QEl;

OsErr FSOpen(fileName, vRefNum, refNum)
Str255 fileName;
int vRefNum;
int *refNum;

OsErr FSClose(refNum)
int refNum;

OsErr FSRead(refNum, count, buffPtr)
int refNum;
long *count;
Ptr buffPtr;

OsErr FSWrite(refNum, count, buffPtr)
int refNum;
long *count;
Ptr buffPtr;

OsErr Control(refNum, csCode, csParamPtr)
int refNum;
int csCode;
Ptr csParamPtr;
```

```
OsErr Status(refNum, csCode, csParamPtr)
int refNum;
int csCode;
Ptr csParamPtr;

OsErr KillIO(refNum)
int refNum;

/* -volume level calls- */
OsErr GetVInfo(drvNum, volName, vRefNum, FreeBytes)
int drvNum;
StringPtr volName;
int *vRefNum;
long *FreeBytes;

OsErr GetFInfo(fileName, vRefNum, FndrInfo)
Str255 fileName;
int vRefNum;
FInfo *FndrInfo;

OsErr GetVol(volName, vRefNum)
StringPtr volName;
int *vRefNum;

OsErr SetVol(volName, vRefNum)
StringPtr volName;
int vRefNum;

OsErr UnmountVol(volName, vRefNum)
StringPtr volName;
int vRefNum;

OsErr Eject(volName, vRefNum)
StringPtr volName;
int vRefNum;

OsErr FlushVol(volName, vRefNum)
StringPtr volName;
int vRefNum;

/* -file level calls for unopened files- */
OsErr Create(fileName, vRefNum, creator, fileType)
Str255 fileName;
int vRefNum;
OSType creator;
OSType fileType;

OsErr FSDelete(fileName, vRefNum)
```

```
Str255 fileName;
int vRefNum;

OsErr OpenRF(fileName, vRefNum, refNum)
Str255 fileName;
int vRefNum;
int *refNum;

OsErr Rename(oldName, vRefNum, newName)
Str255 oldName;
int vRefNum;
Str255 newName;

OsErr SetFInfo(fileName, vRefNum, FndrInfo)
Str255 fileName;
int vRefNum;
FInfo *FndrInfo;

OsErr SetFLock(fileName, vRefNum)
Str255 fileName;
int vRefNum;

OsErr RstFLock(fileName, vRefNum)
Str255 fileName;
int vRefNum;

/* -file level calls for opened files- */
OsErr Allocate(refNum, count)
int refNum;
long *count;

OsErr GetEOF(refNum, LogEOF)
int refNum;
long *LogEOF;

OsErr SetEOF(refNum, LogEOF)
int refNum;
long LogEOF;

OsErr GetFPos(refNum, filePos)
int refNum;
long *filePos;

OsErr SetFPos(refNum, posMode, posOff)
int refNum;
int posMode;
long posOff;
```

```
OsErr GetVRefNum(fileRefNum, vRefNum)
int fileRefNum;
int *vRefNum;

/* queue routines - part of Macintosh core Utility routines /*
void FInitQueue()

QHdrPtr GetFSQHdr()

QHdrPtr GetDrvQHdr()

QHdrPtr GetVCBQHdr()
```

```
void InitFonts()

void GetFontName(familyID, theName)
int familyID;
Str255  theName;

void GetFNum(theName, familyID)
Str255 theName;
int *familyID;

void SetFontLock(lockFlag)
Boolean lockFlag;

FMOutPtr FMSwapFont(inRec)
FMInput *inRec;

Boolean RealFont(famID, size)
int famID;
int size;
```

```
Handle IUGetIntl(theID)
int theID;

void IUSetIntl(refNum, theID, intlParam)
int refNum;
int theID;
Handle intlParam;

void IUDateString(dateTime, longFlag, result)
long dateTime;
DateForm longFlag;
Str255 *result;

void IUDatePString(dateTime, longFlag, result, intlParam)
long dateTime;
DateForm longFlag;
Str255 *result;
Handle intlParam;

void IUTimeString(dateTime, wantSeconds, result)
long dateTime;
Boolean wantSeconds;
Str255 *result;

void IUTimePString(dateTime, wantSeconds, result, intlParam)
long dateTime;
Boolean wantSeconds;
Str255 *result;
Handle intlParam;

Boolean IUMetric()

int IUCompString(aStr, bStr)
Str255 aStr, bStr;

int IUEqualString(aStr, bStr)
Str255 aStr, bStr;

int IUMagString(aPtr, bPtr, aLen, bLen)
Ptr aPtr, bPtr;
int aLen, bLen;

int IUMagIDString(aPtr, bPtr, aLen, bLen)
Ptr aPtr, bPtr;
int aLen, bLen;
```

```
void SetApplBase(startPtr)
Ptr startPtr;

void InitApplZone()

void InitZone(pgrowZone, cmoreMasters, limitPtr, startPtr)
ProcPtr pgrowZone;
int cmoreMasters;
Ptr limitPtr, startPtr;

THz GetZone()

void SetZone(hz)
THz hz;

THz ApplicZone()

THz SystemZone()

Size CompactMem(cbNeeded)
Size cbNeeded;

void PurgeMem(cbNeeded)
Size cbNeeded;

long FreeMem()

void ResrvMem(cbNeeded)
Size cbNeeded;

Size MaxMem(grow)
Size *grow;

Ptr TopMem()

void SetGrowZone(growZone)
ProcPtr growZone;

void SetApplLimit(zoneLimit)
Ptr zoneLimit;

Ptr GetApplLimit()

void MaxApplZone()

void MoveHHi(h)
Handle h;
```

```
Ptr NewPtr(byteCount)
Size byteCount;

void DisposPtr(p)
Ptr p;

Size GetPtrSize(p)
Ptr p;

void SetPtrSize(p, newSize)
Ptr p;
Size newSize;

THz PtrZone(p)
Ptr p;

Handle NewHandle(byteCount)
Size byteCount;

void DisposHandle(h)
Handle h;

Size GetHandleSize(h)
Handle h;

void SetHandleSize(h, newSize)
Handle h;
Size newSize;

THz HandleZone(h)
Handle h;

Handle RecoverHandle(p)
Ptr p;

void EmptyHandle(h)
Handle h;

void ReallocHandle(h, byteCount)
Handle h;
Size byteCount;

void HLock(h)
Handle h;

void HUnlock(h)
Handle h;

void HPurge(h)
Handle h;
```

```
void HNoPurge(h)
Handle h;

void MoreMasters()

void BlockMove(srcPtr, destPtr, byteCount)
Ptr srcPtr, destPtr;
Size byteCount;

OsErr MemError()


Handle GZSaveHnd()
```

Menu Manager

```
void InitMenus()

MenuHandle NewMenu(menuID, menuTitle)
int menuID;
Str255 menuTitle;

MenuHandle GetMenu(rsrcID)
int rsrcID;

void DisposeMenu(menu)
MenuHandle menu;

void AppendMenu(menu, data)
MenuHandle menu;
Str255 data;

void InsertMenu(menu, beforeId)
MenuHandle menu;
int beforeId;

void DeleteMenu(menuId)
int menuId;

void DrawMenuBar()

void ClearMenuBar()

Handle GetMenuBar()

Handle GetNewMBar(menuBarID)
int menuBarID;

void SetMenuBar(menuBar)
Handle menuBar;

long MenuSelect(startPt)
Point startPt;

long MenuKey(ch)
char ch;

void HiliteMenu(menuId)
int menuId;

void SetItem(menu, item, itemString)
MenuHandle menu;
int item;
Str255 itemString;
```

```
void GetItem(menu, item, itemString)
MenuHandle menu;
int item;
Str255  itemString;

void EnableItem(menu, item)
MenuHandle menu;
int item;

void DisableItem(menu, item)
MenuHandle menu;
int item;

void CheckItem(menu, item, checked)
MenuHandle menu;
int item;
Boolean checked;

void SetItemIcon(menu, item, iconNum)
MenuHandle menu;
int item;
Byte iconNum;

void GetItemIcon(menu, item, iconNum)
MenuHandle menu;
int item;
Byte *iconNum;

void SetItemStyle(menu, item, styleVal)
MenuHandle menu;
int item;
Style styleVal;

void GetItemStyle(menu, item, styleVal)
MenuHandle menu;
int item;
Style *styleVal;

void SetItemMark(menu, item, markChar)
MenuHandle menu;
int item;
char markChar;

void GetItemMark(menu, item, markChar)
MenuHandle menu;
int item;
char *markChar;

void SetMenuFlash(flashCount)
int flashCount;
```

```
void FlashMenuBar(menuID)
int menuID;

MenuHandle GetMHandle(menuID)
int menuID;

int CountMItems(menu)
MenuHandle menu;

void AddResMenu(menu, theType)
MenuHandle menu;
ResType theType;

void InsertResMenu(menu, theType, afterItem)
MenuHandle menu;
ResType theType;
int afterItem;

void CalcMenuSize(menu)
MenuHandle menu;
```

Package Manager

```
void InitAllPacks()

void InitPack(packID)
int packID;
```

```
/*  GrafPort Routines  */
void InitGraf(globalPtr)
Ptr globalPtr;

void OpenPort(port)
GrafPtr port;

void InitPort(port)
GrafPtr port;

void ClosePort(port)
GrafPtr port;

void SetPort(port)
GrafPtr port;

void GetPort(port)
GrafPtr *port;

void GrafDevice(device)
int device;

void SetPortBits(bm)
BitMap *bm;

void PortSize(width, height)
int width, height;

void MovePortTo(leftGlobal, topGlobal)
int leftGlobal, topGlobal;

void SetOrigin(h, v)
int h, v;

void SetClip(rgn)
RgnHandle rgn;

void GetClip(rgn)
RgnHandle rgn;

void ClipRect(r)
Rect *r;

void BackPat(pat)
Pattern pat;

/*  Cursor Routines  */
```

```
void InitCursor()

void SetCursor(crsr)
Cursor *crsr;

void HideCursor()

void ShowCursor()

void ObscureCursor()

/*  Line Routines  */
void HidePen()

void ShowPen()

void GetPen(pt)
Point *pt;

void GetPenState(pnState)
PenState *pnState;

void SetPenState(pnState)
PenState *pnState;

void PenSize(width, height)
int width, height;

void PenMode(mode)
int mode;

void PenPat(pat)
Pattern pat;

void PenNormal()

void MoveTo(h, v)
int h, v;

void Move(dh, dv)
int dh, dv;

void LineTo(h, v)
int h, v;

void Line(dh, dv)
int dh, dv;

/*  Text Routines  */
void TextFont(font)
int font;
```

```
void TextFace(face)
Style face;

void TextMode(mode)
int mode;

void TextSize(size)
int size;

void SpaceExtra(extra)
Fixed extra;

void DrawChar(ch)
char ch;

void DrawString(s)
Str255 s;

void DrawText(textBuf, firstByte, byteCount)
Ptr textBuf;
int firstByte, byteCount;

int CharWidth(ch)
char ch;

int StringWidth(s)
Str255 s;

int TextWidth(textBuf, firstByte, byteCount)
Ptr textBuf;
int firstByte, byteCount;

void GetFontInfo(info)
FontInfo *info;

/*  Point Calculations  */
void AddPt(src, dst)
Point src;
Point *dst;

void SubPt(src, dst)
Point src;
Point *dst;

void SetPt(pt, h, v)
Point *pt;
int h, v;

Boolean EqualPt(pt1, pt2)
Point pt1, pt2;
```

```
void ScalePt(pt, fromRect, toRect)
Point *pt;
Rect *fromRect, *toRect;

void MapPt(pt, fromRect, toRect)
Point *pt;
Rect *fromRect, *toRect;

void LocalToGlobal(pt)
Point *pt;

void GlobalToLocal(pt)
Point *pt;

/*  Rectangle Calculations  */
void SetRect(r, left, top, right, bottom)
Rect *r;
int left, top, right, bottom;

Boolean EqualRect(rect1, rect2)
Rect *rect1, *rect2;

Boolean EmptyRect(r)
Rect *r;

void OffsetRect(r, dh, dv)
Rect *r;
int dh, dv;

void MapRect(r, fromRect, toRect)
Rect *r;
Rect *fromRect, *toRect;

void InsetRect(r, dh, dv)
Rect *r;
int dh, dv;

Boolean SectRect(src1, src2, dstRect)
Rect *src1, *src2;
Rect *dstRect;

void UnionRect(src1, src2, dstRect)
Rect *src1, *src2;
Rect *dstRect;

Boolean PtInRect(pt, r)
Point pt;
Rect *r;

void Pt2Rect(pt1, pt2, dstRect)
```

```
Point pt1, pt2;
Rect *dstRect;

/*  Graphical Operations on Rectangles  */
void FrameRect(r)
Rect *r;

void PaintRect(r)
Rect *r;

void EraseRect(r)
Rect *r;

void InvertRect(r)
Rect *r;

void FillRect(r, pat)
Rect *r;
Pattern pat;

/*  RoundRect Routines  */
void FrameRoundRect(r, ovWd, ovHt)
Rect *r;
int ovWd, ovHt;

void PaintRoundRect(r, ovWd, ovHt)
Rect *r;
int ovWd, ovHt;

void EraseRoundRect(r, ovWd, ovHt)
Rect *r;
int ovWd, ovHt;

void InvertRoundRect(r, ovWd, ovHt)
Rect *r;
int ovWd, ovHt;

void FillRoundRect(r, ovWd, ovHt, pat)
Rect *r;
int ovWd, ovHt;
Pattern pat;

/*  Oval Routines  */
void FrameOval(r)
Rect *r;

void PaintOval(r)
Rect *r;

void EraseOval(r)
Rect *r;
```

```
void InvertOval(r)
Rect *r;

void FillOval(r, pat)
Rect *r;
Pattern pat;

/*  Arc Routines  */
void FrameArc(r, startAngle, arcAngle)
Rect *r;
int startAngle, arcAngle;

void PaintArc(r, startAngle, arcAngle)
Rect *r;
int startAngle, arcAngle;

void EraseArc(r, startAngle, arcAngle)
Rect *r;
int startAngle, arcAngle;

void InvertArc(r, startAngle, arcAngle)
Rect *r;
int startAngle, arcAngle;

void FillArc(r, startAngle, arcAngle, pat)
Rect *r;
int startAngle, arcAngle;
Pattern pat;

void PtToAngle(r, pt, angle)
Rect *r;
Point pt;
int *angle;

/*  Polygon Routines  */
PolyHandle OpenPoly()

void ClosePoly()

void KillPoly(poly)
PolyHandle poly;

void OffsetPoly(poly, dh, dv)
PolyHandle poly;
int dh, dv;

void MapPoly(poly, fromRect, toRect)
PolyHandle poly;
Rect *fromRect, *toRect;
```

```
void FramePoly(poly)
PolyHandle poly;

void PaintPoly(poly)
PolyHandle poly;

void ErasePoly(poly)
PolyHandle poly;

void InvertPoly(poly)
PolyHandle poly;

void FillPoly(poly, pat)
PolyHandle poly;
Pattern pat;

/*  Region Calculations  */
RgnHandle NewRgn()

void DisposeRgn(rgn)
RgnHandle rgn;

void CopyRgn(srcRgn, dstRgn)
RgnHandle srcRgn, dstRgn;

void SetEmptyRgn(rgn)
RgnHandle rgn;

void SetRectRgn(rgn, left, top, right, bottom)
RgnHandle rgn;
int left, top, right, bottom;

void RectRgn(rgn, r)
RgnHandle rgn;
Rect *r;

void OpenRgn()

void CloseRgn(dstRgn)
RgnHandle dstRgn;

void OffsetRgn(rgn, dh, dv)
RgnHandle rgn;
int dh, dv;

void MapRgn(rgn, fromRect, toRect)
RgnHandle rgn;
Rect *fromRect, *toRect;

void InsetRgn(rgn, dh, dv)
RgnHandle rgn;
```

```
int dh, dv;

void SectRgn(srcRgnA, srcRgnB, dstRgn)
RgnHandle srcRgnA, srcRgnB, dstRgn;

void UnionRgn(srcRgnA, srcRgnB, dstRgn)
RgnHandle srcRgnA, srcRgnB, dstRgn;

void DiffRgn(srcRgnA, srcRgnB, dstRgn)
RgnHandle srcRgnA, srcRgnB, dstRgn;

void XorRgn(srcRgnA, srcRgnB, dstRgn)
RgnHandle srcRgnA, srcRgnB, dstRgn;

Boolean EqualRgn(rgnA, rgnB)
RgnHandle rgnA, rgnB;

Boolean EmptyRgn(rgn)
RgnHandle rgn;

Boolean PtInRgn(pt, rgn)
Point pt;
RgnHandle rgn;

Boolean RectInRgn(r, rgn)
Rect *r;
RgnHandle rgn;

/*  Graphical Operations on Regions  */
void FrameRgn(rgn)
RgnHandle rgn;

void PaintRgn(rgn)
RgnHandle rgn;

void EraseRgn(rgn)
RgnHandle rgn;

void InvertRgn(rgn)
RgnHandle rgn;

void FillRgn(rgn, pat)
RgnHandle rgn;
Pattern pat;

/*  Graphical Operations on BitMaps  */
void ScrollRect(dstRect, dh, dv, updateRgn)
Rect *dstRect;
int dh, dv;
RgnHandle updateRgn;
```

```
void CopyBits(srcBits, dstBits, srcRect, dstRect, mode, maskRgn)
BitMap *srcBits, *dstBits;
Rect *srcRect, *dstRect;
int mode;
RgnHandle maskRgn;

/*  Picture Routines  */
PicHandle OpenPicture(picFrame)
Rect *picFrame;

void ClosePicture()

void DrawPicture(myPicture, dstRect)
PicHandle myPicture;
Rect *dstRect;

void PicComment(kind, dataSize, dataHandle)
int kind, dataSize;
Handle dataHandle;

void KillPicture(myPicture)
PicHandle myPicture;

/*   The Bottleneck Interface:    */
void SetStdProcs(procs)
QDProcs *procs;

void StdText(count, textAddr, numer, denom)
int count;
Ptr textAddr;
Point numer, denom;

void StdLine(newPt)
Point newPt;

void StdRect(verb, r)
GrafVerb verb;
Rect *r;

void StdRRect(verb, r, ovWd, ovHt)
GrafVerb verb;
Rect *r;
int ovWd, ovHt;

void StdOval(verb, r)
GrafVerb verb;
Rect *r;
```

```
void StdArc(verb, r, startAngle, arcAngle)
GrafVerb verb;
Rect *r;
int startAngle, arcAngle;

void StdPoly(verb, poly)
GrafVerb verb;
PolyHandle poly;

void StdRgn(verb, rgn)
GrafVerb verb;
RgnHandle rgn;

void StdBits(srcBits, srcRect, dstRect, mode, maskRgn)
BitMap *srcBits;
Rect *srcRect, *dstRect;
int mode;
RgnHandle maskRgn;

void StdComment(kind, dataSize, dataHandle)
int kind, dataSize;
Handle dataHandle;

int StdTxMeas(count, textAddr, numer, denom, info)
int count;
Ptr textAddr;
Point *numer, *denom;
FontInfo *info;

void StdGetPic(dataPtr, byteCount)
Ptr dataPtr;
int byteCount;

void StdPutPic(dataPtr, byteCount)
Ptr dataPtr;
int byteCount;

/*  Misc Utility Routines  */
Boolean GetPixel(h, v)
int h, v;

int Random()

void StuffHex(thingptr, s)
Ptr thingptr;
Str255 s;

void ForeColor(color)
long color;
```

```
void BackColor(color)
long color;

void ColorBit(whichBit)
int whichBit;
```

```
int InitResources()

void RsrcZoneInit()

void CreateResFile(fileName)
Str255 fileName;

int OpenResFile(fileName)
Str255 fileName;

void UseResFile(refNum)
int refNum;

int GetResFileAttrs(refNum)
int refNum;

void SetResFileAttrs(refNum, attrs)
int refNum;
int attrs;

void UpdateResFile(refNum)
int refNum;

void CloseResFile(refNum)
int refNum;

void SetResPurge(install)
Boolean install;

void SetResLoad(AutoLoad)
Boolean AutoLoad;

int CountResources(theType)
ResType theType;

Handle GetIndResource(theType, index)
ResType theType;
int index;

int CountTypes()

void GetIndType(theType, index)
ResType *theType;
int index;

int UniqueID(theType)
ResType theType;
```

```
Handle GetResource(theType, ID)
ResType theType;
int ID;

Handle GetNamedResource(theType, name)
ResType theType;
Str255 name;

void LoadResource(theResource)
Handle theResource;

void ReleaseResource(theResource)
Handle theResource;

void DetachResource(theResource)
Handle theResource;

void ChangedResource(theResource)
Handle theResource;

void WriteResource(theResource)
Handle theResource;

int HomeResFile(theResource)
Handle theResource;

int CurResFile()

int GetResAttrs(theResource)
Handle theResource;

void SetResAttrs(theResource, attrs)
Handle theResource;
int attrs;

void GetResInfo(theResource, theID, theType, name)
Handle theResource;
int *theID;
ResType *theType;
Str255  name;

void SetResInfo(theResource, theID, name)
Handle theResource;
int theID;
Str255 name;

void AddResource(theResource, theType, theID, name)
Handle theResource;
ResType theType;
int theID;
```

```
Str255 name;

void AddReference(theResource, theID, name)
Handle theResource;
int theID;
Str255 name;

void RmveResource(theResource)
Handle theResource;

void RmveReference(theResource)
Handle theResource;

long SizeResource(theResource)
Handle theResource;

int ResError()
```

```
long GetScrap(hDest, what, offset)
Handle hDest;
ResType what;
long *offset;

PScrapStuff InfoScrap()

long LoadScrap()

long PutScrap(length, what, source)
long length;
ResType what;
Ptr source;

long UnloadScrap()

long ZeroScrap()
```

Segment Loader

```
void UnLoadSeg(routineAddr)
Ptr routineAddr;

void ExitToShell()

void GetAppParms(apName, apRefNum, apParam)
Str255  apName;
int *apRefNum;
Handle *apParam;

void CountAppFiles(message, count)
int *message;
int *count;

void GetAppFiles(index, theFile)
int index;
AppFile *theFile;

void ClrAppFiles(index)
int index;
```

```
OSErr SerReset(refNum, serConfig)
int refNum;
int serConfig;

OSErr SerSetBuf(refNum, serBPtr, serBLen)
int refNum;
Ptr serBPtr;
int serBLen;

OSErr SerHShake(refNum, flags)
int refNum;
SerShk *flags;

OSErr SerSetBrk(refNum)
int refNum;

OSErr SerClrBrk(refNum)
int refNum;

OSErr SerGetBuf(refNum, count)
int refNum;
long *count;

OSErr SerStatus(refNum, serSta)
int refNum;
SerStaRec *serSta;

OSErr RAMSDOpen(whichPort)
SPortSel whichPort;

void RAMSDClose(whichPort)
SPortSel whichPort;
```

Sound Driver

```
void SetSoundVol(level)
int level;

void GetSoundVol(level)
int *level;

void StartSound(synthRec, numBytes, CompletionRtn)
Ptr synthRec;
long numBytes;
ProcPtr CompletionRtn;

void StopSound()

Boolean SoundDone()
```

```
void SFPutFile(where, prompt, origName, dlgHook, reply)
Point where;
Str255 prompt;
Str255 origName;
ProcPtr dlgHook;
SFReply *reply;

void SFPPutFile(where, prompt, origName, dlgHook, reply, dlgID, filterProc)
Point where;
Str255 prompt;
Str255 origName;
ProcPtr dlgHook;
SFReply *reply;
int dlgID;
ProcPtr filterProc;

void SFGetFile(where, prompt, fileFilter, numTypes, typeList, dlgHook, reply)
Point where;
Str255 prompt;
ProcPtr fileFilter;
int numTypes;
SFTypeList typeList;
ProcPtr dlgHook;
SFReply *reply;

void SFPGetFile(where, prompt, fileFilter, numTypes, typeList, dlgHook, reply,
                dlgID, filterProc)
Point where;
Str255 prompt;
ProcPtr fileFilter;
int numTypes;
SFTypeList typeList;
ProcPtr dlgHook;
SFReply *reply;
int dlgID;
ProcPtr filterProc;
```

System Error Handler

```
void SysError (errorCode)
int errorCode;
```

```
void TEActivate(h)
TEHandle h;

void TECalText(h)
TEHandle h;

void TEClick(pt, xtend, h)
Point pt;
Boolean xtend;
TEHandle h;

void TECopy(h)
TEHandle h;

void TECut(h)
TEHandle h;

void TEDeactivate(h)
TEHandle h;

void TEDelete(h)
TEHandle h;

void TEDispose(h)
TEHandle h;

void TEIdle(h)
TEHandle h;

void TEInit()

void TEKey(key, h)
char key;
TEHandle h;

TEHandle TENew(dest, view)
Rect *dest, *view;

void TEPaste(h)
TEHandle h;

void TEScroll(dh, dv, h)
int dh, dv;
TEHandle h;
```

```
void TESetSelect(selStart, selEnd, h)
long selStart, selEnd;
TEHandle h;

void TESetText(inText, textLength, h)
Ptr inText;
long textLength;
TEHandle h;

void TEInsert(inText, textLength, h)
Ptr inText;
long textLength;
TEHandle h;

void TEUpdate(rUpdate, h)
Rect *rUpdate;
TEHandle h;

void TESetJust(just, h)
int just;
TEHandle h;

CharsHandle TEGetText(h)
TEHandle h;

Handle TEScrapHandle()

long TEGetScrapLen()

void TESetScrapLen(length)
long length;

OsErr TEFromScrap()

OsErr TEToScrap()

void SetWordBreak(wBrkProc, hTE)
ProcPtr wBrkProc;
TEHandle hTE;

void SetClikLoop(clikProc, hTE)
ProcPtr clikProc;
TEHandle hTE;

/* Box drawing utility */
void TextBox(inText, textLength, r, style)
Ptr inText;
```

```
long textLength;
Rect *r;
int style;
```

```
long GetTrapAddress(trapNum)
int trapNum;

void SetTrapAddress(trapAddr, trapNum)
long trapAddr;
int trapNum;

SysPPtr GetSysPPtr()

OsErr WriteParam()

OsErr SetDateTime(time)
long time;

OsErr ReadDateTime(time)
long *time;

void GetDateTime(secs)
long *secs;

void SetTime(d)
DateTimeRec *d;

void GetTime(d)
DateTimeRec *d;

void Date2Secs(d, s)
DateTimeRec *d;
long *s;

void Secs2Date(s, d)
long s;
DateTimeRec *d;

void Delay(numTicks, finalTicks)
long numTicks;
long *finalTicks;

Boolean EqualString(str1, str2, caseSens, diacSens)
Str255 str1, str2;
Boolean caseSens, diacSens;

void UprString(theString, diacSens)
Str255  theString;
Boolean diacSens;

OsErr InitUtil()
```

```
void Enqueue(qElement, qHeader)
QElemPtr qElement;
QHdrPtr qHeader;

OsErr Dequeue(qElement, qHeader)
QElemPtr qElement;
QHdrPtr qHeader;

OsErr HandToHand(theHndl)
Handle *theHndl;

OsErr PtrToXHand(srcPtr, dstHndl, size)
Ptr srcPtr;
Handle dstHndl;
long size;

OsErr PtrToHand(srcPtr, dstHndl, size)
Ptr srcPtr;
Handle *dstHndl;
long size;

OsErr HandAndHand(hand1, hand2)
Handle hand1, hand2;

OsErr PtrAndHand(ptr1, hand2, size)
Ptr ptr1;
Handle hand2;
long size;

void SysBeep(duration)
int duration;

void Environs(rom, machine)
int *rom, *machine;

void Restart()

void SetUpA5()

void RestoreA5()
```

```
long BitAnd(long1, long2)
long long1, long2;

long BitOr(long1, long2)
long long1, long2;

long BitXor(long1, long2)
long long1, long2;

long BitNot(long1)
long long1;

long BitShift(long1, count)
long long1;
int count;

Boolean BitTst(bytePtr, bitNum)
Ptr bytePtr;
long bitNum;

void BitSet(bytePtr, bitNum)
Ptr bytePtr;
long bitNum;

void BitClr(bytePtr, bitNum)
Ptr bytePtr;
long bitNum;

void LongMul(a, b, dst)
long a, b;
Int64Bit *dst;

Fixed FixMul(a, b)
Fixed a, b;

Fixed FixRatio(numer, denom)
int numer, denom;

int HiWord(x)
long x;

int LoWord(x)
long x;
```

```
int FixRound(x)
Fixed x;

void PackBits(srcPtr, dstPtr, srcBytes)
Ptr *srcPtr, *dstPtr;
int srcBytes;

void UnPackBits(srcPtr, dstPtr, dstBytes)
Ptr *srcPtr, *dstPtr;
int dstBytes;

Fixed SlopeFromAngle(angle)
int angle;

int AngleFromSlope(slope)
Fixed slope;

long DeltaPoint(ptA, ptB)
Point ptA, ptB;

StringHandle NewString(theString)
Str255 theString;

void SetString(theString, strNew)
StringHandle theString;
Str255 strNew;

StringHandle GetString(stringID)
int stringID;

void GetIndString(theString, strListID, index)
Str255 theString;
int strListID;
int index;

long Munger(h, offset, ptr1, len1, ptr2, len2)
Handle h;
long offset;
Ptr ptr1;
long len1;
Ptr ptr2;
long len2;

Handle GetIcon(iconID)
int iconID;

void PlotIcon(theRect, theIcon)
Rect *theRect;
```

```
Handle theIcon;

CursHandle GetCursor(cursorID)
int cursorID;

PatHandle GetPattern(patID)
int patID;

PicHandle GetPicture(picID)
int picID;

void GetIndPattern(thePat, patListID, index)
Pattern  thePat;
int patListID;
int index;

void ShieldCursor(shieldRect, offsetPt)
Rect *shieldRect;
Point offsetPt;
```

Vertical Retrace Manager

```
OsErr VInstall(VBLTaskPtr)
QElemPtr VBLTaskPtr;

OsErr VRemove(VBLTaskPtr)
QElemPtr VBLTaskPtr;

QHdrPtr GetVBLQHdr()
```

```
void ClipAbove(window)
WindowPeek window;

void PaintOne(window, clobbered)
WindowPeek window;
RgnHandle clobbered;

void PaintBehind(startWindow, clobbered)
WindowPeek startWindow;
RgnHandle clobbered;

void SaveOld(window)
WindowPeek window;

void DrawNew(window, fUpdate)
WindowPeek window;
Boolean fUpdate;

void CalcVis(window)
WindowPeek window;

void CalcVisBehind(startWindow, clobbered)
WindowPeek startWindow;
RgnHandle clobbered;

void ShowHide(window, showFlag)
WindowPtr window;
Boolean showFlag;

Boolean CheckUpdate(theEvent)
EventRecord *theEvent;

void GetWMgrPort(wPort)
GrafPtr *wPort;

void InitWindows()

WindowPtr NewWindow(wStorage, boundsRect, title, visible, theProc, behind,
                    goAwayFlag, refCon)
Ptr wStorage;
Rect *boundsRect;
Str255 title;
Boolean visible;
int theProc;
WindowPtr behind;
Boolean goAwayFlag;
```

```
long refCon;

void DisposeWindow(theWindow)
WindowPtr theWindow;

void CloseWindow(theWindow)
WindowPtr theWindow;

void MoveWindow(theWindow, h, v, BringToFront)
WindowPtr theWindow;
int h, v;
Boolean BringToFront;

void SizeWindow(theWindow, width, height, fUpdate)
WindowPtr theWindow;
int width, height;
Boolean fUpdate;

long GrowWindow(theWindow, startPt, bBox)
WindowPtr theWindow;
Point startPt;
Rect *bBox;

void DragWindow(theWindow, startPt, boundsRect)
WindowPtr theWindow;
Point startPt;
Rect *boundsRect;

void ShowWindow(theWindow)
WindowPtr theWindow;

void HideWindow(theWindow)
WindowPtr theWindow;

void SetWTitle(theWindow, title)
WindowPtr theWindow;
Str255 title;

void GetWTitle(theWindow, title)
WindowPtr theWindow;
Str255  title;

void HiliteWindow(theWindow, fHiLite)
WindowPtr theWindow;
Boolean fHiLite;

void BeginUpdate(theWindow)
WindowPtr theWindow;
```

```
void EndUpdate(theWindow)
WindowPtr theWindow;

void SetWRefCon(theWindow, data)
WindowPtr theWindow;
long data;

long GetWRefCon(theWindow)
WindowPtr theWindow;

void SetWindowPic(theWindow, thePic)
WindowPtr theWindow;
PicHandle thePic;

PicHandle GetWindowPic(theWindow)
WindowPtr theWindow;

void BringToFront(theWindow)
WindowPtr theWindow;

void SendBehind(theWindow, behind)
WindowPtr theWindow, behind;

WindowPtr FrontWindow()

void SelectWindow(theWindow)
WindowPtr theWindow;

Boolean TrackGoAway(theWindow, thePt)
WindowPtr theWindow;
Point thePt;

void DrawGrowIcon(theWindow)
WindowPtr theWindow;

void ValidRect(goodRect)
Rect *goodRect;

void ValidRgn(goodRgn)
RgnHandle goodRgn;

void InvalRect(badRect)
Rect *badRect;

void InvalRgn(badRgn)
RgnHandle badRgn;

int FindWindow(thePoint, theWindow)
Point thePoint;
```

```
WindowPtr *theWindow;

WindowPtr GetNewWindow(windowID, wStorage, behind)
int windowID;
Ptr wStorage;
WindowPtr behind;

long PinRect(theRect, thePt)
Rect *theRect;
Point thePt;

long DragGrayRgn(theRgn, startPt, boundsRect, slopRect, axis, actionProc)
RgnHandle theRgn;
Point startPt;
Rect *boundsRect, *slopRect;
int axis;
ProcPtr actionProc;
```

PART FOUR

APPENDICES

Appendix A
Macsbug Reference

Introduction

The Macsbug family of Macintosh debuggers gives the programmer visibility into and control of executing Macintosh programs. The debugger resides on the same machine as the executing program. Macsbug allows the programmer to interrupt and resume the program, set and clear breakpoints, and display memory in a variety of formats. The debugging information is displayed on the executing program's Macintosh screen (temporarily replacing the program's screen image).

Macsbug uses about 40K of memory when it's installed. It uses about 18K of disk space.

Macsbug takes control under the following circumstances:

1. When the programmer interrupts execution by pushing the "programmer switch" interrupt button (the rearmost of the programmer switch pair);

2. When most of the fatal system errors occur; or

3. When the `Debugger` trap (`$A9FF`) or `DebugStr` trap (`$ABFF`) is executed.

LightspeedC is distributed with two members of the Macsbug family: MaxBug, which uses a full-size alternate screen, and knows about procedure and trap names; and MacXLBug, which works with the Mac XL (a Lisa with MacWorks) and also uses a full-size alternate screen.

To install Macsbug on a disk, use the Finder to move the appropriate debugger onto your boot disk. Rename the debugger file as *Macsbug*, then choose **Shut Down** from the **Special** menu. Re-insert your disk, and the Macintosh will boot. The boot screen will then say **Macsbug installed** in addition to **Welcome to Macintosh** (unless you have a custom start up screen).

The Macintosh expects the debugger to be named *Macsbug*, so if you don't want Macsbug installed when you boot, you can just rename the file something other than *Macsbug*.

Using Macsbug

Macsbug will take control when you hit the interrupt switch, when most fatal system errors occur or when a `Debugger` or `DebugStr` trap is executed. When a system error occurs, the error is diplayed and you receive a > prompt. (There are some system errors which will still give you the "bomb" dialog box. Most of the time you can hit the interrupt button to get into Macsbug from there.) On other breaks, Macsbug will display the machine state (the program counter (PC)), the current instruction, the status register (SR), the data registers (D0 through D7), the address registers (A0 through A7) and prompt you for a command with a >.

Command Syntax

The following sections summarize the Macsbug command syntax.

Program Execution Control

G *a*	Go; resume program execution at address *a* (if *a* is omitted, the program resumes at the value of the PC)
T	Trace; execute one instruction (treats A-Traps as one instruction)
S *n*	Step; execute *n* instructions (*n* defaults to 1), step into A-Traps
BR *a n*	BReakpoint; interrupt program execution the *n*-th time the PC reaches *a* (*n* defaults to 1). Up to six breakpoints may be active at a time.
CL *a*	CLear the breakpoint at address *a*. If *a* is omitted, CL clears all breakpoints.
GT *a*	Go Till; interrupt program execution the next time the PC reaches *a*
ST *a*	Step Till; step through instructions until the PC reaches *a*
MR	Magic Return. Execute until the PC reaches the address on the top of the stack. Most useful when issued just following a JSR or BSR, to return to the instruction after the call.

(NOTE: BR, GT, and MR will not work if the target address is in ROM. To break in ROM you must use ST.)

Set and Display commands

DM *a n*	Display Memory; starting at address *a*, display *n* bytes (*n* defaults to $10 (hex 10)). *n* may take one of three four-character string values, and the data is displayed accordingly. These possible values for *n* are: 'IOPB' - display as I/O parameter block 'WIND' - display as window record 'TERC' - display as text edit record
SM *a v v v...*	Set Memory; starting at address *a*, replace memory by values *v*
D*n v*	Data Register; sets data register *n* to value *v* (if *v* is omitted, displays the data register value)
A*n v*	Address Register; sets address register *n* to value *v* (if *v* is omitted, displays the value in the address register)

PC *v*	Program Counter; sets the PC to value *v* (if *v* is omitted, displays the value of the PC)
SR *v*	Status Register; sets the SR to value *v* (if *v* is omitted, displays the value of the SR)
TD	Total Display; displays the next instruction to execute, all the registers, the PC and SR
IL *a n*	Instruction List; starting at address *a*, disassemble and display *n* instructions (*n* defaults to $10)
ID *a*	Instruction Disassembly; disassemble and display the instruction at address *a*

Expressions

Wherever a command accepts an address or a value an expression may be substituted. The following notations are used in Macsbug expressions:

+	Addition
–	Subtraction
.	Last address given to DM, SM, IL or ID
RA*n*	The value in address register *n*
RD*n*	The value in data register *n*
@	"Contents of" operator
$	Interpret the following integer as hexadecimal (default)
&	Interpret the following integer as decimal
PC	the value of the PC
'xxx...'	the value of the given string

Macsbug understands trap names. A trap name evaluates to its trap number. In addition, if the **Macsbug Symbols** option is set, you may use your LightspeedC function names in expressions. For more details, see Chapter 7, "Debugging", in Part Two of this manual.

A-Trap Commands

These commands monitor the 68000 *A-Trap* instructions, which are used to invoke Macintosh Operating System and Toolbox routines in the ROM. These commands take up to 6 parameters. *T1* and *T2* indicate the range of A-Traps to monitor, *A1* and *A2* specify the address range which to monitor for A-Trap calls, and *D1* and *D2* specify the range in which D0 must be for the break to occur. All of these parameters are optional. If both *T1* and *T2* are absent, *T1* defaults to 0, *T2* to $1FE. If *T1* is the only argument, the command is only active for that trap. *A1* defaults to 0, *A2* to the top of memory, *D1* to 0 and *D2* to $FFFFFFFF.

In using the A-Trap commands, the trap numbers *T1* and *T2* are actually the low order 9 bits of the trap instruction. For instance, to break on the trap $A97D (NewDialog), use the value $17D. In addition, you may use the trap name instead of the number.

AB *T1 T2 A1 A2 D1 D2*	causes a break when the conditions are met
AT *T1 T2 A1 A2 D1 D2*	traces the A-Traps meeting the conditions
AH *T1 T2 A1 A2 D1 D2*	does an HC (see below) before executing the trap when conditions are met. *T1* must be at least $2F.
AR *T1 T2 A1 A2 D1 D2*	remembers the last A-Trap which met the given conditions. Use AR without arguments to list the trap remembered.
AX	clears all A-Trap commands

Program Termination

RB	ReBoots the Macintosh (This doesn't always work well—especially on HyperDrive—use the reset button)
ES	Exit to Shell: returns to the Finder (or to LightspeedC)
EA	Exit to Application: relaunches the current application

Heap Zone Commands

HD *mask*

Heap Dump: Displays a list of all blocks in the current heap. The display format is:

addr type size [flags pointer] [*] [refnum id type]

mask is optional. It limits the heap display and summary to those blocks which satisfy the condition in the mask. Possible values for the mask are:

'H' : only "handle" (relocatable) blocks are displayed
'P': only "pointer" (non-relocatable) blocks are displayed
'F': only free blocks are displayed
'R': only resource blocks are displayed
'xxxx': blocks of the given resource type are displayed

HT *mask*

Heap Total. Displays a heap summary line for the *mask* given. The format of the summary line is:

HLP PF *#handle-blocks #locked-blocks #purgeable-blocks space-occupied-by-purgeable-blocks #pointer-blocks free-space*

HC — Heap Check. Checks the consistency of the heap. Warning: This command will hang if the heap is munged badly.

HX — Heap eXchange. Toggles the heap display between system and application heaps.

Miscellaneous Commands

F *a n d m* — Find. Starting at address *a*, search *n* bytes for data *d*, after masking the data with *m*.

WH *a* — WHere. If *a* is less than 512, give the PC at which the trap code resides. Otherwise, display the trap and trap starting address which is closest to *a* .

CV *a* — ConVert. Displays *a* in unsigned hexadecimal, signed hexadecimal, signed decimal and text. When used with expressions, you have an all-purpose programmer's calculator!

CS *a1 a2* — CheckSum. Does a checksum of the address range *a1..a2*. When no arguments are present, it returns `CHKSUM T` if the checksum of the last given range is the same as when it was last calculated, `CHKSUM F` otherwise.

SS *a1 a2* — Step Spy. Does a checksum of the contents of the address range *a1..a2* between the execution of each instruction. Breaks when the checksum changes. (Caution: Very slow but very useful. Disk access does not work well with `SS` active.)

AS *A1 A2* — A-Trap Spy. Similar to Step Spy, but calculates the checksum before each A-Trap. (This command is cancelled by `AX`.)

SC — Stack Crawl. This assumes that `LINK` and `UNLK` instructions have been performed at the beginning and end of each function (generally true in LightspeedC). It returns a list of the active stack frames and their return addresses. This is useful to see who called what and where. (Hint: If you are at a `LINK` instruction, step beyond it before you do `SC`.)

PX — Procedure symbol eXchange. Toggles the visibility of procedure names in `IL` and `ID` commands. Defaults to enabled.

RX — Register eXchange. Toggles the visibility of the register dump during tracing and stepping. Defaults to enabled.

DX — Debugger eXchange. Toggles Debugger trap entry. When enabled, calling `Debugger()` or `DebugStr()` will break into Macsbug; otherwise it won't cause a break. Defaults to enabled.

Handy Hints

Depressing the return key will repeat the previous command. Exceptions: after an IL or ID command, hitting return will display the next instructions; after DM it will display the next memory locations. After an MR command, it will execute a T (Trace).

Hitting the backspace key while a Macsbug command is listing will terminate the command. Depressing the space key will pause the listing. Hitting the space key again will restart the display.

Hitting the ` key (at the top left of the keyboard) while in Macsbug will toggle the display between the Macsbug screen and the program's screen.

Sometimes you will break into Macsbug with the disk still spinning. To stop the disk, enter DM DFF1FF. This will hit the Woz machine (disk drive controller) address that stops the disk.

The word $4E71 is the NOP instruction. If you want to take instructions out of your program, use this word to NOP out instructions in your code stream. Note: the NOP is a two byte instruction. If the instruction you are going to NOP is longer than two bytes, remember to use enough NOPs to clear out the whole instruction.

Appendix B
RMaker Reference

Resource Files and Macintosh Program Design

Macintosh programs are designed around *objects*. Windows, menus, dialog boxes, and alerts are all objects which the Macintosh knows about. These objects may be constructed "on the fly" from within a Macintosh program, or they may be specified independently from the program and stored in the resource fork of the application's file. This independent specification has the advantage of separating the program's functions (as specified in the code) from the program's user interface design (its "look"). This is an important first principle in designing programs for the Macintosh.

RMaker is a *resource compiler*. It takes a textual specification of the objects to be used in a program and produces the resource data structures which are understood by the Macintosh Toolbox routines.

Using RMaker

To create a resource file, use the LightspeedC editor to create the RMaker source file. Transfer to RMaker. It will display a file selection box and wait for you to choose a file. Choose your RMaker source file as input, and RMaker will produce a resource file from the given source.

RMaker File Format

RMaker input is line-oriented and has a quite rigid syntax. The RMaker input file consists of an output specification followed by an arbitrary number of resource definitions separated by blank lines. Comments are also allowed, either as whole lines or as tags at the end of lines. Comment lines begin with an asterisk `*`. Comments at the end of lines are preceded by two semicolons `;;`.

RMaker files begin with the output specification: the name of the output file on one line, followed by a line assigning a file signature. The file signature is a four-character file type followed by a four-character file type. For example, if I want to name the output file `Sample` and give it the file type `????` and creator `SAMP`, the first two lines of the RMaker input file would be:

```
Sample              ;; the name of the output file
????SAMP            ;; the file type ???? and creator SAMP
```

The file signature line may be left blank, in which case the file type and file creator will be set to nulls (four bytes of 0 each). The output specification may be preceded by an arbitrary number of blank lines or comment lines.

If the file name line starts with an exclamation point, this tells RMaker to merge the resources in the RMaker specification into the file name given. In this case, omitting the file signature will leave the file's signature unchanged.

The body of an RMaker source file consists of declarations of groups of resources headed by a resource type clause. Resource type sections are introduced by a `TYPE` declaration:

(Note: in the following syntax, the brackets [] enclose optional data. Words in italics describe user-supplied data.)

`TYPE` *resource type* [= *resource type*]

If the optional "=" clause is not present, the first resource type must be a predefined type. If the clause is present, the resource type named there must be a predefined type. The = clause allows users to define their own resource types.

RMaker resource declarations are always in the following form:

`[name],` *ID* `[(`*attributes*`)]`
type-specific resource data

The attributes byte must be surrounded in parentheses. See the "Resource Manager" chapter of *Inside Macintosh* for the definition of resource attributes.

The statement

`INCLUDE` *filename*

tells RMaker to take the resources from *filename* and include them in this resource specification.

Numbers are decimal by default.

Special characters may be entered with a backslash followed by two hex digits giving the ASCII code of the character (e.g. `\14` for the apple symbol).

Put ++ at the end of a line to signify that the line is continued on the next line.

Individual Resources and Their RMaker Constructs

Resource definitions are grouped according to type. A resource type group is headed by the word TYPE followed by the resource type name on one line, followed by definitions of resources of that type. Resource definitions consist of a first line containing the resource name (optional), followed by a comma and a number, followed by a resource attribute byte in parentheses (optional). Following this first line is resource-specific data which may be on one or many lines, according to the type of the resource.

Resource definitions must be terminated by a blank line.

There are twelve defined resource types recognized by RMaker:

'ALRT' - Alert
'BNDL' - Bundle
'CNTL' - Control
'DITL' - Dialog (or Alert) Item List
'DLOG' - Dialog
'FREF' - File Reference
'GNRL' - General
'MENU' - Menu
'PROC' - Procedure (contains code)
'STR ' - String
'STR#' - String List
'WIND' - Window

The following sections illustrate the different types of resource declarations:

'ALRT' - Alert Template

```
TYPE  ALRT
   , 128          ;; the resource number
70 100 150 412    ;; rectangle for the alert (top left bottom right)
10                ;; the resource ID for the item list
FFFF              ;; stages word (in hex)

* always remember the blank line at the end of a resource
* definition - it is a required separator
```

'BNDL' - Bundle Template for Application

```
TYPE  BNDL
   ,128              ;; resource number
SAMP 0               ;; creator for bundle
ICN#                 ;; resource type (icon list)
0 128 1 129          ;; local ID 0 maps to ICN# resource 128, 1 to 129
FREF                 ;; resource type (file reference)
0 128 1 129          ;; local to FREF mapping 0 to 128, 1 to 129
```

'CNTL' - Control Template

```
TYPE  CNTL
   ,128              ;; resource number
MyControl            ;; title for control
10 10 20 20          ;; rectangle for control (top left bottom right)
Visible              ;; may also be Invisible
0                    ;; CDEF proc ID
0                    ;; reference constant (defines control type)
0 100 0              ;; minimum value, maximum value, initial value
```

'DITL' - Dialog (or Alert) Item List

```
TYPE  DITL
   ,10               ;; resource number
9                    ;; number of items in the item list

button               ;; enabled button items (enabled by default)
20 20 40 100         ;; rectangle (window-relative coordinates)
Cancel               ;; text in the button

radioButton          ;; radio button item
50 20 70 120         ;; rectangle (includes button and text)
Push Me              ;; the text will go to the right of the button

radioButton disabled          ;; dimmed radio button item
50 20 70 120         ;; rectangle (includes button and text)
Can't Push Me        ;; you can't push a disabled button

checkBox             ;; check box item
80 20 100 120        ;; rectangle (includes check box and text)
Check Me             ;; the text goes to the right of the box

staticText           ;; static text item
20 120 40 320        ;; the rectangle into which the text is placed
This text would get placed next to the Cancel button ;; the text
```

```
editText Disabled ;; disabled editable text item
50 140 90 320     ;; coordinates of the box for editable test
initial string    ;; the edit test gets initialized to this string.

editText                ;; editable text item (enabled by default)
100 140 120 320         ;; rectangle
you can edit this text  ;; the initial string for editable item

iconItem          ;; for the display of an icon
100 100 132 132   ;; rectangle should be 32x32
3                 ;; resource ID for icon (Type ICON)

picItem           ;; for the display of a Quickdraw picture
30 100 20 200     ;; display rectangle (picture will be scaled)
57                ;; resource ID for picture (Type PICT)

userItem          ;; a user-defined item
30 40 80 90       ;; the rectangle
```

'DLOG' - Dialog Template

```
TYPE  DLOG
   ,128           ;; the resource number
My Dialog Box     ;; a message
70 100 150 412    ;; the rectangle (top left bottom right)
Visible NoGoAway  ;; may also be Invisible, and may also be GoAway
0                 ;; the dialog definition ID
0                 ;; the refCon, available to the user
10                ;; the resource ID for the dialog item list
```

'FREF' - File Reference

```
TYPE  FREF
   ,128           ;; the resource number
APPL 0            ;; the file type and local ID
```

'GNRL' - General

'GNRL' is used to define your own resource types and define their format. The resource's format is constructed from "elements". The elements available are:

- .P Pascal string
- .S String without a leading length byte
- .I Decimal integer
- .L Decimal 32-bit integer
- .H Hexadecimal integer
- .R Read the given resource from the given file.
  .R takes the arguments *filename resource type resource ID*

```
TYPE ICN# = GNRL                ;; define the type ICN#
   ,128                         ;; the resource ID
.H                              ;; hexadecimal data follows
0001 8000 0002 4000             ;; ICN#'s need 2 icons (icon and
0003 C000 0004 2000             ;; mask) of 32x32 bits each, or 32
...                             ;; lines of two longwords apiece.
FFFF FFFF FFFF FFFF
```

'MENU' - Menu

```
TYPE  MENU
   ,10              ;; the resource number (Menu ID)
MyMenu              ;; the menu title
First Item          ;; the first menu item
Second Item /S      ;; the second menu item with a command-key "S"
(Third Item         ;; the third item (disabled by the preceding "(")
(-                  ;; a grey line (this is item #4)
Fifth Item          ;; the fifth item
```

'PROC' - Procedure (contains code)

```
TYPE  PROC
   ,128             ;; the resource number
Filename            ;; the code from this file will get placed
                    ;; in the resource
```

'STR ' - String

```
TYPE  STR           ;; spelled 'STR ' - trailing space  required!
   ,128             ;; the resource number
My Wild Irish Rose ;; the string assigned to the resource 128
```

'STR#' - String List

```
TYPE  STR#
   ,128           ;; the resource number
2                 ;; the number of strings in the list
The First String
And the second string          ;; the two strings in the list
```

'WIND' - Window

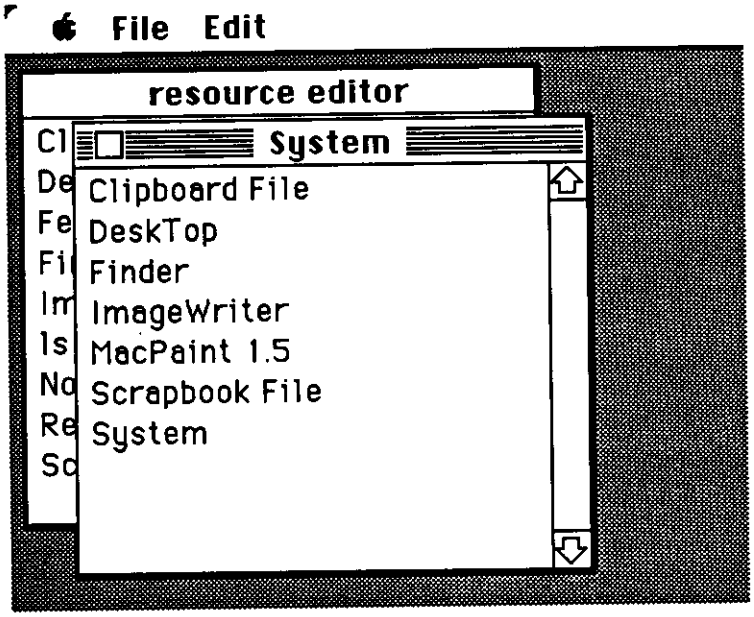
```
TYPE  WIND
   ,128           ;; the resource ID
My Window         ;; the window title
40 40 200 472     ;; the window rectangle (top left bottom right)
Visible GoAway    ;; may also be Invisible and may also be NoGoAway
0                 ;; the window definition ID
0                 ;; refCon, a long word available to the User
```

Appendix C
ResEdit Reference

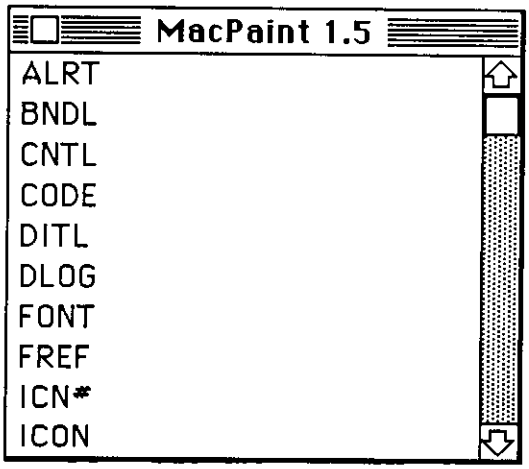
Introduction

ResEdit is a graphically based resource editor, which allows the creation of resources and editing of existing resources. It is composed of several individual editors for specific resource types, plus a general editing facility which lets you edit the hexadecimal resource data directly for any resource type.

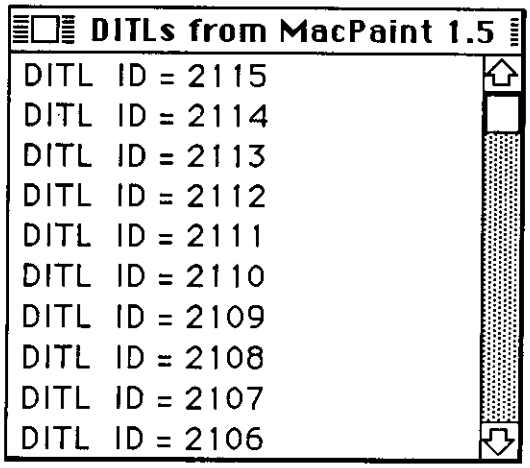
When you launch ResEdit, you get a window for each of your mounted disks listing the files on the disk.



Opening a file shows a list of all the resource types in the resource fork of the file.



If there is no resource fork for the file, ResEdit will inquire whether you want to create a resource fork for the file. Opening a resource type lists the individual resources of that type in the file.



Opening a particular resource will invoke the specific resource editor for that type if there is one. If not, the general (hexadecimal) resource data editor is invoked.

ResEdit is easy to use and largely self-documenting. Essentially, what you see is what you get. All of the resource editors either put up dialog boxes that you fill in, or provide pictures of the resources that you manipulate graphically.

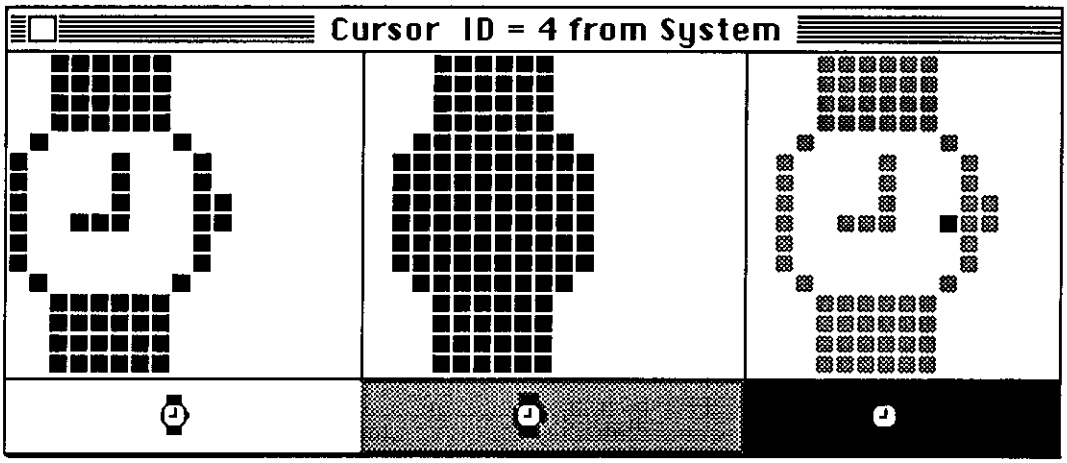
The standard meanings of the **File** and **Edit** menus are altered slightly depending upon what kind of window is in front. For instance, **New** will create a new instance of the kind of element in the front window. If the disk contents window is in front, it creates a new file; if the file resource type list is in front, a new resource type is created; if a resource ID list is in front, a new resource of that type is created. The **Edit** menu commands similarly operate on the type of window that is in front.

When a particular resource is selected out of a resource ID list, or whenever a window for a particular resource is front, the command **Get Info** will put up a dialog box containing that resource's parameters in editable fields. You can use this feature to change the resource ID, set resource attributes and other properties particular to that resource.

When you double-click on a particular resource, the resource editor for that type of resource is invoked. If no specialized editor exists, the general resource editor is invoked.

The CURS Editor

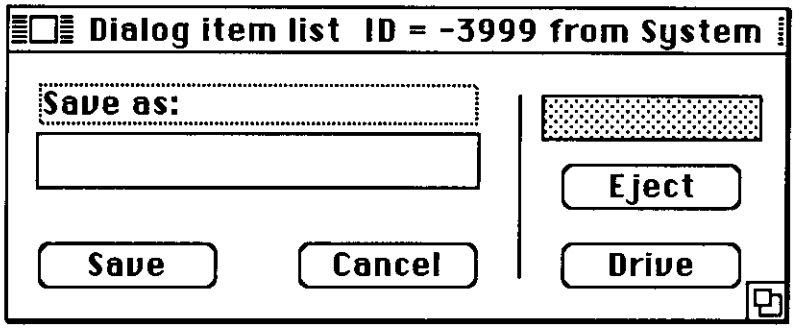
When the CURS editor is invoked, it displays a window with three images of the cursor which may manipulated by the mouse.



The leftmost image is how the cursor will appear. The image in the center is the mask for the cursor. This will determine how the cursor will appear on different backgrounds (as you can see on the display beneath the editable cursor fields). The rightmost panel lets you set the cursor's hotspot, which appears at a black dot among the gray dots of the cursor.

The DITL Editor

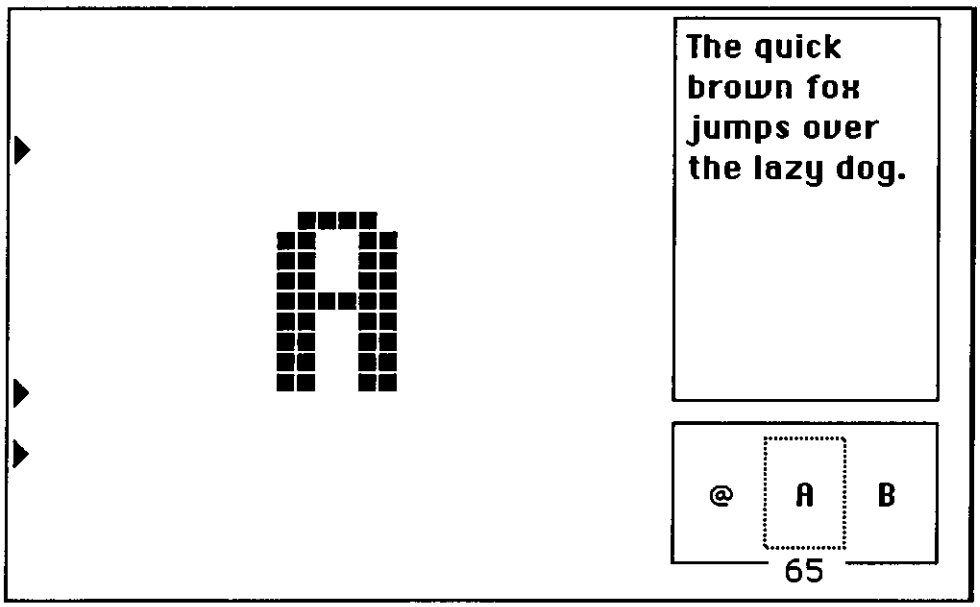
When you start the DITL resource editor, a window is displayed showing the dialog items as you would see them in the dialog window.



You can select items by double-clicking on them, which brings up a window allowing you to edit that item individually. You can change the size of an item by clicking and dragging the lower right corner of the item. You can move an item by dragging it with the mouse. When the dialog window is front, the menu command **New** will create a new dialog item.

The FONT Editor

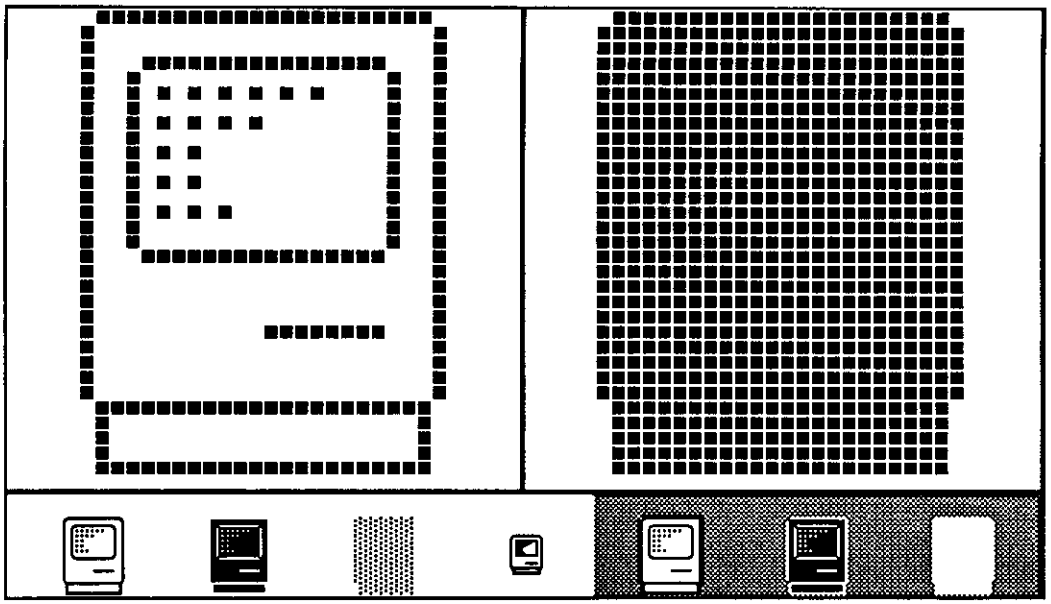
The FONT editor displays three panes: the sample text pane, the character selection pane and an editable window which shows one character at a time.



The selection pane shows three characters. Clicking on the left character in the pane moves you downwards in the character set, clicking on the right character moves upward through the set. You select a character in the selection pane, and edit the character in the edit pane. The three triangles in the edit pane represent the font ascent, baseline and descent.

The ICN# Editor

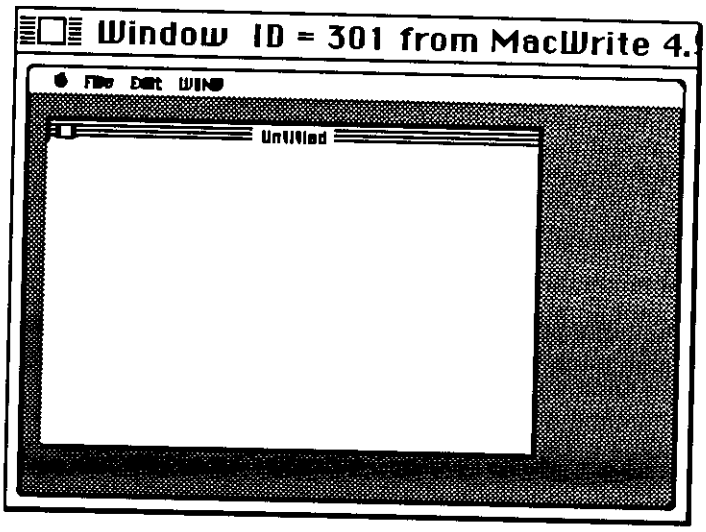
The `ICN#` editor displays two panes:



The left pane represents the icon, the right the icon mask. Along the bottom of the window the icon is displayed in its unselected, selected and dimmed states on white and gray backgrounds. You use the mouse to set bits in the large icon and mask panes to create your icons.

The WIND and DLOG Editor

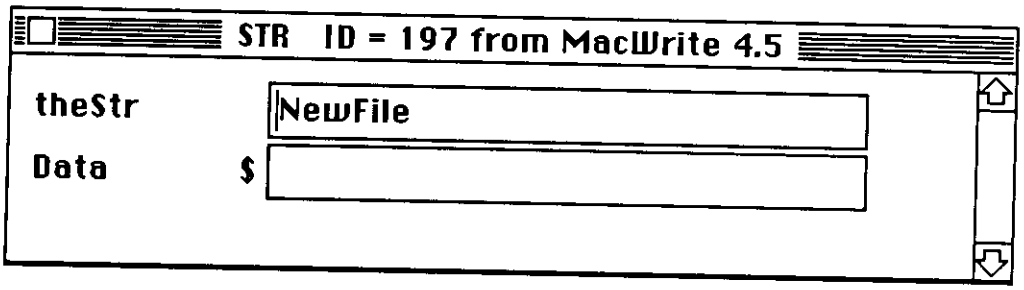
The WIND editor displays a window with a picture of a window on a desktop.



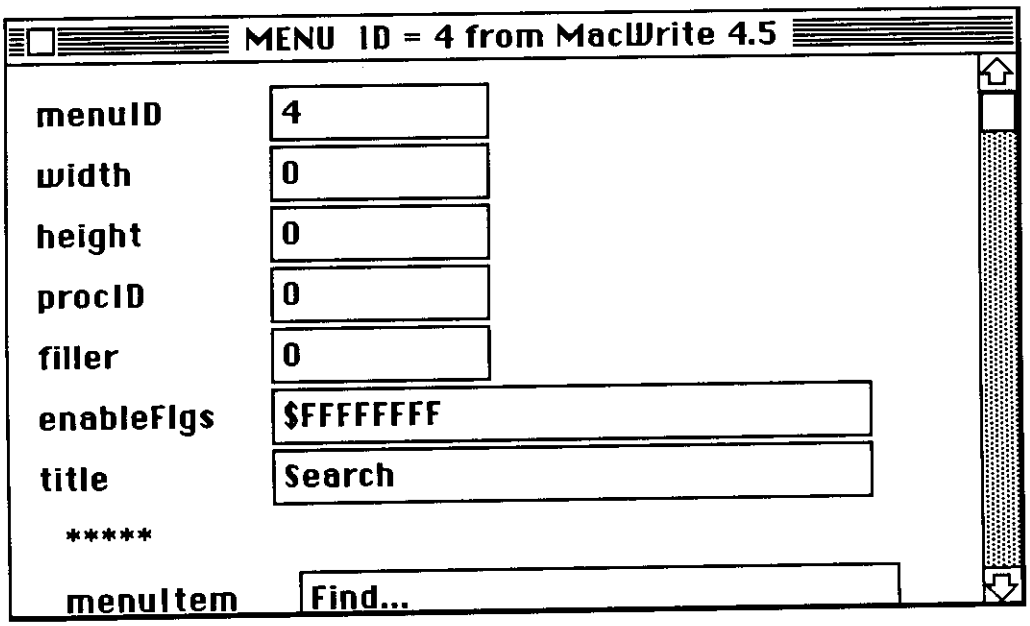
You can grow the window picture by dragging its lower right corner. You move this window picture by dragging its title bar. This editor is also invoked for the resource type DLOG.

Other Resource Type Editors

ResEdit allows you to edit other types of resources with a dialog box style editor. The fields of the particular resource type may be set by typing the information for the field into the box along side the field name. Here's an example of the dialog box you get for the type STR:



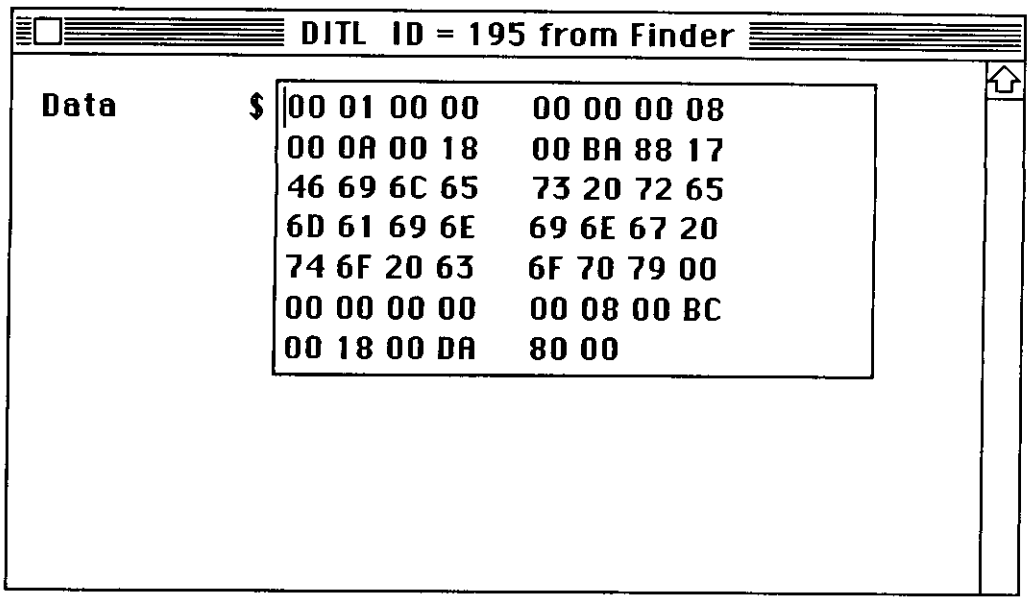
Here's what you get for the resource type MENU:



Click on the row of asterisks to select the menu item below the row. Double-click on the row of asterisks to insert a new menu item below the row.

The General Editor

The general editor shows a dialog box with one editable field labelled **Data.**



You may change the hexadecimal data in this field. Make sure you understand the data format for the resource before editing it! (Resource format documentation can be found in *Inside Macintosh.*)

Appendix D
Compiler Error Messages

(Error messages are listed in alphabetical order.)

'&' (address-of) operator illegal here

The address-of operator was used on an object (such as a register variable or a bitfield) that doesn't have an address. Example:

```
function()
{
    register int i;
    int *p = &i;    /* Illegal - i is a register */
}
```

'*symbol*' has not been declared

The identifier *symbol* has been used before it has been declared.

'*symbol*' is not a formal parameter

The *symbol* was declared as if it was a formal parameter in a function definition, but it is not in the parameter list. Example:

```
function()
int left_out_of_parameter_list;  /* Illegal */
{
}
```

a code resource may not have global or static data

If the LightspeedC project is a code resource, it is illegal to have global or static data. Example:

```
static int i;  /* Illegal in a code resource */
```

"array of functions" is not allowed

An array of functions is not allowed. However an array of pointers to functions is allowed. Example:

```
int array_of_functions[]();                 /* Illegal */
int (*array_of_ptrs_to_functions[SIZE])(); /* Legal */
```

"array of void" is not allowed

Since `void` objects are not allowed, arrays of `void` objects are not allowed either. However, it is legal to have an array of pointers to `void`. This is a C idiom for an array of generic pointers. Example:

```
void abyss[SIZE];              /* Illegal */
void *generic_ptr_array[SIZE]; /* Legal */
```

break outside of loop or switch

The `break` statement must be inside a loop (`while`, `do-while`, `for`) or a `switch`. Example:

```
function()
{
    int i;

    break;              /* Illegal */
    while(1)
        break;          /* Legal */
    do { break; }       /* Legal */
        while(1);
    switch(i){
    case 1:
        break;          /* Legal */
    }
}
```

call of non-function

An expression in the function call position does not evaluate to a function. Example:

```
function(f_ptr)
int (*f_ptr)();   /* f_ptr is a pointer to a function */
{
    int i, j, k, result;

    /* probably left out an operator between i and (j+k) */
    result = i(j+k);          /* Illegal */
    result = f_ptr(i, j);     /* Illegal */
    result = (*f_ptr)(i,j); /* Legal */
}
```

cannot initialize auto struct/union/array

Only global or static structures/unions/arrays may be declared with initializers. Example:

```
int array[6] = { 0,1,2,3,4,5 };      /* Legal */
function()
{
    int array[6] = { 0,1,2,3,4,5 }; /* Illegal - auto array */
    static struct{
        double pi;
    } pi = { 3.14159 };        /* Legal - because of static */
}
```

can't open #include'd file

The file to be `#included` cannot be opened. Its name may be misspelled, or it may be on the wrong volume.

case not in switch

The `case` keyword was found outside of a `switch` block.

constant required

A constant was required, but none was found. Example:

```
#define aConstant  12
function()
{
    int h;
    int x[2];  /* Legal - array dimension is a constant */
    int y[h];  /* Illegal - h is not a constant */
    enum{
       color = h, /* Illegal - enum values must be constant */
       shape = -2.4,   /* Illegal - enum values must be int */
       size = aConstant /* Legal */
    } attributes;
    struct{
        unsigned illegal_bitfield : h;  /* Illegal */
       unsigned legal_bitfield : size; /* Legal */
    }bits;
    int z[size];  /* Legal - array dimension is a constant */
    int w[3.14159];  /* Illegal - constant must be an int */
}
```

continue outside of loop

The `continue` statement must be inside a loop (`while`, `do-while`, `for`). Example:

```
function()
{
    int i;

    continue;              /* Illegal */
    while(1)
        continue;          /* Legal */
    do { continue; }       /* Legal */
        while(1);
    switch(i){
    case 1:
        continue;          /* Illegal */
    }
}
```

declarator too complex

The declaration statement is too complex. Example:

```
int *************************
    x[1][1][1][1][1][1][1][1][1][1][1][1];
```

default not in switch

The `default` keyword was found outside of a `switch` block.

duplicate case

A `case` constant expression evaluates to the same value as a previous `case` constant expression within the same `switch` block.

duplicate default

There may only be one `default` specified in a `switch` block.

expression too complex

An expression was too complex. Try breaking up the expression into subexpressions. Example:

```
typedef int ****************************** indirect_type;
function(meta_ptr)
indirect_type ******************************meta_ptr;
{
     int i;
     indirect_type intermediary;

     /* The following expression is too complex */
     i = ******************************
         ****************************** meta_ptr;

     /* The equivalent broken up into two subexpressions */
     intermediary = ******************************meta_ptr;
     i = ****************************** intermediary;
}
```

formal parameter '*symbol*' appears more than once

The formal parameter *symbol* was used in the function parameter list more than once. Example:

```
function(again, again, again)  /* Illegal */
int again;
{
}
```

"function returning array" is not allowed

A function cannot return an array. However a function can return a pointer that is the address of an array. Example:

```
char function_returning_array()[SIZE];        /* Illegal */
char *function_returning_ptr_to_array();      /* Legal */
```

"function returning function" is not allowed

A function cannot return a function. However a function can return a pointer to a function. Example:

```
int function_returning_function()();    /* Illegal */

typedef int (* function_ptr)();
function_ptr  f_returning_ptr_to_f();   /* Legal */
```

identifier does not match #ifdef

The `#endif` and `#else` macro preprocessor statements take an optional argument which is the symbol that the matching `#ifdef` or `#ifndef` (but not `#if` or `#elif`) used. If the optional argument is present, it must match the corresponding `#ifdef` or `#ifndef`. Example:

```
#ifdef macro_name
#else  Other_name /* Illegal - identifier does not match */
#endif Other_name /* Illegal - identifier does not match */

#ifdef macro_name

#endif macro_name /* Legal */
```

illegal #else

Every `#else` or `#elif` must have a matching `#if`, `#ifdef`, or `#ifndef`. Example:

```
#ifdef name
#else
#else              /* Illegal - has no matching #ifdef */
#endif name
#elif condition    /* Illegal - has no matching #if */
```

illegal #endif

An `#endif` was encountered that didn't have a matching `#if`, `#ifdef`, or `#ifndef`.

illegal array bounds

Examples:

```
int a[-7];       /* Illegal - bounds can't be negative */
int b[0];        /* Illegal - bounds can't be zero */
char c[32768];   /* Illegal - total array size > 32767 bytes */
int d[16384];    /* Illegal - total array size > 32767 bytes */
long e[8192];    /* Illegal - total array size > 32767 bytes */
```

illegal cast

It is illegal to cast `structs`/`unions` to other types. It is legal to cast numerical values/pointers to other numerical values/pointers. Example:

```
typedef struct {int v, h;} Point;
function()
{
    Point p;
    long l;

    p = (Point) l;  /* Illegal */
}
```

illegal floating-point operation

You cannot use some of the C operators on floating point values. Example:

```
function()
{
    double d1;
    double d2;
    double d3;

    d3 = d1 % d2;    /* Illegal - can't do floating modulo */
    d3 = d1 << 3;    /* Illegal - can't do floating shifts */
    d3 = 3 << d1;    /* Illegal - can't do floating shifts */
}
```

illegal operation on array

You cannot use many of the C operators on arrays. Example:

```
function()
{
    char a1[4];
    char a2[4];
    char a3[4];
    int i;

    if ( a1 == a2 )  /* Legal - compares arrays' addresses */
        a1++;        /* Illegal - can't increment arrays */
    a1 = 0;          /* Illegal - can't assign to an array */
    a1 = a2;         /* Illegal - can't assign arrays */
    i = a1 - a2;     /* Legal - subtracts arrays' addresses */
}
```

illegal operation on function

You cannot use many of the C operators on functions. Example:

```
function()
{
    int f1();
    int f2();
    int f3();
    int i;
```

```
    if (f1 == f2)   /* Legal - compares functions' addresses */
        f1++;       /* Illegal - can't increment functions */
    f1 = 0;         /* Illegal - can't assign to a function */
    f1 = f2;        /* Illegal - can't assign functions */
}
```

illegal operation on struct/union

You cannot use many of the C operators on `structs/unions`. Example:

```
int function()
{
    struct{
        int x;
    }s1, s2, s3;
    int i;

    if (s1 == s2) /* Illegal - can't compare structs */
        s1++;     /* Illegal - can't increment structs */
    s1 = 0;       /* Illegal - can't assign int to a struct */
    s1 = s2;      /* Legal - struct assignment is allowed */
    s3 = s1 + s2; /* Illegal - can't add structs */
    i = s1 - s2;  /* Illegal - can't add structs */
    i = (long)s1; /* Illegal - can't cast a struct */
    return(s1);   /* Illegal - return does an implicit cast */
}
```

illegal pointer/integer combination

Integers may not be assigned to pointers with the exception of the constant zero. In addition, pointers may not be compared with integers, again with the exception of the constant zero. In both cases a cast can be used to force the assignment or comparison where necessary. Example:

```
function()
{
    int *int_ptr;
    int i;

    int_ptr = 0x220;          /* Illegal */
    int_ptr = 0;              /* Legal */
    int_ptr = (int *) 0x220; /* Legal - address of MemErr */
    i = * 0x220;              /* Illegal */
    i = * (int *) 0x220;      /* Legal - contents of MemErr */
    return(int_ptr);          /* Illegal - implicit cast */
}
```

illegal pointer arithmetic

The arithmetic being performed on the pointer is illegal. Example:

```
function()
{
    char *p1, *p2;
    long result;

    result = p1 + p2;  /* Illegal - can't add pointers */
    result = p1 / p2;  /* Illegal - can't divide pointers */
}
```

illegal return type for pascal function

Pascal functions are only allowed to return `void`, integers, and pointers. Example:

```
typedef struct {int v, h;} Point;

pascal Point /* Illegal - can't return a Point */
pascal_function(x,y)
{
    Point p;
    p.v = x;
    p.h = y;

    return p;
}
pascal double illegal_pascal_fnc(); /* Illegal return type */
pascal void pascal_proc();          /* Legal */
```

illegal size for bitfield

Bitfield sizes must be less than or equal to the number of bits in the word type. (See Chapter 12, "C Language Reference", § 8.5.) Labelled bitfield sizes must be greater than zero. Unlabelled bitfield sizes equal to 0 force a word alignment. Example:

```
struct bitfields{
    char c : 9;      /* Illegal - chars have 8 or fewer bits */
    int i : 17;      /* Illegal - ints have 16 or fewer bits */
    int zero : 0;    /* Illegal - bitfield size must be > 0 */
    int good : 11;   /* Legal */
    int : 0;         /* Legal - this forces word alignment */
    long l : 33;     /* Illegal - longs have <= 32 bits */
    int z : -4;      /* Illegal - bitfield size must be > 0 */
};
```

illegal token

Certain characters that are part of the ASCII character set and all of the characters in the extended Macintosh character set are illegal tokens in LightspeedC. The character # is also an illegal token when it is other than the first character in a line preceded by any number of whitespace characters. However, any character can occur within comments, string literals or character literals. Example:

```
int a$variable;          /* Illegal */
char c = '$';            /* Legal to have $ here */
#define N    "###"       /* Legal to have # here */
char d = #define M 4;    /* Illegal */
```

illegal type for bitfield

The only allowed types for a bitfield are `char`, `short`, `int`, and `long`, and these types prefixed by the modifier `unsigned`. Any other type for a bitfield is illegal. Example:

```
struct bitfields{
    char c : 5;              /* Legal */
    unsigned int i : 5;      /* Legal */
    long l : 17;             /* Legal */
    double d : 10;           /* Illegal */
    void *p : 8;             /* Illegal */
};
```

illegal use of inline Macintosh function

Even though the inline Toolbox Trap calls look like function calls, they are not. Hence, it is illegal to take the address of an inline Toolbox Trap call. Another possible source of error is forgetting to put in the parameter parentheses in a built-in call that takes no arguments. Example:

```
function()
{
    void *generic_pointer = FrameRect; /* Illegal */

    /* what a forgetful Pascal programmer might write */
    while (Button)       /* Illegal - () missing */
        ;

    while(Button()) /* Legal */
        ;
}
```

illegal use of type name '*symbol*'

A `typedef` name has appeared where it shouldn't. If a `typedef` name is used instead of a variable name, then an error occurs. Example:

```
typedef char Byte;
function()
{
    return Byte;      /* Illegal */
}
```

illegal use of void

You cannot use most of the C operators on `void` values. Example:

```
void f()
{
    int i, j;
    i = (void)j + 5; /* Illegal - can't add a void */
    i = f() - 3;     /* Illegal - void result from f()
                                   can't be used */
}
```

incomplete macro call

A preprocessor command was found while reading macro parameters during macro expansion. Preprocessor commands are not allowed in this situation. Example:

```
#define macro(arg1, arg2) (arg1 > arg2)

macro(x,
#ifdef AndIfYouCallNow    /* Illegal */
1
#else
2
#endif
)
```

initialized object too complex

Initialization of a deeply nested `struct` will cause this error. Example:

```
struct s1{
struct s2{
...
struct s21{
    int i;
}...}}s = {4};        /* Too deeply nested */
```

initialization to an address is illegal in a non-application

If the LightspeedC project is a code resource, desk accessory, or device driver, it is illegal to initialize an address in global or static memory. Example:

```
static int i;
static int *p = &i;  /* Illegal in non-application */
```

integer constant too large

The absolute value of decimal integer constants must be less than or equal to `2147483647` ($2^{31}-1$). Hexadecimal integer constants must range from `0x0` to `0xffffffff` inclusive. If you want to express an unsigned number larger than `2147483647`, you must express it as a hexadecimal number. Example:

```
unsigned long a = 2147483648;  /* Illegal */
unsigned long b = 0x80000000;  /* Legal - equals 2147483648 */
unsigned long c = 0x100000000; /* Illegal - larger than 32
                                                    bits */
```

invalid declaration

Example:

```
/* Illegal - only a defining instance can be initialized */
extern int x = 1;

extern int y;
int y = 1;          /* Legal */
```

invalid function definition

Example:

```
/* Illegal - function definition can't be declared extern */
extern f1()
{
}

f2() = 2;  /* Illegal */
```

invalid redeclaration of '*symbol*'

An identifier was declared twice and this redeclaration is incompatible with the first. Example:

```
extern int x;
long x;     /* Illegal - x is redeclared differently */

extern int y;
int y;      /* Legal - y is redeclared as the same type */

int z;
int z;      /* Illegal - can only have one defining instance */

typedef int IsARose;
IsARose IsARose;    /* Illegal */
```

invalid storage class

Incompatible attributes have been explicitly or implicitly applied to a data item. Example:

```
register int aGlobal;  /* Illegal - register global */
extern auto int y; /* Illegal - contradictory storage class */
main()
{
}
```

invalid type

The specified combination of type keywords create an ambiguous or impossible data item. Example:

```
unsigned double d;          /* Illegal */
long struct {int i} s;      /* Illegal */
```

label '*symbol*' already defined

The label *symbol*: has been defined twice within the function.

label '*symbol*' not defined

There was a `goto` to the label *symbol*: but none has been defined within the function.

lvalue required

An lvalue is an expression that refers to an object in memory that can be stored to as well as examined. See Harbison & Steele or *K&R* for more details. Example:

```
int int_f();
int *pint_f();
function()
{
    int i;
    int *p = &i;

    /* operand of ++ must be an lvalue */
    7++;            /* Illegal */
    int_f()++;      /* Illegal */
    pint_f()++;     /* Illegal */

    /* left operand of an assignment must be an lvalue */
    int_f() = i;    /* Illegal */
    pint_f() = i;   /* Illegal */
    *pint_f() = i;  /* Legal - * produces an lvalue */
    (*p)++;         /* Legal */
    (*pint_f())++;  /* Legal */
}
```

macro name already #define'd

It is illegal to `#define` a macro name that has already been `#defined`. However, it is always legal to `#undef` a macro name whether or not it has been `#defined` before. If you use a `#undef` before a `#define`, you will be sure that a subsequent `#define` will always work. Example:

```
#define macro_name 1
#define macro_name 2    /* Illegal - already #define'd */

#undef macro_name
#define macro_name 3    /* Legal to #define AFTER an #undef */
```

macro parameter '*symbol*' appears more than once

The formal parameters in a macro definition must appear only once. Example:

```
#define macro(again,again)   (again+7)   /* Illegal */
```

missing #endif

Every `#if`, `#ifdef`, and `#ifndef` must have a matching `#endif`.

missing '('

`if`, `while`, `do-while`, `switch`, and `for` statements require the expressions following them to be contained inside parentheses. Example:

```
function()
{
    int flag, value, i;

    if (flag) return 1;              /* Legal */
    if flag return 2;                /* Illegal */
    while flag {...}                 /* Illegal */
    do {...} while flag;             /* Illegal */
    switch value {...}               /* Illegal */
    for i = 0; i < 10; i++ {...}     /* Illegal */
}
```

missing ')'

There was a missing close parenthesis. Example:

```
function(i)
int i;
{
     if (i > 4     /* Illegal - missing close parenthesis */
          /* then clause */
}
```

missing ':'

There is a missing colon in a conditional (`?:`) expression or in a `case` or `default` label. Example:

```
function(flag)
int flag;
{
     int i;
     i = flag ? 3 4; /* Illegal - Missing colon */
     switch(i){
     case 3          /* Illegal - Missing colon */
          return i+5;
     default         /* Illegal - Missing colon */
          return i+6;
     }
}
```

missing ';'

A semicolon was expected, but one was not found. Sometimes it is not possible for LightspeedC to determine that a semicolon was missing which will result in the error message **syntax error**.

missing ']'

A closing bracket was expected to match an open bracket, but one was not found.

no members defined

A `struct/union` was defined without members. Example:

```
struct memberless_struct{ };     /* Illegal */
union memberless_union { };      /* Illegal */
```

parameter list is inappropriate here

A declaration involving a function must not have a parameter list, except when a function is being defined. Example:

```
int function(illegal_parameter);            /* Illegal */
int (*ptr_to_function)(illegal_parameter); /* Illegal */
```

pascal argument wrong size

The call to a built-in Toolbox or OS function had a non-integral argument of the wrong size. Example:

```
function()
{
    Rect r;
    Point p;
    ...
    FrameRect(r);    /* Illegal - Should pass ptr to Rect */
    FrameRect(&r);   /* Legal - Correct call to FrameRect */
    FrameRect(4);    /* Legal but wrong - 4 gets cast to ptr */

    SetPt(&p,10L,3); /* Legal - 10L gets cast to an int */
    SetPt(p,4,5);    /* Legal but wrong- sizeof(p)==sizeof(&p)*/
    SetPt(&p, &p, 5);/*Illegal - ptr won't be cast to int */
    SetPt(&p, p, 5); /* Illegal - struct won't be cast to int */
}
```

pointer types do not match

Incompatible pointers were used in an assignment, comparison, or subtraction. In LightspeedC, pointers are more strictly typed than some other C compilers. In the case of assignment and comparison the two pointer types must match or be of type `void *`. In the case of subtracting two pointers, the types must match and not be `void *`. Of course, pointers may be cast to force compatiblity. Example:

```
function()
{
     char *char_ptr;
     int *int_ptr;
     void *void_ptr;
     int result;
     long difference;

     char_ptr = int_ptr;                        /* Illegal */
     char_ptr = (char *)int_ptr;                /* Legal */
     void_ptr = int_ptr;                        /* Legal */
     int_ptr = void_ptr;                        /* Legal */
     result = (char_ptr == int_ptr);            /* Illegal */
     result = ((int *)char_ptr == int_ptr);     /* Legal */
     result = (char_ptr == (void *)int_ptr);    /* Legal */
     result = (void_ptr == int_ptr);            /* Legal */
     difference = char_ptr - int_ptr;           /* Illegal */
     difference = char_ptr - (char *)int_ptr;   /*   legal */
}
```

pointer required

A pointer was implied by an operator, but there was no pointer. Example:

```
function()
{
     int x, result;
     result = *x;            /* Illegal - x is not a pointer */
     result = x->a_member;   /* Illegal - x is not a pointer */
}
```

recursive #include or preprocessor overflow

The most likely cause of this error is an `#include` file has `#included` itself directly or indirectly. Another possibility is deeply nested `#ifdefs` or macro invocations. Example:

```
#define name_1    0
#define name_2    name_1
...
#define name_54   name_53

int i = name_54;
/* Deeply nested macro overflows preprocessor */
```

```
#ifdef name_1
#ifdef name_2
...
#ifdef name_54
/* Deeply nested #ifdef overflows preprocessor */
```

redefinition of existing struct/union/enum

A `struct/union/enum` with the same tag was declared twice. Only one defining instance is allowed. `struct`, `union`, and `enum` tags share the same name space. Example:

```
struct Rumpelstiltskin{
    int member;
};
/* The following are illegal only because the tag   */
/* Rumpelstiltskin has been used previously         */

/* Illegal */
struct Rumpelstiltskin {
    int member;
};

/* Illegal - union name conflicts with previous struct name */
union Rumpelstiltskin {
    void *where_prohibited;
    long l;
};

/* Illegal - enum name conflicts with previous struct name */
enum Rumpelstiltskin {
    Jakob_Ludwig_Karl_Grimm,
    Wilhelm_Karl_Grimm
};
```

required array bounds missing

No size was specified for an array when one was needed to determine data storage space. The size can be explicit or implicit through the use of an initializer. Example:

```
extern char a[];    /* Legal - bounds not needed */
int i = sizeof(a); /* Illegal - need to know size */
function(b)
char b[];        /* Legal - bounds not needed */
{
    char c[];    /* Illegal - auto array needs bounds */
}
```

stack frame too large

The local and temporary variable stack space required by a function exceeded 32768 bytes.

statements nested too deeply

LightspeedC allows nesting of at least 20 statements. `else ifs` do not introduce nested `if` statements and can be put together in arbitrarily long chains. Example:

```
/* maximum nesting is 20 */
if (condition_1) ... if (condition_20) {/* then clause */}

/* arbitrary number of else-ifs can be chained */
if (condition_1)          {/* then clause 1 */}
else if (condition_2)     {/* then clause 2 */}
...
else if (condition_many) {/* then clause many */}
```

struct/union too large

The declared `struct/union` exceeded 32768 bytes of data storage.

switch value must be integral

A switch expression must be of type `char`, `int`, or `long` or an `unsigned` variant. Example:

```
char *p;
float f;
switch(p){...}      /* Illegal - switch of pointer */
switch(f){...}      /* Illegal - switch of float */
```

syntax error

There was a syntax error. Some common errors: too many `}`s, label without `:`, malformed expressions.

there are no void objects!

Declarations provide storage space for the variable declared. It is an error to have a variable that has no storage space. It is legal to have a pointer to `void`; this is a C idiom for a generic pointer. Example:

```
void nugatory;          /* Illegal */
void *generic_pointer; /* Legal */
```

too many formal parameters

LightspeedC allows you to have up to 25 formal parameters in a function definition. Example:

```
f(arg1, arg2..., arg25, arg26)  /* one too many args */
{
}
```

too many initializers

The number of initialization values exceeds the expected number of data items specified in the declaration of the data structure. Example:

```
char *directions[4] =
    {"north", "east", "south", "west", "lost" }; /* Illegal */
struct { int a,b,c; } x[2] = { 1, 2, {3, 4 } };  /* Illegal */
```

too many macro parameters

Macros are allowed to have up to 25 arguments. Example:

```
#define macro(arg1, arg2, ..., arg26)  /* one too many args */
```

undefined enumeration

An enumeration declaration refers to an enumeration tag that has not been defined. Example:

```
enum unknown colors;  /* Illegal */
```

undefined struct/union

A `struct/union` must be defined before an instance of it can be declared. However, a pointer to an undefined `struct/union` is legal since the size of the pointer is known and the size of undefined `struct/union` is not needed. Example:

```
struct not_previously_defined s;          /* Illegal */
struct not_previously_defined *p;         /* Legal */
int i = sizeof(p);                        /* Legal */
int j = sizeof(*p);                       /* Illegal */
struct link_list_element{
    struct link_list_element *next;       /* Legal */
    struct link_list_element recursive;   /* Illegal */
};
```

unexpected end-of-file

End of file was reached before a C language construct was completed. Example:

```
main(

/* EOF - end of file encountered before close parenthesis */
```

unions may not have bitfields

`unions` may not have bitfields. However, they may include `structs` that have bitfields. Example:

```
union {
    int i;
    unsigned int bits : 5;    /* Illegal */
}union_1;

typedef struct { unsigned int bits : 5; } bitfield_type;
union {
    int i;
    bitfield_type b;          /* Legal */
}union_2;
```

unknown struct/union member '*symbol*'

In a . or -> expression either the left operand was not a `struct/union`, or the right operand was not the name of a member of the type of the left operand. Example:

```
function()
{
    int non_struct, *int_ptr;
    struct s1_struct{ int member_1; }s1, *p1;
    struct s2_struct{ int member_2; }s2, *p2;

    /* These are all illegal */
    s1.non_member;        /* non_member is not in s1_struct */
    p1 -> non_member;     /* non_member is not in s1_struct */
    non_struct.member_1; /* non_struct is not a struct */
    int_ptr -> member_1; /* int_ptr is not a struct pointer */
    s1.member_2;          /* member_2 is not in s1_struct */
}
```

unterminated comment

End of file was reached before end of comment was detected.

unterminated quote

Either a character constant or a string constant is missing its end quote. Example:

```
function()
{
    long file_type;

    /* Illegal - missing " after world */
    function("hello world);

    /* Illegal - missing ' after TEXT  */
    file_type = 'TEXT;

    /* Legal                           */
    file_type = 'TEXT';
}
```

use of struct/union/enum does not match declaration

A `struct/union/enum` tag was used in a declaration that conflicts with the orginal `struct/union/enum` tag declaration. `struct/union/enum` tags share the same name space. Example:

```
struct Rumpelstiltskin {
    int member;
};
union Rumpelstiltskin anIllegalUnion;  /* Illegal */
enum Rumpelstiltskin anIllegalEnum;    /* Illegal */
```

void function must not return a value

Example:

```
void in_partners_suit(condition)
int condition;
{
    if (condition)
        return;     /* Legal */
    else
        return(1); /* Illegal */
}
```

wrong number of arguments to '*symbol*'

A call to the built-in Macintosh Toolbox or OS function *symbol* was made with the wrong number of parameters. Example:

```
SetPt(&aPoint, 3, 4, 8);  /* one too many arguments */
```

wrong number of arguments to macro '*symbol*'

A macro was called with the wrong number of arguments. Example:

```
#define twice(x)   (x+x)
function()
{
    int i;
    i = twice(3,4); /* Illegal - macro called with 2 args */
}
```

zero-sized object

An operator was used illegally on a zero-sized object. Example:

```
void vacuous()
{
    int i;
    void *nowhere, *erewhon;

    i = sizeof( vacuous() );     /* Illegal */
    i = nowhere - erewhon;       /* Illegal */
}
```

Appendix E
Data Sheet

General Product Description

Complete C language (*K&R* with Harbison & Steele extensions), production programming environment for the Macintosh. Macintosh-style user interface with integrated multi-file text editor, high performance native code compiler, ultra-fast linker, Auto-Make facility and full Macintosh Toolbox/OS and UNIX-compatibility library support.

Summary of Language Features

Complete implementation of the C language as defined by Kernighan and Ritchie (*K&R*) incorporating more recent features added to C, such as the `void` and `enum` types, as described in *C: A Reference Manual*, by Harbison and Steele as well as features for Pascal support required by the design of the Macintosh.

Implementation of Data Types

`char`	8 bits
`short int`	16 bits
`int`	16 bits
`long int`	32 bits
`float`	32 bits
`short double`	64 bits
`double`	80 bits

The range of floating point numbers expressed by `double` is $\pm 10^{\pm 4932}$. All sizes of `int` (including `char`) may be `unsigned`.

Standard Features Not Always Found in C Implementations

- IEEE standard floating point support. All floating-point arithmetic is performed at extended precision.
- Enumeration specifiers.
- Bitfield operations, including initialization of bitfields.
- Register variables.
- Post-*K&R* operations on structures (structures can be assigned, passed as arguments, and returned from functions).
- The `void` type is supported for functions that do not return a value. In addition, `void *` is supported as an "anonymous" pointer type.
- Extensive UNIX compatibility library support is provided (including `stdio`). Full sources are included.

Additional Language Features for Macintosh Support

- Support for Pascal calling conventions. This allows C functions to call (and be called by) the Pascal-based Macintosh Toolbox and Operating System routines.
- Support for all Macintosh Operating System and Toolbox calls described in *Inside Macintosh*, including those that are listed as [Not in ROM]. There is in-line support for all stack-based Toolbox and OS calls. (Other Toolbox calls are provided in libraries.)
- Actual integer arguments to Toolbox and OS routines are automatically coerced to the right size, eliminating a common source of errors.
- Any function definition declared `pascal` will expect to be called using Pascal calling conventions. Types in Pascal have equivalent types in C as shown below:

Pascal	C
`Boolean`	`char`
`char`	`int` (not `char`!)
`integer`	`int`
`longint`	`long`
`record`	`struct`
`str255`	`char *`
`ptr,handle,etc.`	any `*` type, or `long`

- Pascal-style string literals are supported.

Text Editor Features

- Standard Macintosh-style cut-and-paste, multi-file screen editor.
- File size limited only by available memory.
- Exceptionally fast response to typing and editing operations, especially for large files.
- Auto-indent, tabs and Shift Left/Right by tab stop for complete program formatting control.
- Multi-file search and replace; single file global replace.
- Open file named by a selection for quick access to `#include` files.
- "Save a copy as..." for convenient backup of a variant source file.

Compiler Features

- Extremely fast compilation speed. RAM-based compilation speed ranges from 250 to 500 lines per second depending on the expression density of lines. (See Appendix F for more detail.)
- Superior code generation. Standard sieve benchmark executes in 5.65 seconds, requires 358 bytes. (See Appendix F for more detail.)
- Over 80 distinct error messages.
- Compiler error automatically opens file and puts you into editor with caret on offending line.
- Capable of generating code for Macintosh non-application environments (desk accessories, device drivers, and code resources) and provides built-in support for an alternate global area where appropriate.
- For applications, strings do not occupy space in either code or data segments.
- 32K available for directly referenced global variables.
- Jump Table space for over 4000 intersegmentally referenced function calls.
- Compiler automatically generates code that prevents dangling intersegment function name references.
- Support for up to 255 distinct code segments in a single project.
- Optionally generates inline symbols which can be used by the Macsbug family of machine-level debuggers.
- Optionally generates calls to code profiler.

Ultra-Fast Linker Features

- Swift. Less than 5 seconds, even for large applications.
- Can build applications, desk accessories, device drivers and code resources.
- Intelligent linker automatically removes unreferenced code in standalone versions.
- Full support for Macintosh-style segmentation.
- Complete support for MDS-generated assembly language *.Rel* file-based libraries.
- No separate link control file required.
- Projects may be included within other projects without limit.

Project Management and Auto-Make Facility Features

- Project window shows each source file, library or included project along with the segment it has been put in and its maximum object size after compilation or loading.
- Easy to add and remove source files, libraries or included projects from a project.
- Easy to open source files for editing directly from the project window display. Easy to open `#included` files directly from an editing window.
- A source file, library or included project can be moved from one segment to another by simply dragging its entry in the project window.
- "Get Info" feature shows more detailed information on segments and object components.
- Auto-Make automatically tracks changes to source, `#include`, and library files in a project and can display source files that may need to be recompiled or libraries that may need to be reloaded.
- Auto-Make can use its tracking information to automatically bring a project up to date and run it or build a standalone version of it with a single command.
- "Use Disk" feature enables Auto-Make to update its tracking information for files that may have been modified outside of the LightspeedC environment.
- Handy Auto-Make window allows you to arbitrarily select any combination of multiple source files for recompilation and libraries for reloading.
- All automatic operations have convenient manual overrides.
- Default settings can be remembered for future sessions.

Separate Utilities Supplied

- "Transfer..." menu item provides quick access to utilities.
- *.Rel* file conversion utility for linking with MDS assembly language libraries.
- File compare utility.
- Code profiler.
- Maxbug and MacXLbug machine language debuggers.
- RMaker resource compiler utility.
- ResEdit resource editor utility.

Appendix F
Benchmarks

Code Performance Benchmarks

(Source code from BYTE Magazine, Nov. 1985)

		Consulair (MacC V4.0)	Aztec (V1.06G)	Megamax (V2.1)	LightspeedC (V0.40)
fib	size (bytes)	482	488	466	318
	time (secs.)	32.85	24.77	27.08	25.13
float	size (bytes)	872	930	924	798
	time (secs.)	275.40	289.72	289.02	289.37
pointer	size (bytes)	304	298	326	210
	time (secs.)	31.78	25.57	31.07	20.10
sieve	size (bytes)	438	432	510	344
	time (secs.)	7.75	5.88	6.45	5.65

Large Program Benchmark

(XLISP 1.4, approximately 16.5K source lines)

	Consulair (MacC V4.0)	Aztec (V1.06G)	Megamax (V2.1)	LightspeedC (V0.40)
Generated code size (in bytes)	36770	34566	37698	33870
Program build time (in secs.)				
a. Compile the world	887	654	354	194
b. link-to-run	99	49	95	5
(includes launch)	---	---	---	---
Total program build	986	703	449	199
Turnaround time (in secs.) (time to change one source file and relink-to-run)	159	108	127	9

(All benchmarks performed on a 512K Macintosh with 10Mb HyperDrive.)

Appendix G
The Code Profiler

The LightspeedC system supplies a profiling utility that will provide a function-by-function accounting of relative execution times. Modules which intend to use the profiler must be compiled with the **Profile** option from the **Options** menu. This will insert two extra instructions after the `LINK` at the start of each function. These instructions push the address of a Pascal-style string onto the stack, then call the profiler runtime system. The runtime system handles the calls generated at the start of each profiled function and intercepts the return address for that function to calculate an execution time for each function.

Upon program termination a report will be written to the file `stdout` which contains the following items for each function profiled:

- the minimum amount of time spent in the routine;
- the maximum amount of time spent in the routine;
- the average amount of time spent in the routine;
- the percentage of the profiling period spent in this routine (the profiling period is the total time accumulated in routines compiled with the profile option); and,
- the total number of times that routine was called.

Since the output is written to `stdout`, it may be redirected using the *unix main.c* module and the standard UNIX command line conventions. (Refer to "Converting from UNIX" in Chapter 10, "Achieving Lightspeed", for instructions on how to use this feature.)

The time measurements may be done in one of two ways. By default, the accumulated times of all routines called by a given routine are not incorporated into the time charged to the routine itself. To override this default, comment out the `#define` for `SINGLE` in *profile.c*, and recompile the profile code. Warning: When compiling the C profile code, DO NOT HAVE THE **Profile** OPTION SELECTED! This will result in an infinite loop at runtime.

Time is measured in the profiler using the `TickCount` trap. This provides a granularity of 1/60 of a second. If higher granularity is needed, then a different clock must be used.

The profiling may be turned on and off by the user. An external integer `_profile` controls this feature. If `_profile` is non-zero then profiling is defined to be "on". If the variable is zero, then the profiling is defined to be "off". This allows the user explicit runtime control over what is profiled.

As a side effect of the profiler, an indented call-tree can be written to `stdout` as the program is running. The names of routines are printed out as they are called. This feature is normally "off", but may be turned on by setting the external integer variable `_trace` to a non-zero value.

Using the profiling package is simple. Just compile those files that you wish to monitor with the **Profile** option, and include the *profile* and *stdio* libraries in your project.

Routines

`DumpProfile`	write to `stdout` the current state of the profiling. This routine can be called by the user at any time. It is called automatically on completion of the application.
`_profile_`	assembly language routine that is called by the code generated by the **Profile** option of LightspeedC. It changes the return address for the routine to be `_profile_exit_`.
`__profile`	C code to do calling tree and profile management.
`_profile_exit_`	assembly language routine that marks the end of a profiled function, cleans up the stack after profiling, and preserves the user's function result.
`__profile_exit`	C code to compute profile statistics on a function exit. Returns the real return address for the user's function.

Variables

`_profile`	if non-zero, enables profiling
`_trace`	if non-zero, enables call-tree generation

Index to Functions

abort (unix) 13-10
abs (math) 13-75
acos (math) 13-74
allmem (storageu) 13-11
asin (math) 13-74
atan (math) 13-74
atan2 (math) 13-74
atof (unix) 13-12
atoi (unix) 13-12
atol (unix) 13-12
bldmem (storageu) 13-13
calloc (storage) 13-14
CallPascal (MacTraps) 13-16
CallPascalB (MacTraps) 13-16
CallPascalL (MacTraps) 13-16
CallPascalW (MacTraps) 13-16
ceil (math) 13-76
cfree (storage) 13-17
cgetpid (unix) 13-18
cgets (stdio) 13-19
circle (unix) 13-20
clalloc (storage) 13-21
clearerr (stdio) 13-22
close (unix) 13-23
_closeall (stdio) 13-24
clrerr (stdio) 13-22
cont (unix) 13-25
cos (math) 13-74
cosh (math) 13-76
cprintf (stdio) 13-26
cputs (stdio) 13-27
creat (unix) 13-28
cscanf (stdio) 13-29
ctime (unix) 13-30
CtoPstr (MacTraps) 13-31
eraseplot (unix) 13-34
_exit (unix) 13-35
exit (unix) 13-35
exp (math) 13-75
fabs (math) 13-76
fclose (stdio) 13-36
feof (stdio) 13-37
ferror (stdio) 13-38
fflush (stdio) 13-39
fgetc (stdio) 13-40
fgets (stdio) 13-41
fileno (unix) 13-42
floor (math) 13-76
fmod (math) 13-75
fopen (stdio) 13-43
fprintf (stdio) 13-45
fputc (stdio) 13-46
fputs (stdio) 13-47
fread (stdio) 13-48
free (storage) 13-49
freopen (stdio) 13-50
frexp (math) 13-75
fscanf (stdio) 13-51
fseek (stdio) 13-52
ftell (stdio) 13-53
fwrite (stdio) 13-54
getc (stdio) 13-55
getch (stdio) 13-56
getchar (stdio) 13-57
getche (stdio) 13-58
getmem (storageu) 13-59
getml (storageu) 13-60
getpid (unix) 13-61
gets (stdio) 13-62
getuid (unix) 13-63
getw (unix) 13-64
gotoxy (stdio) 13-65
isalnum (stdio) 13-32
isalpha (stdio) 13-32
isascii (stdio) 13-32
iscntrl (stdio) 13-32
iscsym (stdio) 13-32
iscsymf (stdio) 13-32
isdigit (stdio) 13-32
isgraph (stdio) 13-32
islower (stdio) 13-32
isodigit (stdio) 13-32
isprint (stdio) 13-32
ispunct (stdio) 13-32
isspace (stdio) 13-32
isupper (stdio) 13-32
isxdigit (stdio) 13-32
kbhit (stdio) 13-66
label (unix) 13-67
labs (math) 13-76
ldexp (math) 13-75
line (unix) 13-68
localtime (unix) 13-69
locv (unix) 13-70

log (math) 13-75
log10 (math) 13-75
longjump (unix) 13-114
lsbrk (storageu) 13-71
lseek (unix) 13-72
malloc (storage) 13-73
mlalloc (storage) 13-77
modf (math) 13-75
move (unix) 13-78
movmem (unix) 13-79
onexit (stdio) 13-80
open (unix) 13-81
perror (unix) 13-82
point (unix) 13-83
pow (math) 13-75
printf (stdio) 13-84
PtoCstr (MacTraps) 13-88
putc (stdio) 13-89
putch (stdio) 13-90
putchar (stdio) 13-91
puts (stdio) 13-92
putw (unix) 13-93
qksort (unix) 13-94
qsort (unix) 13-95
rand (math) 13-76
rbrk (storageu) 13-96
read (unix) 13-97
realloc (storageu) 13-98
relalloc (storageu) 13-99
remove (unix) 13-100
rename (unix) 13-101
repmem (unix) 13-102
rewind (stdio) 13-103
rlsmem (storageu) 13-104
rlsml (storageu) 13-105
rstmem (storageu) 13-106
sbrk (storageu) 13-107
scanf (stdio) 13-108
setbuf (unix) 13-112
Set_Echo (stdio) 13-113
setjmp (unix) 13-114
setmem (storageu) 13-115
setnbuf (unix) 13-116
setpid (unix) 13-117
Set_Tab (stdio) 13-118
setuid (unix) 13-119
sin (math) 13-74
sinh (math) 13-76
sleep (unix) 13-120
sprintf (stdio) 13-121
sqrt (math) 13-75
srand (math) 13-76
sscanf (stdio) 13-122
stcarg (unix_strings) 13-123
stccpy (strings) 13-124
stcd_i (unix) 13-125
stch_i (unix) 13-126
stci_d (unix) 13-127
stcis (strings) 13-128
stcisn (strings) 13-129
stclen (strings) 13-130
stcpm (unix_strings) 13-131
stcpma (unix_strings) 13-133
stcu_d (unix) 13-135
Stdio_config (stdio) 13-136
stpblk (strings) 13-137
stpbrk (strings) 13-138
stpchr (strings) 13-139
stpcpy (strings) 13-140
stpsym (unix_strings) 13-141
stptok (unix_strings) 13-143
strcat (strings) 13-144
strchr (strings) 13-145
strcmp (strings) 13-146
strcpy (strings) 13-147
strcspn (strings) 13-148
strlen (strings) 13-149
strncat (strings) 13-150
strncmp (strings) 13-151
strncpy (strings) 13-152
strpbrk (strings) 13-153
strpos (strings) 13-154
strrchr (strings) 13-155
strrpbrk (strings) 13-156
strrpos (strings) 13-157
strspn (strings) 13-158
stscmp (string) 13-159
stspfp (unix) 13-160
tan (math) 13-74
tanh (math) 13-76
tell (unix) 13-161
time (unix) 13-162
toascii 13-163
toint (stdio) 13-164
_tolower (stdio) 13-165
tolower (stdio) 13-165
_toupper (stdio) 13-166
toupper (stdio) 13-166
ttyn (unix) 13-167
ungetc (stdio) 13-168
ungetch (stdio) 13-169
unlink (unix) 13-170
vcprintf (stdio) 13-171
vfprintf (stdio) 13-171
vprintf (stdio) 13-171
vsprintf (stdio) 13-171
write (unix) 13-172

Index

"(",missing in if, do-while, switch or for statement D-16
")",missing in if, do-while, switch or for statement D-17
":",missing D-17
";"missing D-17
"]" missing D-17

A

abort() 13-10
abs() 13-76
acos() 13-74
Add command 3-1, 3-2, 6-2
Add... command 3-1, 3-4, 3-5, 3-11, 6-2, 6-12, 6-13, 5-4
additive operator 12-6
address error 7-3
address, initialization to illegal in non-application D-13
Address Register (A) Macsbug command A-2
address registers, dump of using Macsbug 7-2
address-of operator D-1
allmem() 13-11
allocate a block of memory 13-11, 13-13, 13-14, 13-22, 13-73, 13-77
ALRT resource B-3
append to string 13-143, 13-150
arguments, wrong number of D-25
array
  bounds, illegal D-7
  error attempting to initialize D-3
  illegal operations on D-8
  of functions, error D-2
  of void, error D-2
  required bounds missing D-20
asin() 13-74
assembly language 7-1, 9-7 10-5
assignment operator 12-6
atan() 13-74
atan2() 13-74
atof() 13-12
atoi() 13-12
atol() 13-12
A-Trap Remembered (AR) Macsbug command 7-10
A-Trap Spy (AS) Macsbug command 7-12
A-Traps, in Macsbug 7-6
attach private buffer to file 13-112
Auto-Make 1-2, 2-1, 3-11, 4-4, 5-1, 5-2, 6-11, 6-14 - 6-16
Aztec C, 10-1 - 10-6

B

benchmarks F-1
bitfields 12-7, E-2
  illegal in union D-23
  illegal size for D-10
  illegal type for D-11
bldmem() 13-13
BNDL resource B-4
BOOLEAN parameters, C equivalent to 9-6
break statement, outside of loop or switch D-2
Breakpoint (BR) Macsbug command 7-5, A-2
BUFSIZ 13-4
Build Application... command 3-13, 5-2, 5-5, 6-10, 6-11, 6-13, 6-16, 8-1 - 8-2, 9-15
Build Code Resource... command 9-15
Build Desk Accessory... command 9-15
Build Device Driver... command 9-15
Build Library... command 5-4 - 5-6, 6-10 - 6-13, 6-16, 9-16

C

calling conventions 9-2, 9-7
calloc() 13-14
CallPascal() 9-7, 13-16
CallPascalB() 13-16
CallPascalL() 13-16
CallPascalW() 13-16
case statement 12-9
  duplicate in switch D-5
  keyword out of place D-3
cast, illegal D-7
ceil() 13-76
cfree() 13-17
cgetpid() 13-28
cgets() 13-19
change size of allocated memory region 13-98, 13-99
character constants 12-2, 12-4
Check Link command 6-10 - 6-11
Check Syntax command 6-2, 6-12, 6-13
circle() 13-20

calloc() 13-21
Clear Breakpoints (CL) Macsbug command 7-5, A-2
Clear command 4-5, 6-6
clear last error flag of stream 13-23
close() 13-23
close
  all open files 13-24
  all open files and exit 13-35
  file 13-36
  file number fn 13-24
_closeall() 13-24
Close command (Project Menu) 3-13, 6-3, 6-10, 6-16
Close command (File Menu) 3-7, 3-13, 4-10, 6-2, 6-3
clrerr() 13-22
CNTL resource B-4
CODE, component of Lightspeed program 9-15
code resources 9-12 - 9-14, D-2
comment, unterminated D-24
Compact Project option 6-15, 6-16
compare strings 13-146, 13-151, 13-159
Compile command 1-1, 1-2, 3-1, 3-6 - 3-10, 5-1, 5-2, 6-12, 6-13
compiler options (see Options Menu)
compiling 1-1, 3-1, 3-5, 3-7 - 3-11, 5-1, 5-2, 6-10 - 6-14
components of Lightspeed programs 9-15
concatenate string with process id number 13-19
conditional operator 12-6
Confirm Auto-Make option 5-2, 6-11, 6-15, 6-16
Confirm Saves option 3-13, 6-3, 6-10, 6-15, 6-16
_console 13-4
console 13-4
  formatted print to 13-27
  get string from 13-20
  move cursor on 13-65
  put a character on screen 13-91
  put string to 13-28
  read text and data from 13-30
  set tab width of 13-118
  set to echo keyboard input 13-113
constants 12-2
  constant expressions 12-10
  integer constant too large D-13
Consulair, 10-1 -10-6
cont() 13-25
continue statement, outside of loop D-4
conversion
  arithmetic 12-4
  ASCII hex value to integer 13-126
  ASCII integer value to integer 13-125
  C string to Pascal string 13-17
  character to lower case 13-165
  character to upper case 13-166
  hexadecimal character to integer 13-164
  integer to ASCII character 13-163
  integer to ASCII string 13-127, 13-135
  long output conversion 13-70
  Pascal string to C string 13-88
  string to number 13-12
Convert (CV) Macsbug command 7-12
copy a block of memory 13-79
Copy command 4-5, 4-6, 4-7, 6-6
copy pattern into memory 13-102
cos() 13-74
cosh() 13-76
count number of characters in string 13-123
cprintf() 13-26
cputs() 13-27
creat() 13-28
cscanf() 13-29
ctime() 13-30
CtoPstr() 13-31
ctype functions 13-32
CURS editor, in ResEdit C-3
cursor
  move on console screen 13-65
  move to x,y position 13-78
Cut command 4-5, 6-6

D

DATA, component of Lightspeed program 9-15
Data Register (D) Macsbug command A-2
data registers, dump of using Macsbug 7-2
data types, hardware characteristics of 12-3, E-1
Debugger(), 7-9
debugging 2-1, 3-8, 6-15, 6-16, 7-1 - 7-6, A-1 - A-3
DebugStr(), 7-9
declaration, invalid D-13
declarator too complex D-5
default keyword, outside of switch D-5
deleting text 4-4 - 4-5, 6-6 - 6-7
desk accessories 9-8 - 9-12, E-3
  fields in header of 9-9
  returning from 9-9
device drivers 9-8 - 9-12
  calling 9-9
  fields in header of 9-9
  returning from 9-9
Display Memory (DM) Macsbug command 7-3, A-2
DITL editor, in ResEdit C-4
DITL resource B-4
DLOG editor, in ResEdit C-6
DLOG resource B-5
drawing
  circle 13-21
  erase plot 13-34
  line from cursor to x,y 13-24
  line from x1,y1 to x2,y2 13-68
  place point at location x,y 13-83
DumpProfile() G-1
dynamically allocate memory 13-59, 13-60, 13-71, 13-107

E

Edit Menu
  Clear command 4-5, 6-6
  Copy command 4-5, 4-6, 4-7, 6-6
  Cut command 4-5, 6-6
  Paste command 4-5, 4-7, 6-7
  Set Font... command 6-7

Set Tabs... command 4-6, 6-7
Shift Left command 4-6, 6-7
Shift Right command 4-6, 6-7
Undo command 4-5, 6-6
edit window
changing size of 4-2
moving 4-3
scrolling 4-2
editing (see Edit Menu)
#else statement 12-10
illegal D-7
end of file (see EOF)
#endif statement 12-10
illegal D-7
missing D-16
enums, 12-3, 12-7
name space in 12-9
redefinition of existing D-20
undefined D-22
use does not match declaration D-25
EOF 13-4
unexpected D-23
equality operator 12-6
eraseplot() 13-34
errno 13-4
Error messages D-1 - D-26
error number, print user's string and 13-81
_exit() 13-35
exit and call system debugger 13-10
Exit to Application (EA) Macsbug command 7-9, A-4
Exit to Shell (ES) Macsbug command 7-3, 7-9, A-4
exiting Lightspeed C 3-2, 3-13, 6-5, 6-16
exp() 13-75
expression too complex D-5
expressions, primary 12-5

F

fabs() 13-76
fclose() 13-36
feof() 13-37
ferror() 13-38
fflush() 13-39
fgetc() 13-40
fgets() 13-41
FILE 13-8
_file 13-8
file descriptor 13-8
file handling routines
attach private buffer to file 13-112
close all open files 13-25
close all open files and exit 13-35
close file 13-36
close file number fn 13-24
create new file 13-29
exit and call system debugger 13-10
format and make output into string 13-121
format and print to file 13-45
get file number 13-42
get next character from file 13-55
move I/O position to start of file 13-103
move to a different point inside file 13-52
open file 13-43, 13-81
parse filename pattern 13-160
print and format text and data to stdout 13-84
put character into file 13-46, 13-89
put string into file 13-47
read characters from file 13-48, 13-99
read data and text from 13-51
redirect output to different file 13-50
remove filename from directory 13-100
rename a file 13-101
return position within file 13-53, 13-161
scan and format input 13-51
set a file to be unbuffered 13-116
unlink file from directory 13-170
variable format and print 13-171
write block of characters to a file 13-54, 13-172
File Menu
Close command 3-7, 3-13, 4-10, 6-2, 6-3
New command 3-1, 4-1, 4-10, 6-2
Open... command 3-1, 3-4, 4-1, 4-10, 6-2
Open Selection command 4-3, 4-4, 4-5, 6-2
Page Setup... command 4-9, 6-2, 6-4 - 6-5
Print... command 4-9, 6-2, 6-5 - 6-6
Quit command 3-2, 3-12, 3-13, 6-2, 6-5, 6-16
Revert command 4-10, 6-2, 6-4
Save a Copy As... command 3-6, 4-10, 6-2, 6-4
Save As... command 3-6 - 3-7, 3-9, 4-10, 6-2, 6-4
Save command 3-6 - 3-7, 3-9, 4-10, 6-2, 6-4
Transfer... command 5-5, 6-2, 6-5, 6-16
file number 13-8
file pointer 13-8
file specification 4-4 - 4-5
fileno() 13-42
Find Again command 4-6 - 4-8, 6-7 - 6-9
Find... command 4-6 - 4-8, 6-7 - 6-9, 6-15
Find in Next File command 4-8 - 4-9, 6-7, 6-9
finder 3-11
floating point, 12-4
constants 12-2
illegal operations on D-8
range of E-1
floor() 13-76
flush buffer of stream 13-39
fmod() 13-75
FONT editor, in ResEdit C-4
fonts, setting default from program 13-136
fopen() 13-43
format and make output into string 13-121
format and print to file 13-45
fprintf() 13-45
fputc() 13-46
fputs() 13-47
fread() 13-48
free() 13-49
free memory 13-18
FREF resource B-5
freopen() 13-50
frexp() 13-75
fscanf() 13-51

fseek() 13-52
ftell() 13-53
functions
  accepting a variable number of arguments 9-3
  entry 9-2, 9-5
  exit 9-2, 9-5
  external 12-9
  illegal operations on D-8
  invalid definition D-14
  parameter list in D-18
  returning array D-6
  returning function D-6
  returning struct, union, or double 9-3
fwrite() 13-54

G

general resource editor C-8
get
  character from stream 13-40
  file number 13-42
  next character from file 13-55
  next character from keyboard, 13-56, 13-58
  string from console 13-20
  string from stream and put in array 13-41
Get Info command 6-12 - 6-13
getc() 13-55
getch() 13-56
getchar() 13-57
getche() 13-58
getmem() 13-59
getml() 13-60
getpid() 13-61
gets() 13-62
getuid() 13-63
getw() 13-64
global symbols 9-7, 9-10, E-3
GNRL resource B-6
Go (G) Macsbug command A-2
Go Until (GT) Macsbug command 7-5, A-2
gotoxy() 13-65

H

heap 7-7
Heap Check (HC) Macsbug command 7-8, A-4
Heap Dump (HD) Macsbug command 7-8, A-4
Heap Exchange (HE) Macsbug command A-5
Heap Total (HT) Macsbug command 7-8, A-4
HFS (see Hierarchical File System)
Hierarchical File System (HFS) 2-1, 4-3, 4-4, 6-3

I

ICN# editor, in ResEdit C-5
identifiers 12-1
  length and capitalization 10-4
#ifdef statement 12-10
  deeply nested D-19
  identifier does not match D-6
Ignore Case option 6-15
#include files 1-2, 2-1, 3-4, 3-6, 4-3 - 4-5, 5-1, 5-2, 6-2, D-3, D-19
indentation 4-5, 4-6, 6-6 - 6-7
indirect calls to Pascal functions 9-7
initialize memory 13-115
initializers, too many D-22
insertion point, in text editor 4-2, 6-7
installing
  LightspeedC 2-1
  Macsbug A-1
Instruction Disassembly (ID) Macsbug command 7-3, A-3
Instruction Listing (IL) Macsbug command 7-3, A-3
integer constants 12-2
isalnum() 13-32
isalpha() 13-32
isascii() 13-32
iscntrl() 13-32
iscsym() 13-32
iscsymf() 13-32
isdigit() 13-32
isgraph() 13-32
islower() 13-32
isodigit() 13-32
isprint() 13-32
ispunct() 13-32
isspace() 13-32
isupper() 13-32
isxdigit() 13-32

J

JUMP, component of Lightspeed program 9-15
jumptable space E-3

K

kbhit() 13-66
keyboard
  get next character from and don't echo 13-56
  get next character from and echo to screen 13-58
  push character back to keyboard input stream 13-169
  test whether keyboard hit 13-66
keywords 12-1
K&R, differences from 12-1 - 12-12

L

label() 13-67
labs() 13-76
ldexp() 13-75
lexical scope 12-9
libraries 9-16
  math 13-1, 13-74 - 13-76
  stdio 13-1
  storage 13-1

storageu 13-1
strings 13-1
unix 13-1
unix_strings 13-1
library files 1-1, 1-2, 3-4 - 3-7, 3-11, 5-1 - 5-2, 5-5, 6-11 - 6-14
loading 3-7, 3-11, 5-1 - 5-2, 5-5, 6-12 - 6-14
types of 5-5, 6-13
line() 13-68
linking 1-1, 1-2, 2-1, 5-5, 6-10 - 6-13
Load Library command 3-10, 5-2, 6-12 - 6-13
localtime() 13-69
locv() 13-70
log() 13-75
log10() 13-75
longjump() 13-112
lsbrk() 13-71
lseek() 13-72
lvalue, required D-15

M

Mac C (see Consulair)
macro calls
incomplete D-12
name already #defined D-16
parameter symbol appears more than once D-16
too many parameters D-22
wrong number of arguments to D-25
Macsbug 1-2, 2-1, 3-13, 6-11, 7-1, A-1 - A-6, E-3
Address Register (A) command A-2
A-Trap Remembered (AR) command 7-10
A-Trap Spy (AS) command 7-12
Breakpoint (BR) command 7-5, A-2
Clear Breakpoints (CL) command 7-5, A-2
Convert (CV) command 7-12
Data Register (D) command A-2
Display Memory (DM) command 7-3, A-2
Exit to Application (EA) command 7-9, A-4
Exit to Shell (ES) command 7-3, 7-9, A-4
Go (G) command A-2
Go Until (GT) command 7-5, A-2
Heap Check (HC) command 7-8, A-4
Heap Dump (HD) command 7-8, A-4
Heap Exchange (HE) command A-5
Heap Total (HT) command 7-8, A-4
Instruction Disassembly (ID) command 7-3, A-3
Instruction Listing (IL) command 7-3, A-3
Magic Return (MR) command 7-7, A-2
Program Counter (PC) command A-3
Reboot (RB) Command A-4
Set Memory (SM) command 7-5, A-2
Stack Crawl (SC) command 7-11, A-2
Status Register (SR) command A-3
Step (S) command A-2
Step Spy (SS) command 7-1, A-22
Step Until (ST) command 7-5, A-2
Symbols option 6-15, 6-16, 7-2, 7-3, 9-4
Total Display (TD) command A-3
Trace (T) command 7-7, A-2
Where (WH) command 7-12, A-2
MacXLbug (see Macsbug)
Magic Return (MR) Macsbug command 7-7, A-2
Make command 1-2, 3-10, 4-4, 5-1 - 5-4, 6-12 - 6-13, 11-2
malloc() 13-73
match pattern at start of string 13-133
Match Words option 6-15
math library 13-1, 13-74 - 13-76
Maxbug (see Macsbug)
Megamax C, 10-1 - 10-6
memory allocation
allocate a block of memory 13-11, 13-13, 13-14, 13-22, 13-73, 13-77
attach private buffer to a file 13-112
change size of allocated region 13-98, 13-99
copy a block of memory 13-79
copy pattern into memory 13-102
dynamically allocate 13-59, 13-60, 13-71, 13-107
free memory 13-18
initialize 13-115
release block of memory 13-49, 13-104, 13-105, 13-106
reset memory break point 13-96
set a file unbuffered 13-116
memory management 13-3
MENU resource B-6
MiniEdit sample application 3-2, 3-12
mlalloc() 13-77
modf() 13-75
move() 13-78
move
I/O position to start of file 13-103
to a different point inside file 13-52, 13-72
to a point on the screen 13-65, 13-78
movmem() 13-79
multiplicative operator 12-5

N

nesting, errors resulting from D-19, D-21
New... command (Project Menu) 3-1 - 3-2, 6-10
New command (File Menu) 3-1, 4-1, 4-10, 6-2
_NFILE 13-9
non-maskable interrupt 7-1
NULL 13-9

O

object code 1-1, 3-5, 3-10 - 3-11, 5-4
onexit() 13-80
open() 13-81
Open... command (File Menu) 3-1, 3-4, 4-1, 4-10, 6-2, 6-10
Open... command (Project Menu) 3-1, 3-4, 6-10
Open Selection command 4-3, 4-4, 4-5, 6-2
opening
a file for editing 3-1, 4-1, 4-3 - 4-4, 6-2 - 6-3
a file from a C program 13-81
multiple files 4-3 - 4-4
operators 12-5 - 12-6

Options Menu
Compact Project 6-15, 6-16
Confirm Auto-Make 5-2, 6-11, 6-15, 6-16
Confirm Saves 3-13, 6-3, 6-10, 6-15, 6-16
Ignore Case 6-15
Macsbug Symbols 6-15, 6-16
Match Words 6-15
Profile 6-15, 6-16
Save Settings 4-7, 6-15, 6-16
Wrap Around 6-15

P

packages (see code resources)
PACKED ARRAY[1..4] OF CHAR, C equivalent to 9-6
Page Setup... command 4-9, 6-2, 6-4 - 6-5
parameter list, inappropriate use of D-18
parameters, too many D-22
parsing
filename pattern 13-160
first symbol in input string 13-141
first token in input string 13-143
Pascal 9-1
argument wrong size D-18
BOOLEAN parameters 9-6
calling functions indirectly 13-16
function entry 9-4
function exit 9-5
illegal return type for functions D-10
PACKED ARRAY[1..4] OF CHAR 9-6
pascal keyword 9-1, 12-7, E-2
Point 9-6
PROCEDUREs 9-6
RECORD 9-6
string conversion 13-17, 13-87
strings 12-2
type considerations 9-5
VAR parameters 9-5
Paste command 4-5, 4-7, 6-7
pathname (see Hierarchical File System)
perror() 13-82
point() 13-83
Point, passing as argument 9-6, 10-3
pointer
converting 12-10
illegal arithmetic D-10
illegal combination with integer D-9
implied by operator D-19
types do not match D-18
portability, 10-1 - 10-7, 12-10, 13-1
pow() 13-75
preprocessor
format of control lines 12-9
overflow D-19
Print... command 4-9, 6-2, 6-5 - 6-6
printf() 13-84
printing
source files 4-9
text and data to stdout 13-84
user's string and error number 13-83
PROC resource B-6
PROCEDUREs, C equivalent to 9-6
process id number
concatenate string with 13-19
get 13-61
set 13-117
__profile() G-2
profile() G-2
Profile option 6-15, 6-16, 9-4
__profile_exit() G-2
_profile_exit_() G-2
profiler E-3, E-5, G-1, G-2
Program Counter 7-2
Program Counter (PC) Macsbug command A-3
programmer's switch 7-1
project
adding files to 3-1, 3-4, 3-11, 5-4, 6-2, 6-12, 6-13
as a form of library 9-16
bringing up to date 3-6, 3-11, 6-16
changing type 3-11, 3-13, 5-4, 6-10, 6-11
closing 3-12, 6-10, 6-11, 6-16
compacting 6-15, 6-16
creating a new 3-1, 3-2, 6-10
including in another project 6-13
opening 3-1, 6-10
running 3-2, 3-7 - 3-11, 5-1, 5-2, 6-10, 6-11, 6-16
Project Menu
Build Application... command 3-13, 5-5, 5-6, 6-10 - 6-11, 6-16
Build Library... command 5-4 - 5-6, 6-10 - 6-13, 6-16
Check Link command 6-10 - 6-11
Close command 3-13, 6-3, 6-10, 6-16
New... command 3-1 - 3-2, 6-10
Open... command 3-1, 3-4, 6-10
Run command 3-2, 3-7 -3-12, 5-1, 6-10 - 6-11, 6-16
Set Project Type command 3-11, 5-4, 6-10
PtoCstr() 13-88
push character back to keyboard input stream 13-169
put
character into file 13-46, 13-89
string into file 13-47
string to console 13-28
string to stdout 13-92
putc() 13-89
putch() 13-90
putchar() 13-91
puts() 13-92
putw() 13-93

Q

qksort() 13-94
qsort() 13-95
QuickDraw globals 9-11
Quit command 3-2, 3-12, 3-13, 6-2, 6-5, 6-16
quote, unterminated D-24

R

rbrk() 13-96
read() 13-97
read
  characters from file 13-48, 13-99
  data and text from file 13-51
realloc() 13-98
Reboot (RB) Macsbug command A-4
RECORD, C equivalent to 9-6
redirection, of output and input 10-7, 13-50
register variables 12-7
register-saving conventions 9-8
relalloc() 13-99
relational operator 12-6
RelConv 9-8
release block of memory 13-49, 13-104, 13-105, 13-106
remove() 13-100
Remove command 6-12
remove filename from directory 13-100
rename() 13-101
rename a file 13-101
Replace All command 4-7 - 4-8, 6-7, 6-9
Replace and Find Again command 4-7, 6-7, 6-9, 6-15
Replace command 4-7 - 4-8, 6-7, 6-9
repmem() 13-102
ResEdit 1-2, 2-1, 3-13, 6-11, 9-8, C-1 - C-8
reserved words (see keywords)
reset memory break point 13-96
resource files 3-13, 8-1, B-1 - B-7, C-1 - C-8
Resource types
  ALRT B-3
  BNDL B-4
  CNTL B-4
  DITL B-4
  DLOG B-5
  FREF B-5
  GNRL B-6
  MENU B-6
  PROC B-6
  STR B-6
  STR# B-7
  WIND B-7
return
  character to input stream buffer 13-168
  length of string 13-130, 13-149
  next two bytes of stream as integer 13-64
  position within file 13-53, 13-161
return statement 12-9
Revert command 4-10, 6-2, 6-4
rewind() 13-103
rlsmem() 13-104
rlsml() 13-105
RMaker 2-1, 3-13, 6-11, 8-1, 8-3, 9-8, B-1 - B-7
rstmem() 13-106
Run command 3-2, 3-7 - 3-12, 5-1, 6-10 - 6-11, 6-16, 8-1
runtime environment 10-5

S

SANE 12-3, 12-4
Save a Copy As... command 3-6, 4-10, 6-2, 6-4
Save As... command 3-6 - 3-7, 3-9, 4-10, 6-2, 6-4
Save command 3-6 - 3-7, 3-9, 4-10, 6-2, 6-4
Save Settings option 4-7, 6-15, 6-16
saving edits (see File Menu)
sbrk() 13-107
scan and format input 13-29, 13-51, 13-108, 13-122
scanf() 13-108
scratch registers 9-8
scrolling 4-2
search
  for pattern anywhere in string 13-131
  in text editor (see Search Menu and Options Menu)
Search Menu
  Find Again command 4-6 - 4-8, 6-7 - 6-9
  Find... command 4-6 - 4-8, 6-7 - 6-9, 6-15
  Find in Next File command 4-8 - 4-9, 6-7, 6-9
  Replace All command 4-7 - 4-8, 6-7, 6-9
  Replace and Find Again command 4-7, 6-7, 6-9, 6-15
  Replace command 4-7 - 4-8, 6-7, 6-9
search options (see Options Menu)
segmentation 5-1, 5-3 -5-4, 6-13, E-3
selecting text, in text editor 4-5
set a file unbuffered 13-116
Set Font... command 6-7
Set Memory (SM) Macsbug command 7-5, A-2
Set Project Type command 3-11, 5-4, 6-10
Set Tabs... command 4-6, 6-7
setbuf() 13-112
Set_Echo() 13-113
setjmp() 13-114
setmem() 13-115
setnbuf() 13-116
setpid() 13-117
Set_Tab() 13-118
setuid() 13-119
Shift Left command 4-6, 6-7
shift operator 12-6
Shift Right command 4-6, 6-7
sieve benchmark E-3
sign extension 12-3
sin() 13-74
sinh() 13-76
size of types 10-2
sleep() 13-120
sort a table in place 13-94, 13-95
source files 1-2, 3-4 - 3-8, 4-10, 5-1, 5-2, 6-4, 6-12, 6-13
Source Menu
  Add command 3-1, 3-2, 6-2
  Add... command 3-1, 3-4, 3-5, 3-11, 6-2, 6-12, 6-13, 5-4
  Check Syntax command 6-2, 6-12, 6-13
  Compile command 1-1, 1-2, 3-1 3-6 - 3-10, 5-1, 5-2, 6-12, 6-13
  Get Info command 6-12 - 6-13
  Load Library command 3-10, 5-2, 6-12 - 6-13
  Make command 1-2, 3-10, 4-4, 5-1 - 5-4, 6-12 - 6-14
  Remove command 6-12

Run command 3-7 - 3-12, 5-1 - 5-2, 6-10 - 6-11, 6-16
sprintf() 13-121
sqrt() 13-75
sscanf() 13-122
stack 7-7
Stack Crawl (SC) Macsbug command 7-11, A-2
stack frame, too large D-21
Stack Pointer 7-2
Standard Apple Numeric Environment (see SANE)
starting Lightspeed C 3-2
statement, nested too deeply D-21
Status Register 7-2
Status Register (SR) Macsbug command A-3
stcarg() 13-123
stccpy() 13-124
stcd_i() 13-125
stch_i() 13-126
stci_d() 13-127
stcis() 13-128
stcisn() 13-129
stclen() 13-130
stcpm() 13-131
stcpma() 13-133
stcu_d() 13-135
stderr 13-9
stdin 13-9
  get characters from and store in array 13-62
  get next character from 13-57
  scan and format input from 13-109
stdio library 13-1
Stdio_config() 13-136
stdout 13-9
  print and format text and data to 13-84
  put character to 13-91
  put string to 13-92
  variable format and print to 13-171
Step (S) Macsbug command A-2
Step Spy (SS) Macsbug command 7-1, A-22
Step Until (ST) Macsbug command 7-5, A-2
storage class
  invalid D-14
  specifiers 12-7
storage library 13-1
storageu library 13-1
stpblk() 13-137
stpbrk() 13-138
stpchr() 13-139
stpcpy() 13-140
stpsym() 13-141
stptok() 13-143
STR resource B-6
STR# resource B-7
strcat() 13-144
strchr() 13-145
strcmp() 13-146
strcpy() 13-147
strcspn() 13-148
stream
  clear last error flag of 13-23
  flush buffer of 13-39
  get character from 13-40
  get string from stream and put in array 13-41
  push character back to keyboard input stream 13-169
  return character to input stream buffer 13-168
  return next two bytes of as integer 13-64
  test whether at end of 13-37
  test whether I/O error flag is set for 13-38
  write word as 2 byte number to 13-93
strings 12-2
  append to 13-143, 13-150
  compare 13-146, 13-151, 13-159
  concatenate with process id number 13-19
  convert integer to ASCII string 13-127, 13-135
  convert to number 13-12
  copy 13-124, 13-140, 13-147, 13-152
  count number of characters in 13-123
  format and make output into 13-121
  get from console 13-20
  get from stream and put in array 13-41
  library 13-1
  match pattern at start of 13-133
  parse first symbol in input 13-141
  parse first token in input 13-143
  Pascal 12-2
  print user's string and error number 13-83
  put into file 13-47
  put to console 13-28
  put to stdout 13-92
  return length of 13-130, 13-149
  scan and format input from 13-122
  search for pattern anywhere in 13-131
  write string at cursor position 13-67
strlen() 13-149
strncat() 13-150
strncmp() 13-151
strncpy() 13-152
strpbrk() 13-153
strpos() 13-154
strrchr() 13-155
strrpbrk() 13-156
strrpos() 13-157
STRS, component of Lightspeed program 9-15
strspn() 13-158
struct
  array without size as last member 12-8
  defined with no members D-18
  error attempting to initialize D-3
  error initializing D-13
  illegal operation on D-9
  illegal to cast to another type D-7
  initialization of 12-8
  name space in 12-9
  padded to even size 12-8
  passing as argument 10-3
  redefinition of existing D-20
  undefined D-23
  unknown member D-24
  use does not match declaration D-25
stscmp() 13-159
stspfp() 13-160
switch statement 12-9
  value must be integral D-21

symbol
already defined D-15
invalid redeclaration of D-14
not defined D-15
used more than once in parameter list D-6
syntax errors D-21
system errors 7-2

T

tan() 13-74
tanh() 13-76
TE, 11-2
tell() 13-161
terminal id number 13-167
test
whether at end of stream 13-37
whether character has characteristic 13-32
whether I/O error flag is set for stream 13-38
whether keyboard was hit 13-66
time() 13-162
time
display 13-31
return in seconds 13-162
return pointer to time record 13-69
toascii() 13-163
toint() 13-164
tokens, illegal D-11
_tolower() 13-165
tolower() 13-165
Toolbox
argument wrong size D-18
illegal use of inline function D-11
routines 3-11, 14-1 - 14-51
Total Display (TD) Macsbug command A-3
_toupper() 13-166
toupper() 13-166
Trace (T) Macsbug command 7-7, A-2
Tracing, with Macsbug 7-7, 7-10
Transfer... command 5-5, 6-2, 6-5, 6-16, 8-1
ttyn() 13-167
turnaround time 1-1, 11-1, F-1
type
additional specifiers 12-7
considerations in Pascal to C conversion 9-5
invalid D-15
typedef, illegal use of symbol D-12

U

unary operator 12-5
Undo command 4-5, 6-6
ungetc() 13-168
ungetch() 13-169
union
defined with no members D-18
error attempting to initialize D-3
illegal operation on D-9
illegal to cast to another type D-7
initialization of 12-8
may not have bitfields D-23
name space in 12-9
padded to even size 12-8
redefinition of existing D-20
undefined D-23
unknown member D-24
use does not match declaration D-25
UNIX, 10-1, 10-5 - 10-7, E-2
unix library 13-1
unix_strings library 13-1
unlink() 13-170
user id number
return 13-63
set 13-119

V

VAR parameters, C equivalent to 9-5
variable format and print 13-171
vcprintf() 13-171
vfprintf() 13-171
void
function must not return value D-25
functions returning 12-9
generic pointer type 12-9
illegal operation on D-26
illegal to declare as variable D-22
illegal use of D-12
pointers to 12-3
vprintf() 13-171
vsprintf() 13-171

W

Where (WH) Macsbug command 7-12, A-2
WIND editor, in ResEdit C-6
WIND resource B-7
Wrap Around option 6-15
write() 13-172
write
block of characters to a file 13-54, 13-172
string at cursor position 13-67
word as 2 byte number to stream 13-93

X

XLISP 1.4 1-1, F-1

THINK
TECHNOLOGIES, INC.
420 BEDFORD STREET
LEXINGTON, MA 02173

THINK's Plain Language License Statement

This manual and the software described in it were developed and are copyrighted by THINK Technologies, Inc. and are licensed to you on a non-exclusive basis. This manual, or the software, may not be copied in whole or in part except as follows:

(1) You may make backup copies of the software for your use providing that they bear THINK's copyright notice.

(2) You have the right to include object code derived from the libraries in programs that you develop using the software and you also have the right to use, distribute and license such programs to third parties without payment of any further license fees providing that THINK's copyright notice is affixed to any such products in the manner specified in the license agreement.

You may not in any event distribute any of the source files provided or licensed as part of the software.

You may use the software at any number of locations so long as there is no possibility of it being used at more than one location at a time. Multi-user licensing arrangements may be made by contacting THINK Technologies, Inc.

Warranty

Limited Warranty on Media and Manuals

If you discover physical defects in the media on which this software is distributed or in the manuals distributed with the software, THINK will replace the media or manuals at no cost to you provided that you return the defective materials along with a copy of your receipt to THINK or to an authorized THINK dealer during the 60 day period following your receipt of the software.

Limited Warranty on the Product

THINK warrants that the software will perform substantially as described in the User's Manual. If within 60 days of receiving the software you give written notification to THINK of a significant, reproducible error in the software which prevents its operation, and provide a written description of the possible problem along with a machine readable example, if appropriate, THINK will either provide you with corrective or workaround instructions, a corrected copy of the software, a correction to the User's Manual, or THINK will refund your purchase price upon return of all copies of the software and documentation together with a copy of your receipt.

EXCEPT FOR THE LIMITED WARRANTY DESCRIBED ABOVE, THERE ARE NO WARRANTIES TO YOU OR ANY OTHER PERSON OR ENTITY FOR THE PRODUCT EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE, ALL SUCH WARRANTIES ARE EXPRESSLY AND SPECIFICALLY DISCLAIMED. Some states do not allow the exclusion of implied warranties or limitation on how long they last, so the above exclusion and limitation may not apply to you. This limited warranty gives you specific legal rights and you also may have other rights that vary from state to state. IN NO EVENT SHALL THINK BE RESPONSIBLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR SIMILAR DAMAGES OR LOST DATA OR PROFITS TO YOU OR ANY OTHER PERSON OR ENTITY REGARDLESS OF THE LEGAL THEORY, EVEN IF WE HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Some states do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you. The warranty and remedies set forth are exclusive and in lieu of all others, oral or written, express or implied.

LIGHTSPEEDC™ USER'S MANUAL

THINK
TECHNOLOGIES, INC.

THINK's Plain Language License Statement

This manual and the software described in it were developed and are copyrighted by THINK Technologies, Inc. and are licensed to you on a non-exclusive basis. This manual, or the software, may not be copied in whole or in part except as follows:

(1) You may make backup copies of the software for your use providing that they bear THINK's copyright notice.

(2) You have the right to include object code derived from the libraries in programs that you develop using the software and you also have the right to use, distribute and license such programs to third parties without payment of any further license fees providing that THINK's copyright notice is affixed to any such products in the manner specified in the license agreement.

You may not in any event distribute any of the source files provided or licensed as part of the software.

You may use the software at any number of locations so long as there is no possibility of it being used at more than one location at a time. Multi-user licensing arrangements may be made by contacting THINK Technologies, Inc.

Warranty

Limited Warranty on Media and Manuals

If you discover physical defects in the media on which this software is distributed or in the manuals distributed with the software, THINK will replace the media or manuals at no cost to you provided that you return the defective materials along with a copy of your receipt to THINK or to an authorized THINK dealer during the 60 day period following your receipt of the software.

Limited Warranty on the Product

THINK warrants that the software will perform substantially as described in the User's Manual. If within 60 days of receiving the software you give written notification to THINK of a significant, reproducible error in the software which prevents its operation, and provide a written description of the possible problem along with a machine readable example, if appropriate, THINK will either provide you with corrective or workaround instructions, a corrected copy of the software, a correction to the User's Manual, or THINK will refund your purchase price upon return of all copies of the software and documentation together with a copy of your receipt.

EXCEPT FOR THE LIMITED WARRANTY DESCRIBED ABOVE, THERE ARE NO WARRANTIES TO YOU OR ANY OTHER PERSON OR ENTITY FOR THE PRODUCT EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTIBILITY AND FITNESS FOR A PARTICULAR PURPOSE, ALL SUCH WARRANTIES ARE EXPRESSLY AND SPECIFICALLY DISCLAIMED. Some states do not allow the exclusion of implied warranties or limitation on how long they last, so the above exclusion and limitation may not apply to you. This limited warranty gives you specific legal rights and you also may have other rights that vary from state to state. IN NO EVENT SHALL THINK BE RESPONSIBLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR SIMILAR DAMAGES OR LOST DATA OR PROFITS TO YOU OR ANY OTHER PERSON OR ENTITY REGARDLESS OF THE LEGAL THEORY, EVEN IF WE HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE. Some states do not allow the exclusion or limitation of incidental or consequential damages, so the above limitation or exclusion may not apply to you. The warranty and remedies set forth are exclusive and in lieu of all others, oral or written, express or implied.

THINK
TECHNOLOGIES, INC.
420 BEDFORD STREET
LEXINGTON, MA 02173