Mac C™ and Mac C Toolkit

A Programmer's Guide

Consulair Corp

CHAPTER 1:
HOW TO USE THE MAC C COMPILER

INTRODUCTION

The Mac C Compiler translates programs written in the C programming language into 68000 assembly language for Apple's Macintosh 68000 Development System (MDS). The Compiler is fully integrated with this development system and runs on either a standard Macintosh (128K or 512K bytes) with an external drive or hard disk, a Macintosh XL, or a Lisa running MacWorks.

The C language is defined in **The C Programming Language** by Brian W. Kernighan and Dennis M. Ritchie. Major differences between Mac C and standard Kernighan and Ritchie C are defined in Appendix A. The Macintosh 68000 Development System consists of an Editor, Assembler, Linker, Executive, and Debugger. It is available from Apple and all parts except the Executive and the Debugger are required by Mac C. For information on the MDS see the **Macintosh 68000 Development System Manual** or contact your local Apple dealer. For assembly language information, refer to Motorola's **M68000 16/32-Bit Microprocessor Programmer's Reference Manual, fourth edition,** published by Prentice-Hall, Inc. Information on the Macintosh operating system and run-time library may be obtained from Apple's **Inside Macintosh** manual.

Some references for the C programming language are: **The C Programming Language** by Brian W. Kernighan and Dennis M. Ritchie, published by Prentice-Hall, Inc., and **C: A Reference Manual** by Samuel P. Harbison and Guy L. Steele, Jr., published by Prentice-Hall, Inc.

The remaining sections of this chapter provide a brief overview of the C language, and the sequence of steps the user must follow to compile, link, and execute Mac C programs.

OVERVIEW OF THE C LANGUAGE

The C programming language, originally developed at Bell Telephone Laboratories, was designed as a systems implementation language for the UNIX operating system. It is rapidly becoming the dominant systems implementation language over a wide variety of machines and systems.

C is a low-level programming language in the sense that important aspects of the hardware can be directly manipulated from within the language. It also includes features of higher-level languages. C supports a number of data types and offers structured control flow and a wide variety of operators.

Data Types

C recognizes several elementary data types: signed and unsigned characters (8 bits), signed and unsigned short integers (16 bits), signed and unsigned long integers (32 bits), pointers (32 bits), single-precision floating point numbers (32 bits), double-precision floating point numbers (64 bits), and extended-precision floating point numbers* (80 bits). From these elementary data types, more complex collections can be created: arrays of objects, each having the same data type, and structures (called records in some languages) of objects with arbitrary types. Floating point types are only implemented in Mac C 2.0.

This set of data types covers all the hardware-supported data types of the 68000 and offers mechanisms for extension to more complex cases. C provides a facility called **typedef** for creating new data types.

Operators

C provides many operators to manipulate the elementary data types. These operators are listed in Table 1-1. C permits extensive manipulation of pointers, i.e., variables that contain the addresses of operands. In C, the unary operators "*" and "&" can be combined with arithmetic operators to reference memory in a direct and efficient manner.

Table 1-1. C Operators

Arithmetic Operators	
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulus
Relational Operators	
>	Greater Than
>=	Greater Than or Equal To
<	Less Than
<=	Less Than or Equal To
==	Equal To
!=	Not Equal To
&&	Logical AND
||	Logical OR
Bitwise Binary Operators	
&	Bitwise AND
|	Bitwise Inclusive O
^	Bitwise Exclusive OR
<<	Left Shift
>>	Right Shift
Unary Operators	
*	Contents of Address
&	Address of Operand
-	Arithmetic Negate
!	Logical Negate
~	One's Complement
++	Increment
--	Decrement
(type)	Cast (Type Conversion)
sizeof	Size of Object (bytes)
Assignment Operators	
=	Simple Assignment
+=	Add, then Assign
-=	Subtract, then Assign
*=	Multiply, then Assign
/=	Divide, then Assign
%=	Modulus, then Assign
<<=	Left Shift, then Assign
>>=	Right Shift, then Assign
&=	Bitwise AND, then Assign
^=	Bitwise Exclusive OR, then Assign
|=	Bitwise Inclusive OR, then Assign

Functions and Program Organization

The basic organizational unit of C programs is the function. Every C program begins at a function called **main**. It makes use of both predefined functions and user-defined functions which fill the same role as subroutines or procedures in other languages. Functions can be compiled independently and later linked togther for execution. C functions can easily be made both recursive and re-entrant.

All function parameters in C are passed by value, that is, the value of the parameter is passed to the function. In most languages, parameters are passed by reference and the address of the parameter is passed to the function or procedure. Since a pointer can be a parameter value, call by reference is also available in C.

Within functions, control flow statements specify the order in which computations are to be done. C has a number of structured control flow constructs, including **if-else**, **else-if**, **while**, **do-while**, **for**, and **switch-case**, as well as three varieties of jumps: **continue**, **break**, and **goto**.

Storage

The C language allows explicit control over how values are stored and where they may be used. Variables can be specified as local to a function, global to all functions in a source file, or global to all functions in a program. When C programs execute, local variables are stored on a stack. Global variables are stored in a common data area allocated by the run time library.

THE MAC C SOFTWARE PACKAGE

The Mac C Compiler transforms a C source program into a relocatable object module by invoking the C compiler when then invokes the MDS Assembler, if the compilation is completed without errors.

Writing and Developing Mac C Programs

To write a Mac C application on the Macintosh, Macintosh XL, or Lisa running MacWorks, you must know how to use:

- Macintosh Finder, a built-in application for organizing and managing documents (see the **Macintosh** manual).
- EDIT, the MDS program editor.

- EDIT, the MDS program editor.
- Mac C, the C compiler.
- LINK, the MDS linker.

An understanding of all the MDS applications, however, will aid in program development and support. Some of these applications and their functions are:

- EXEC, the MDS Executive, makes it easy to move between applications and to create a batch control file to perform multiple compilations.
- ASM, the Macintosh Assembler, is used to write an assembly language module to be linked with a Mac C program.
- MacsBug, and Termbug, two of several MDS stand-alone debuggers, simplify the debugging process.
- MacDB is a more powerful debugger which allows symbols from a Mac C program to be used in symbolic debugging (MacDB requires two Macintosh machines -- Macintosh, Macintosh XL, or Lisa running MacWorks).
- MacNub, a small debugging utility, must be run on the machine containing the program to be debugged by MacDB. When using MacDB it is useful to make MacNub the startup program.
- RMaker or ASM can be used to create various "resources". The ASM **RESOURCE** pseudo-op allows the inclusion of arbitrary code and data structures as resources. RMaker allows the creation of linker ".REL" files that can be included via LINK in a Mac C program. Alternatively, resources can be merged into an existing program with RMaker.

Installing Mac C

This section describes how to install Mac C on a 128K or 512K byte Macintosh, a Macintosh XL, or a Lisa runing MacWorks. The installation steps are the same for both systems. You need **Mac C** and the Apple Macintosh Development System (**MDS**) to complete this sequence.

Installation Sequence

1. Make a copy of the disk labeled Mac C. This will become your **Mac C working disk**. If you have Mac C 1.5, remove the files **Install C** and **Edit for Ramdisk.**

2. Make a copy of your MDS 1 Disk and move the file **Install C** onto it. If you have Mac C 1.5, **Install C** is on your Mac C disk. If you have Mac C 2.0, **Install C** is on the Mac C Auxilliary disk.

3. Run the application **Install C.** It will print the message "Installation Complete" when it has finished.

4. Now move the following files from your MDS 1 copy onto the **Mac C working disk**:

   ASM The MDS Assembler
   LINK The MDS Linker

5. You now have a **Mac C working disk** which is ready to run. Use this disk in the internal disk drive and put your source disk in the external disk drive.

We at Consulair have worked hard to provide you with the best and most complete C development system available on the Macintosh. We have decided to sell our product without copy protection in order to maximize its utility to you, our customer. Please, if your friends or co-workers wish to use the compiler, abide by the license agreement and ask them to purchase their own copy. We have a discount schedule for multiple copy purchases which we will gladly tell you about. We can only maintain our high level of quality and support if you, our customers, support us and our product.

Thanks for your cooperation!

Making Backup Copies of Mac C and Mac C Toolkit

Once the **Mac C working disk** disk is ready to go, you may want to make backup copies of it. Any disk copying method can be used to make a new disk with all the files from the Mac C disk or Mac C Toolkit disk. You can make backup copies exclusively for your own use, as stated in the license agreement.

Running Mac C

This section describes the commands that should be used to invoke the Mac C Compiler. It is assumed the internal drive contains a **Mac C working disk** with the following files:

C	The Mac C Compiler
ASM	The MDS Assembler
Edit	The MDS Editor
Exec	The MDS Executive
Link	The MDS Linker
System Folder	

These files practically fill a Macintosh disk, so all source files should be saved on the external disk drive. If you have a Macintosh XL or a Macintosh 512K, you may wish to run with a RamDisk. There is a description of how to set this up on page 1-9. There are three application programs from which Mac C can be run: Edit, Exec, and the Finder. How to run Mac C from each of these programs is explained below.

From EDIT

1. Select C from the Transfer menu.

   If the file in the frontmost window has the extension ".c ", the Transfer menu says "C FILENAME.C," and that file will be compiled.

   OR:

   If there is no file with a ".c" extension, or the FIND/SEARCH window is frontmost, the Transfer menu simply says "C." Selecting C from the Transfer menu starts up the Mac C Compiler and allows you to select the file you want. Either select the name of the file and the Compile menu item in the "standard file" dialog, or double click the name of the file you want and the compilation will begin.

2. If there are no errors in the compilation, the intermediate file is assembled automatically and you are left in EXEC. The intermediate file is named FILENAME.ASM If errors are found in the compilation, you will be placed in EDIT with the source file and error file opened. The error file is named FILENAME.CER. Figures 1-1 and 1-2 show various error messages on the Mac screen after a successful and then an unsuccessful compilation.

From EXEC

1. Select C from the Transfer menu. The Mac C compiler will start and allow you to select the desired file from the "standard file" dialog.

2. Either select the name of the file and the Compile menu item in the "standard file" dialog, or double click the name of the file you want and the compilation will begin. If there are no errors in the compilation, the intermediate file is assembled automatically and you are left in EXEC. The intermediate file is named FILENAME.ASM If errors are found in the compilation, you will be placed in EDIT with the source file and error file opened. The error file is named FILENAME.CER.

From the Finder

Double click the "C" icon, to start the Mac C Compiler. Then follow the instructions for running Mac C from EXEC.

How To Stop Mac C

To stop Mac C during a compilation, hold down the Command key while typing a period (.). This combination is detected by Mac C at the following points:

- At the end of the preprocessor pass.
- Whenever an include file is closed.
- Whenever an error is encountered.
- At the end of compilation.

The stop command only needs to be issued one time because it is saved until the next point at which Mac C looks for it. When Mac C recognizes the stop command, all files are closed and control is immediately passed to Exec. ASM is not called even if the compilation is successful, and EDIT is not called if there were syntax errors.

Figure 1-1. Mac C Screen After Successful Compilation

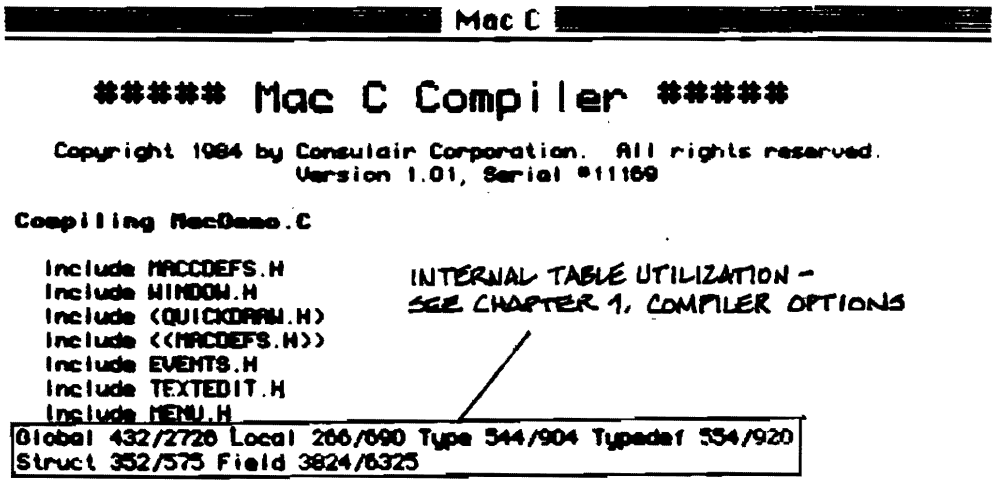
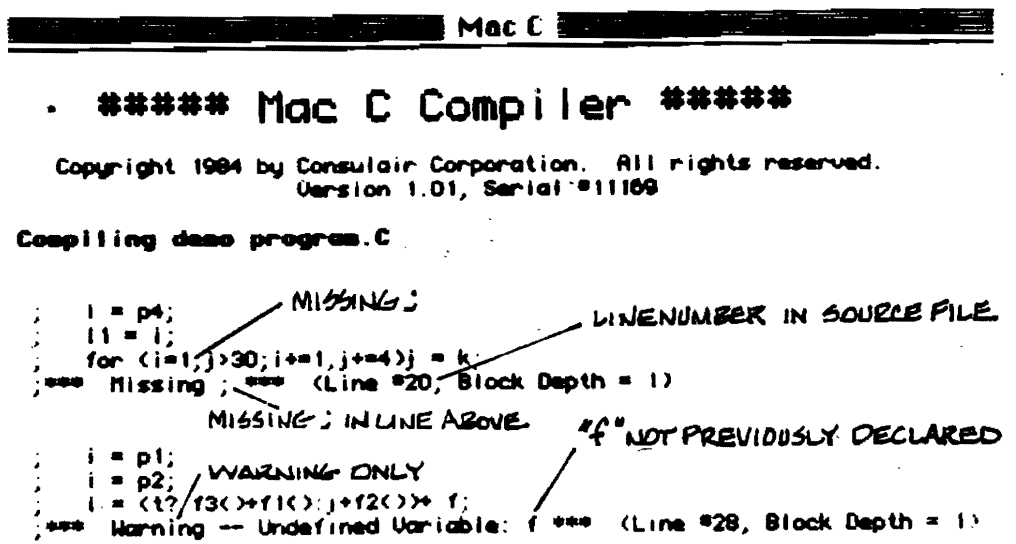


Figure 1-2. Mac C Screen After Compilation with Errors



Using a Ram Disk

If you have a Macintosh with at least 512 K bytes of memory, you may wish to run keeping Mac C on a Ram Disk. On a Macintosh without a hard disk drive, this will free up about 110K bytes on your internal drive. After you create a Ram Disk (using one of the commonly available Ram Disk programs) with at least 115K Bytes of available storage, Insert a ***copy*** of your **Mac C working disk** into the internal drive, and move the file "C" to the Ram Disk. You may now delete it from your working disk copy (be sure to save it elsewhere first!)

Use the Resource Editor (which is available from Apple) to modify the the EDIT transfer menu. Precede the "C" transfer menu item by the Ram disk volume name and a colon. Be sure to include any leading or trailing blanks in the volume name. You may now call the Ram Disk copy of Mac C from EDIT. To call Mac C on a Ram Disk from an EXEC file, you must precede the "C" in the Exec file with the volume name and a colon. You must use the EXEC shipped with the Mac C disk. It will not work with the normal MDS EXEC. For your convience, your Mac C Auxilliary disk (or Mac C disk for version 1.5) contains a copy of EDIT which has been modified to run Mac C from a Ram disk volume named "RamDisk " (note trailing blank).

Mac C Compiler Options

This section describes the Mac C Compiler options that can be changed using either menu items or a preprocessor directive in the source file. Some options pertain to a particular compilation and others are specific to a source file.

Compilation Specific Options

Options pertaining to a specific compilation of a file are set using the **Options** item in the menu bar. As with all standard Macintosh applications, you must select the **Cancel** item in the standard file window before the menu bar can be used. Then, to return to the standard file window, select **Select File** from the **File** menu heading. The options available under the **Options** menu are:

Warnings Are Errors [Default = on]
Treat any warning messges, such as incompatible pointer type, as errors.

Source In ASM .[Default = off]
Interlist the source file in the assembly language intermediate output file. The source statements are written as comments, followed by the generated assembly code. Error messages also appear in this file, so this option is sometimes useful for finding difficult syntax errors.

Verbose Errors [Default = off]
This option causes the Compiler to put more information in error messages, such as line numbers and comments for errors in included files.

List Token File [Default = off]
This option causes the compiler to generate a text file named "C.LIST" from its internal token file. This file is the C program with all comments removed and all defines expanded. This is useful for finding errors in complex macros, since it shows what the Compiler is actually compiling. A few lines inserted at the beginning of this file indicate the settings of certain options.

Program Specific Options

These options control the way a source file is interpreted by Mac C, or certain compilation time parameters which are specific to the source file. They are set by incorporating an **Options** command in the source file (options set in **Include** files are considered carried through to the source file). Any number of options may appear on the same line, separated by spaces. The format is as follows:

#Options <option> <option> . . .

where <option> ::= <Flag Option> | <Allocation Option>

Flag Options
Flag options set and reset Compiler flags and are always a single alphabetic character which may optionally be preceded by "+" (on) or "-" (off). If the sign character is omitted, the default is **on**. The flag options are:

A Convert function arguments to type **int** [Default = on]
This flag, which is normally set to on, controls whether or not the Compiler converts char and short arguments in a function call to type "int". Setting this option to off disables this conversion and results in a smaller program. Be careful, though, since this is dangerous.

G - create ASM file

B List Token File [Default = off]
This is the same flag that is controlled by the **List Token File** menu option.

E Error Flag [Default = off]
This is the same flag that is controlled by the **Verbose Errors** menu option.

H Source in ASM [Default = off]
This is the same flag that is controlled by the **Source in ASM** menu option.

I Integer size [Default = on]
The default is the standard Mac C integer size of 32 bits. Pascal, however, uses 16-bit integers and the architecture of the 68000 itself suggests a 16-bit integer size. Setting this flag to off(-I) causes **ints** to be compiled as 16-bit quantities, and **shorts** to be compiled as 8-bit values. The resulting code is faster and more compact, but it is less compatible with other 68000 compilers. The "standard" C Library assumes a 32-bit integer size for all routines including **scanf** and **printf**. Longs or explicit type casts must be used when calling these routines with 16 bit integers.

N No automatic trap recognition [Default = on]
Turning off this option (-N) suppresses the scanning of Macintosh trap names when compiling a program. If this option is off, trap names are no longer defined as normal functions, so they can be used in a program as function or variable names. Macintosh traps may still be called, but each name must be preceded by "#". The name is, in this context, case insensitive. Otherwise the call is the same. This speeds up compilation, especially if traps are not being used in a source file, and makes it easy to distinguish between normal function calls and Macintosh trap calls in a program.

P Padding Flag [Default = on]
This flag, which is normally set to on, controls whether or not Mac C inserts padding bytes into structures. These padding bytes force even byte alignment for shorts, longs, subordinate structures, and structure sizes. If it is turned off, structures will be compiled precisely as they are declared, so the structure declaration must be written with items properly lined up on word boundaries.

W Warnings are Errors [Default = off]
This is the same flag that is controlled by the **Warnings Are Errors** menu option, ***except that its sense is reversed,*** i.e. setting this flag (+W) makes warnings only warnings, and clearing it (-W) makes them errors.

Allocation options

Allocation options set the symbol table sizes used for compiling a source file, and the index register which is used for accessing the Global Data Segment. The general form of an allocation option is:

<letter> = <number>.

The Global Data Segment index register is set by:

R = <number>

<number> is 2, 3, 4, or 5 for registers A2, A3, A4, or A5, respectively. The Global Data Register is A5 by default, and should only be changed for special applications such as desk accessories. (This requires the Consulair Desk Accessory Maker.)

Symbol table sizes are printed on the screen and written to compiler error files at the end of a compilation. If a compilation causes a symbol table to overflow (this will only occur on a 128K byte Macintosh), examining the table size message will reveal which table has overflowed. The table size message has the format "Table Used/Allocated", where "Table" is **Global**, **Local**, **Type**, **Typedef**, **Struct**, **Field**, **or Float** and "Used" and "Allocated" indicate how many bytes were allocated for the table and how much of that space was actually used. The table which overflowed is usually that for which "Used" and "Allocated" are nearly the same size.

Note that the size of **Global** (the symbol table used for global and static names) may not be set. After all other tables are allocated, **Global** is automatically assigned any remaining storage. This means the **Global** allocation can be increased or decreased by altering the allocations for other tables.

D = <number>
Set Typedef Symbol allocation (default = 920, size factor = 10).

F = <number>
Set Field Symbol allocation (default = 5026, size factor = 5).

L = <number>
Set Local Symbol allocation (default = 920), size factor = 4).

Q = <number>
Set Float Literal allocation (default = 300), size factor = 10).

S = <number>
Set Struct Symbol allocation (default = 575, size factor = 4).

T = <number>
Set Type Storage allocation (default = 904, size factor = 10).

Mac C automatically sizes symbol tables for use with RAM disks. The algorithm for allocating symbol table space for machines with more than 128K bytes is:

The total free memory beyond that allocated to the symbol tables by the deafault settings is calculated, and the size of the file buffers is subtracted (about 16K bytes on the 512K Mac). This extra memory is allocated to the symbol tables in roughly the following proportions:

Local Storage:	4/60
Type Storage	2/60
Typedef Storage:	6/60
Struct Storage:	4/60
Field Storage:	10/60
Float Literal Storage:	10/60
Global Storage:	24/60

The maximum size of any symbol table is limited to 32000 bytes. The **#Options** allocation sets the minimum size in bytes for the indicated symbol table.

RUN-TIME LIBRARY FILES

The minimum run-time library required by a Mac C application is contained in the file named **MacCLib**. It is about 700 bytes in length, and contains the code to do system and data initialization, bit field operations, and various arithmetic functions. Mac C 2.0 also requires **Floatlib.** This minimum library is sufficient to allow a Mac C application access to all Macintosh Toolbox, Quickdraw, and Operating System ROM routines. To gain greater functionality, Mac C programs may be linked with other librarys (the "standard" C library and the "Mac C Toolkit"), which are described in Chapters 3 and 4.

CHAPTER 2:
COMPILER CODE GENERATION

INTRODUCTION

The Mac C Compiler translates programs written in C into 68000 assembly language. Consequently, the characteristic features of C programs--data types, operators, control flow statements, functions, and storage classes--are mapped into the set of 68000 operators and operands. When debugging a C program, it is usually necessary to look at selected sections of the assembly code that were produced, since most debugging facilities operate at the assembly language level. This chapter therefore explains various code strategies of the Mac C Compiler.

REPRESENTATION OF DATA TYPES

The fundamental data types in C are characters, signed and unsigned integers of several lengths, and floating-point numbers (available in Mac C 2.0). From these basic data types more complex collections can be created: arrays of objects (each having the same data type), and structures (called records in some languages) of objects with arbitrary types.

The mapping provided by Mac C between C data types and 68000 data types is as shown in Table 2-1.

Table 2-1. Data Types

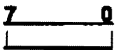
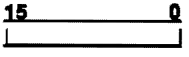
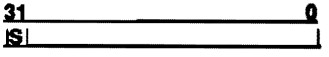
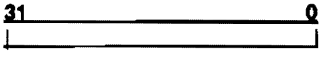
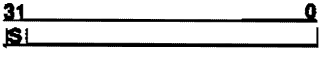
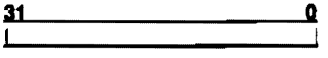
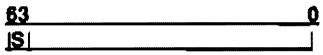
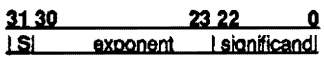
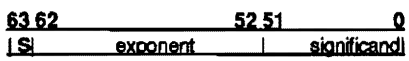
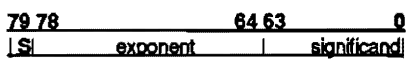
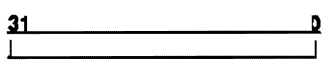
char	An 8-bit value defined by the ASCII character set, or an 8-bit signed integer in the range -128 to 127.  (NOTE: S = Sign Bit)
unsigned char	An 8-bit unsigned integer in the range 0 to 255. 
short int	A 16-bit signed integer in the range -32768 to +32767. 
unsigned short int	A 16-bit unsigned integer in the range 0 to +65536. 
int	A 32-bit signed integer in the range -2,147,483,648 to +2,147,483,647. 
unsigned int	A 32-bit unsigned integer in the range 0 to +4,294,967,295. 
long int	A 32-bit signed integer in the range -2,147,483,648 to +2,147,483,647. 
unsigned long int	A 32-bit unsigned integer in the range 0 to +4,294,967,295. 

Table 2-1. Data Types, continued

comp	A 64-bit signed integer (available with floating point only). 
float	A 32-bit real number in the IEEE Floating Point format. 
double float	A 64-bit real number in the IEEE Floating Point format. 
extended float	An 80-bit real number in the IEEE Floating Point format. 
pointer	An unsigned 32-bit integer in the range 0 to 4,294,967,295 

All variables except **char** will be word-aligned in memory.

REPRESENTATION OF OPERATORS

The C operator set was given in Table 1-1 in Chapter 1. Expressions involving these operators map efficiently onto the set of operators provided by the 68000.

Table 2-2 shows the same operators as Table 1-1, but shown along with each operator is the corresponding 68000 operator that is produced by the compiler.

Table 2-2. C Operators and 68000 Operators

Arithmetic Binary Operators		Unary Operators	
+	ADD*	*	(An)
-	SUB*	&	LEA, PEA
*	MULS*, MULU*, LSL	-	NEG
/	DIVS*, DIVU*, LSR, ASR	~	NOT
%	DIVS*, DIVU*, AND	++	ADDQ, ADD
		--	SUBQ, SUB
Relational Binary Operators		**Assignment Operators**	
>	These operators all have the	=	MOVE
>=	same general format:	+=	ADD
<	CMP expression	-=	SUB
<=	conditional branch	*=	MULS, MULU, LSL
==		/=	DIVS, DIVU, LSR
!=	where branch =	%=	DIVS, DIVU, AND
&&	BNE, BEQ, BLE, BGE,	<<=	LSL
||	BGT, BLT,	>>=	LSR
	BHI, BHS,	&=	AND
	BLO, BLS	^=	EOR
		|=	OR
Bitwise Binary Operators			
&	AND		
|	OR		
^	EOR		
<<	LSL		
>>	LSR		

* Also map into subroutine calls

REPRESENTATION OF CONTROL FLOW STATEMENTS

C includes three categories of instructions for altering the flow of control within the program: repetition statements, conditional branch statements, and unconditional branch statements. The three statements to control repetition are the **for** statement, the **while** statement, and the **do-while** statement. The conditional statements are **if-else** and **switch**. An unconditional branch is accomplished by using the **goto, break,** and **continue** statements.

The general code generation strategy for the repetition and conditional branch statements is a test followed by a conditional jump. The differences in the statements result in different positions for the test and different targets for the jump. Table 2-3 shows the skeleton code produced by the control flow statements.

Table 2-3. Control Flow Statements

Control Flow Statement		Skeleton Code

if (expression)
  statement1;
else
  statement2;
		CMP expression
		Branch if false to LABEL1
		statement1
		BRA LABEL2

	LABEL1:	statement2
	LABEL2:	(next statement)
while (expression) statement;	LABEL1:	CMP expression Branch if false to LABEL2 statement BRA LABEL1
	LABEL2:	(next statement)

do
  statement
while
  (expression);
	LABEL1:	statement
		CMP expression
		Branch if true to LABEL1
		(next statement)

for (expression1; expression2;
  expression3)
  statement;
	expression1
	LABEL1:
		CMP expression2
		Branch if false LABEL2
		statement
		expression3
		Branch to LABEL1

	LABEL2:	
goto (label);	BRA LABEL	
break;	BRA LABEL	
continue;	BRA LABEL	

Table 2-3. Control Flow Statements (cont'd)

Control Flow Statement	Skeleton Code
switch(value)	MOVE value,D0 BRA LABEL1
case v1: statement	LABEL2: statement
case v2: statement break; etc.	LABEL3: statement BRA EXITLABEL
default: statement	DEFAULTLABEL: statement BRA EXITLABEL

	LABEL1:
	CMP #v1,D0
	BEQ LABEL1
	CMP #v2,D0
	BEQ LABEL2
	etc.
	BRA DEFAULTLABEL

	EXITLABEL:
OR (If values are close together, the cases are sorted and the code is)	

	LABEL1:
	SUBQ #lowerbound,D0
	BLO DEFAULTLABEL
	BEQ LABELn1
	SUBQ #delta1,D0
	BEQ LABELn2
	SUBQ #delta2,D0
	BEQ LABELn3
	BRA DEFAULTLABEL

	EXITLABEL:
OR (If there are lots of values close together, the cases are sorted and the code is)	

	LABEL1:
	SUBQ #lowerbound,D0
	BLO DEFAULTLABEL
	CMP #upperbound,D0
	BHI DEFAULTLABEL
	LSL #1,D0
	MOVE LABELt(D0),D0
	BRA LABELt(D0)

	LABELt: DC.W LABELn1-LABELt DC.W LABELn2-LABELt DC.W LABELn3-LABELt etc.
	EXITLABEL:

REPRESENTATION OF FUNCTIONS

This section describes the Mac C Compiler's implementation of function calls. In the following discussion, the term parameter refers to the variable (enclosed in parentheses) named in a function definition; the term argument denotes an expression that is part of a function call.

Calling Conventions

Whenever a function is used in a C source program, as, for example,

```
func (a, b, c, . . . , h, i, j);
```

the Compiler produces a standard sequence of assembly language instructions, called the function calling sequence.

The Mac C function call mechanism has been designed to be fast and easily understood by assembly language routines, and to allow assembly language routines to be called without incurring overhead. Register arguments are saved on the stack by the prologue code in the function being called. Assembly language routines, of course, normally do not save the arguments, but simply leave them in registers.

An alternate parameter passing strategy is employed when a function parameter list contains one or more parameters which is either a floating point value, or a structure value whose size is greater than four bytes. In these cases (and when a function is specified as having a variable number of parameters), the parameters are pushed onto the stack before the function call in reverse order of their appearance in the argument list. Please see Appendix A for details on declaring functions with a variable number of parameters.

Figure 2-1 illustrates the normal C run-time stack structure during the calling sequence. Figure 2-1a shows the stack before a function call is made. Register A7, the stack pointer (SP), points to the topmost element on the stack. Register A6 is the local frame pointer. It always points to the reference point of the stack frame for the currently executing function. The stack frame is the area of the stack accessible during the called function's execution. Figure 2-2 shows the equivalent stack structure for functions using the alternate calling sequence.

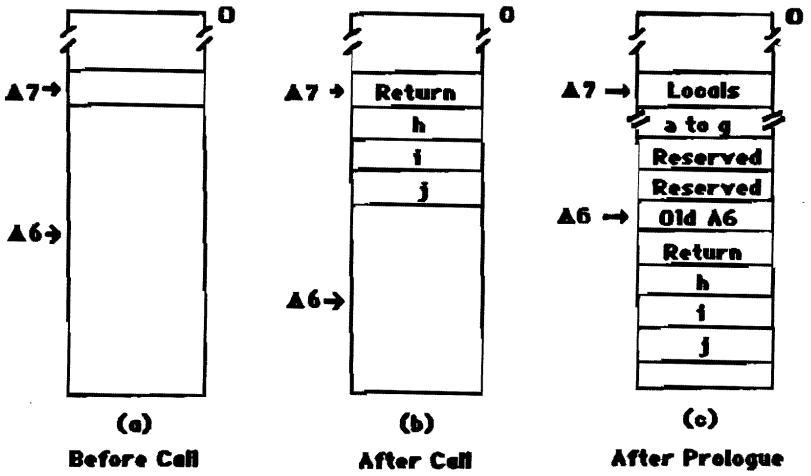


Figure 2-1. Run-Time Stack Format (Normal)

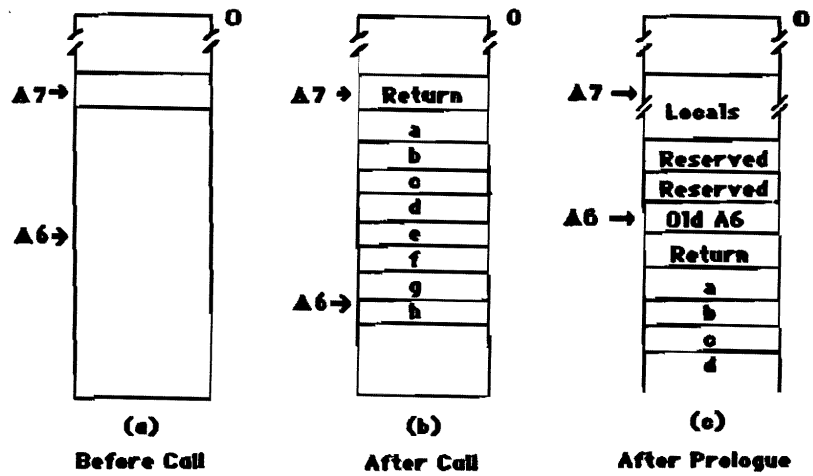


Figure 2-2. Run-Time Stack Format (Alternate)

The normal function calling sequence begins by pushing the surplus arguments (h, i, j) from right to left, onto the stack, and then loading the first seven arguments (a, b, c, . . . , g) in registers D0-D6. The alternate calling sequence pushes all arguments onto the stack at this point, and leaves the register contents undefined. Next, a JSR instruction pushes the return address onto the stack and puts the address of the called function in the Program Counter. Figure 2-1b illustrates the stack at this stage of the calling sequence. Note that A7 now points to the last item pushed onto the stack.

Because the 68000 stack grows toward lower memory addresses, the last argument to be pushed on the stack will have the lowest address.

The actual function being called is compiled into a standard form. At the beginning of the compiled code for each function is the prologue.

```
LINK A6,#-n                    ; Allocate space for parameters
                               ; and/or locals
MOVEM.L D0/D1/.../A6,-m(A6)    ; Move parameters onto stack
```

The 68000 LINK instruction pushes A6 onto the stack, loads the value of A7 into A6 and then decrements A7 by an amount specified in the instruction in order to create space for the called function's local variables (refer to Figure 2-1c).

When the function reaches a **return** statement or when control reaches the end of the function, the compiler produces a function epilogue. These instructions restore the stack pointer and frame pointer to the values they had before the LINK instruction (Figure 2-1b). and return control to the caller. The epilogue code is as follows:

```
UNLK A6
RTS
```

A function that either contains only inline assembly statements, or does not produce any executable code does not have a standard prologue or epilogue. It is simply ended by:

```
RTS
```

return statements within a function body produce the standard epilogue after the values to be returned are loaded into the return register. Only the first **return** statement will produce an epilogue. Other returns branch to the epilogue emitted by the first.

Functions returning floating point and structure values (where the size of the structure is larger than four bytes) are handled slightly differently . Before the arguments are pushed onto the stack for the function call, the address in which to store the result of the function is pushed onto the stack. The **return** statment in the function then uses this location to store the result of the function before returning with the normal epilogue.

Argument Passing

In C, all arguments are passed by value, i.e., the value of an argument, rather than its address, is passed to the function. When "call by reference" is required, the ampersand ("&") operator may be used with the variable name to pass its address.

Passed arguments of types **char** and **short** are expanded to type **int** and arguments of types **float**, **double**, and **comp** are expanded to type **extended**. Structures which are less than 5 bytes in length are passed as register values, and larger ones are passed on the stack. A function name appearing as an argument is converted to the address of the named function, and, an array name is converted to the address of the first element of the array.

Register Conventions

Called assembly language routines must observe the following conventions with regard to register usage:

1. The contents of registers A5, A6, and A7 must be preserved (saved and then restored). If a function is to be called from the Macintosh ROM (as in the case of a control definition or action procedure), registers D3-D7 and A2-A7 must be preserved.

2. The called routine may address only those portions of the stack that are at lower addreses than the current stack pointer value at the time the function is entered.

3. Register D0 contains the result of the function call if the result is a value; A0 contains the result if it is a pointer.

REPRESENTATION OF STORAGE CLASSES

The C concept of the storage class actually involves two different but related notions: the scope of a variable, and where that variable is stored.

Variables can have two possible scopes:

- Available to all functions in the program (global scope).

- Available only within the defining function (local scope).

There are three places where variables can be stored:

- In a register (up to 7 register variables are allowed in Mac C).
- On the stack, allocated dynamically (automatic storage).
- In the Global Data Segment produced by the compiler.

Combining scope and storage location yields four different storage classes for variables:

- Register variables may be stored in registers. Their scope is local.
- Automatic variables are stored on the stack frame. Their scope is local.
- External variables are stored in the Global Data Segment. Their scope is global.
- Static variables are stored in the Global Data Segment. Their scope is local to the source file (external statics) or procedure (internal statics) in which they are declared.

STORAGE OF VARIABLES

The Mac C strategy for allocating storage for variables is fairly simple:

- For externals, storage is allocated in the Global Data Segment.
- For static variables, storage is allocated exactly as with externals, but these variables are not declared external to the linker.
- For automatic variables, storage is allocated on the stack frame when the function is called. Storage is reclaimed at the return.

VARIABLE ADDRESSING

All local data is allocated on the stack, and is referenced by an address of the form:

-n(A6)

where "n" (a decimal number) represents the offset of the data from the stack base, A6.

Global data is allocated in the Global Data Segment which is normally pointed to by A5, and is referenced as:

```
name(A5)
```

or as:

```
name+n(A5)
```

where **name** is the C identifier for the data reference, and **+n** is the optional offset from the identifier (if any). For example, the lines:

```
int i, iarray[10];
.
i = iarray[5];
```

would produce:

```
MOVE.L iarray+20(A5),i(A5)
```

Mac C allows the specification of a register other than A5 as the global data base register to accommodate special applications such as desk accessories (when used with the Consulair Desk Accessory Maker) which may not use storage relative to A5. See Chapter 1: The Mac C Compiler Options.

STORAGE OF CONSTANTS

Constants, other than those used in initializing global variables, are normally stored in the code segment. They are stored either as explicit operands of 68000 instructions (e.g., MOVEQ #1,D0), or as data after the final executable instruction produced from a source file. String and floating point constants are of the latter form. This means that if the address of a string constant is saved and then the segment which contains the string is unloaded, the address will no longer be that of the desired string. Normally, this is not a problem because a segment which is in the calling chain to another segment may not be relocated.

A NOTE ON GLOBAL INITIALIZATION

String constants used to initialize global pointer variables are stored in segment 1. The warning about string constants in the section, STORAGE OF CONSTANTS, applies to these constants. If segment 1, which contains the string, is unloaded, the address stored in the global variable will no longer point to the string. There is a feature/bug in the storage of global string constants. If two global variables are initialized as pointers to the same literal string, the Assembler allocates a single copy of the string in code segment 1, and both global variables are set to point to that string. This is done as a space saving measure, but it has a potentially unpleasant side effect since changes to the string will be reflected through both pointers.

CHAPTER 3:
THE MAC C RUN-TIME ENVIRONMENT

INTRODUCTION

The term run-time environment refers to the hardware and software configuration of the target machine in which a program will actually execute. For the Mac C Compiler, the run-time environment is the Macintosh, Macintosh XL or Lisa with MacWorks, plus its ROM software, and one of the Mac C run-time libraries.

This chapter provides information about the major run-time environment features, including the Mac C-to-Macintosh run-time interface (Macintosh Toolbox Traps), the Mac C run-time libraries, and a way to handle run-time errors with signals.

THE MACINTOSH RUN-TIME INTERFACE

The Mac C-to- Macintosh run-time interface is defined by a group of ".h" (included header) files and the Mac C library functions and global variables. The ".h" files define Mac C and Macintosh system values and structures. These, along with library functions, are discussed in the next section (on the run-time library) and in the section on Mac C Traps.

THE MAC C RUN-TIME LIBRARIES

The Mac C run-time Libraries are a collection of functions that provide run-time support for input/output, initialization, and certain mathematical calculations not provided by the 68000 instruction set. All library functions assume that the integer size is 32 bits (the standard default in Mac C).

Mac C programs must be linked (at Link time) with a Mac C run-time library. Since the linker used on the Macintosh does not allow the selective linking of routines from a library, the programmer must select a library containing the functions his program requires from the available Mac C runtime libraries.

The libraries shipped with Mac C are:

The minimum Library
The "standard" C library
The Mac C Toolkit
The "standard" C library and the Mac C Toolkit

All of the Mac C libraries use the label **start** as the normal entry point. Link control files for applications using the normal libraries should specify **start** as the starting label for the application (just include the line "lstart" at the beginning of the link control file).

The Mac C Toolkit includes the sources to all libraries, which allows the programmer to selectively build his own libraries by removing or modifying routines from the standard ones. The only requirement is that the global initialization code be executed when the application is started, and the arithmetic library routines be provided for programs requiring them. These functions are normally done in the minimum library file, MacCLib (Source file MacCLib.ASM).

Use this as a guide for selecting the library you want to use:

- If you are writing a pure Macintosh application or a desk accessory (requires the Consulair Desk Accessory Maker) use the minimum library (files: MacCLib, Floatlib for Mac C 2.0). This allows full access to the Macintosh ROM routines.

- If you are using the functions provided in the "standard" C library, e.g. printf, scanf, fopen, fclose, strlen, etc., use the "standard" C library (files: StdLib, StdFileIO, StdIOPrim, Floatlib, Floatconv for Mac C 2.0).

- If you need the increased functionality of the Mac C Toolkit, e.g. asynchronous I/O, serial port I/O, string routines, use the Mac C Toolkit (see chapter 4).

- If you need the functionality of both the "standard" C library and the Mac C Toolkit, then use them both (use the file StdLib in place of MacCLib with the Mac C Toolkit. See chapter 4).

To use the routines in the "standard" C Library, the **stdio.h** header file must be included at the start of a source file. The header files required for Mac C Toolkit routines are described in Chapter 4.

All programs must contain a single function named **main**, which is the starting location for the program. Before calling **main**, Mac C initilization code initializes the Macintosh display by calling **_InitGraf**, **_InitFonts**, and **_InitWindows**. Mac C stores the address of the Quickdraw globals in an external global named **QD** which is defined in MacCLib or StdLib and may be used by your program.

Referencing Functions

The first use of a function in a source file determines its type. All functions are of type **Int** unless otherwise specified by a function definition or an external function declaration (for functions defined in other source files or a library). An external declaration is therefore required for any function which is referenced before its definition in a source file (or is defined in another source file or library) if its type is other than **Int**. ***This is critically important when using functions returning pointer values, since they return their result in register A0. If they are not declared as a function returning a pointer, the result will be presumed to be in register D0, even if the function call is cast to be of the proper type.***

The "standard" C Library

The "standard" C library contains a set of functions which are commonly implemented to support I/O, memory, string, and character operations in C. Since many of the functions use the display for teletype-like character input and output, Mac C implements a teletype simulation window for the standard library. This window is created by the initialization code before the function **main** is called. It is possible to suppress the creation of the teletype simulation window by specifiying **altstart** instead of **start** as the starting label in the link control file. Of course, you may not use any of the functions requiring the teletype simulation window if it does not exist, and unpredictable things will happen if you do. The **WindowPtr** for the teletype simulation window is kept in the global variable **console**. You may use this variable to change the size of it, show or hide it, destroy it, or move it. Call the global routine **SetTTY** whenever you have changed its size, or when you wish to activate it. Other windows may be made the current teletype simulation window by using **SetTTY**. You can have multiple teletype simulation windows, but only one may be currently active, and everything written to **stdout** will go there (unless stdout has been changed by your program).

When using the "standard" C library (including printf), you must include the header file **stdio.h** at the start of your source file, and you may not use the **-I** (16 bit integers) compiler option. **stdio.h** contains the external declarations required for functions in the "standard" C library, and definitions for the folowing values:

- ERROR
- EOF
- NULL
- MAXLINE

The type **FILE** is also defined in **stdio.h**. Files should be typed as **FILE ***. Internally, the Mac C library identifies them with short values. There is no difference at the library level between files opened by the **fopen** function and those opened with the **creat** or **open** functions. The **FILE *** definition is included as an aid to porting programs from other environments. Do not assume anything about the values of **FILE** reference numbers. For example, using 0 and 1 for console input and output (which works for many UNIX systems) will not work in Mac C: Both console input and output are 0. Using **stdin** and **stdout**, however, will work.

Testlib

The Mac C Auxilliary disk contains a program in the "Demo Folder" named "Testlib." This source code may serve as a useful reference since it contains examples of how the library routines are called. A listing of the program can be found in Appendix F.

"Testlib" calls and nominally tests the library functions in various ways. When it asks if you want to test a function press, "Return" for yes, "." to return to FINDER, and any other key for no. If you have trouble with one of the library calls, look at "Testlib.c" to see how it is used there.

Testlib.Job and **Testlib.Link** may be used as prototypical "job" and "link" files, respectively. The **Testlib.Job** file, when run from the Exec, will compile, link, and run **Testlib**.

Standard C Library Routines: A Summary

The following pages offer a summary of the "standard" C run-time library routines in which the name and a brief description of each is given. The routines have been categorized into those which operate on characters and strings, standard I/O, memory allocation, and a miscellaneous group. Consult **The C Programming Language** by Brian W. Kernighan and Dennis M. Ritchie or **C: A Reference Manual** by Samuel P Harbison and Guy L. Steele, Jr. for more information.

Character and String Manipulation Routines

For the routines listed below, the argument types are:

```
char c;
char *s, *s1, *s2;
int n;
typedef struct P_Str {char count; char contents[255];} P_Str;
P<-Str  *p;
```

Routine	Function
char isupper(c)	Return non-zero if "A" <= c <= "Z".
char islower(c)	Return non-zero if "a" <= c <= "z".
char isalpha(c)	Return non-zero if "a" <= c <= "z" or "A" <= c <= "Z".
char isdigit(c)	Return non-zero if "0" <= c <= "9".
char isspace(c)	Return non-zero if c = SPACE, TAB, NL (LF), CR, or FF.
char toupper(c)	Return c or upper case value of c if "a" <= c <= "z".
char tolower(c)	Return c or lower case value of c if "A" <= c <= "Z".
char *index(s, c) [or strchr]	Return 0 or pointer to first occurrence of c in s.
int Index(s, c) [or strpos]	Return -1 or index of first occurrence of c in s.
char *rindex(s, c) [or strrchr]	Return 0 or pointer to last occurrence of c in s.
int Rindex(s, c) [or strrpos]	Return -1 or index of last occurrence of c in s.
char *strsave(s)	Returns address of a copy of s (uses malloc()).
char *strcat(s1, s2)	Appends s2 onto s1.
char *CtoPstr(s)	Converts s to a Pascal string (P_Str). **This changes s, so that it is no longer a C string.** Returns s as its result.
P_Str *PtoCstr(p)	Converts p to a C string (char *). **This changes p, so that it is no longer a Pascal string.** Returns p as its result.
char *strncat(s1, s2, n)	Appends up to n bytes of s2 onto s1.
int strcmp(s1, s2)	Compares s1 to s2. Returns 0 if equal, -1 if s1 < s2, 1 if s1 > s2.
int strncmp(s1, s2, n)	Like strcmp, but compares up to n characters.
char *strcpy(s1, s2)	Copy str2 to str1. (s1 must be large enough to hold s2.)
char *strncpy(s1, s2, n)	Copies n characters of s2 to s1. If the length of s2 >= n, then s1 will not be null terminated. If the length of s2< n, then s1 will be null padded.
int strlen(s)	Returns length of s.
int atoi(s)	Returns numeric conversion of number in s, radix = 10.
long atol(s)	Similar to atoi but returns a long value.
extended atof(s)	Similar to atoi but returns an extended value.

Standard I/O Routines

Three variables define the standard input and output and error files (normally keyboard and tty):

```
FILE *stdin;
FILE *stdout;
FILE *stderr;
```

These can be set to any properly obtained FILE *variable or the result from **open** or **creat**.

The following routines use the argument types:

```
char c, *s;
char *buffer *name, *format, *dir;
int n;
short mode, size, w;
long offset;
FILE *file;
```

Routine	**Function**
int printf(format [, arg]...)	Formatted output. Formats and prints stdout. The first argument is a format specifier. Returns EOF on error. (See Kernighan & Ritchie or Harbison & Steele for format details.)
int sprintf(s, format [, arg]...)	Formatted output. Corresponds to printf, but formats to string "s" rather than standard output. Format string is the second argument. Returns EOF on error. (See See Kernighan & Ritchie or Harbison & Steele for format details.)
int fprintf(file, format [, arg]...)	For formatted file I/O, otherwise identical to printf except that output is written to "file". Format string is the second argument. (See Kernighan & Ritchie or Harbison & Steele for format details.)
int scanf(format [, pointer]...)	Formatted console input. Reads characters from stdio, interprets them according to a format, stores results in its arguments. Takes a string as a format specifier, and set of pointer arguments indicating where formatted input should be stored. (See Kernighan & Ritchie or Harbison & Steele for format details.)
int sscanf(s, format [, pointer]...)	Formatted input. Corresponds to scanf except it reads from a string rather than stdio.
int fscanf(file, format [, pointer]...)	For formatted file I/O, otherwise identical to scanf, except that it reads from file tather than stdio.
int getc(file)	Returns next char from file, or EOF if end of file. Does **NOT** echo characters to the TTY window when input is from the keyboard.
int fgetc(file)	Returns next char from file, or EOF if end of file.
short getw(file)	Returns next word (16 bits) from file, ignores end of file.
char *gets (s)	Reads a string terminated by RETURN from stdin.

Routine	Function
char *fgets (s, count, file)	Read up to count bytes from file, terminated by "\n", count, or EOF. Returns 0 if EOF at start of read.
long getl(file)	Returns next long (32 bits) from file, ignores end of file.
int getchar()	Returns next char from stdin. Does **NOT** echo characters to the TTY window when input is from the keyboard.
char putchar(c)	Writes c to stdout, and returns c.
char putc(c, file)	Writes c to file, and returns c.
int ungetc(c, file)	Puts c back onto file (one character maximum).
char fputc(c, file)	Writes str to stdio, and returns c.
int puts(s)	Writes str to file, and returns s.
int fputs(s, file)	Writes c to file, and returns s.
short putw(w, file)	Writes w to file (high byte, low byte) and returns w.
long putl(l, file)	Writes l to file (4 bytes, high order to low order), and returns l.
int creat(name, mode)	Creates a disk file identified by "name". Sets the size to 0, and opens it for writing. Mode is 0x400 = read, 0x2000 = write, 0x7, 0x70 = read/write.
int open(name, mode)	Opens the file identified by "name" according to mode: 0: read 1: write 2: read/write Returns -1 if file cannot be opened.
FILE *fopen(name, dir)	Opens the file identified by "name" according to first character of string "dir": r: read w: write (sets end of file to position 0) a: append (sets end of file to current EOF)
int close(file)	Flushes buffers and closes indicated file. Returns EOF on error.
int fclose(file)	Same as close.
int fflush(file)	Flushes buffers to disk. Returns EOF on error.
int unlink(name)	Deletes file identified by "name" from disk. returns EOF on error.
int read(file, buffer, n)	Reads up to n bytes from file into buffer. Returns actual number of bytes read (0 means EOF).

Routine	Fuction
int fread(buffer, size, n, file)	Reads up to "n" items, each "size" bytes long, into buffer. Returns number of items read.
int write(file, buffer, n)	Like read, but writes instead.
int fwrite (buffer, size, n, file)	Like fread, but writes instead.
long lseek(file, offset, mode)	Positions file according to mode: 0: "Offset" bytes from start of file. 1: "Offset" bytes from currentposition. 2: "Offset" bytes from end of file. Returns resulting position as byte offset from file start.
int fseek(file, offset, mode)	Same as lseek.
long tell(file)	Returns current byte position of file as offset from file start.
int ftell(file)	Same as tell.
int feof(file)	Returns non-zero if file is at EOF. For keyboard or serial ports, returns non-zero unless there is a character in the input buffer.

Memory Allocation Routines

The memory allocation routines use the Macintosh memory manager. All allocated areas are non-relocatable, and begin on even addresses. Areas are forced to be an even number of bytes long. (See Inside Macintosh for details on how the memory manager works.)

The following routines use these declarations:

```
int size, n;
char *ptr;
long *a, *b, value;
```

Routine	Function
char *malloc(size)	Allocates size bytes of memory, and returns pointer to first byte. Returns 0 on failure.
char *calloc(n, size)	Allocates n* size bytes of memory, sets it to 0, and returns pointer to first byte. Returns 0 on failure.
void free(ptr)	Releases space allocated by malloc or calloc.

Miscellaneous Routines

Routine	Function
void swap(a, b)	Exchanges the contents of the long locations addressed by a and b.
void exit(value)	Closes all files and returns to finder (value is ignored).
void _exit(value)	Returns to finder (value is ignored).

MAC C TOOLBOX TRAPS

Traps are used by applications on the Macintosh to make calls on the operating system, Quickdraw, and Toolbox and various manager routines. The code for these routines is normally stored in the ROM, and all ROM routines are accessed through traps. The Macintosh traps are defined in the **Inside Macintosh** manual, which should be consulted for a full description of trap functions.

Mac C allows Macintosh traps to be called just like normal C functions. Arguments are passed in the order indicated by the Pascal procedure definitions in **Inside Macintosh**. Mac C automatically emits the proper parameter loading and trap instruction (no "glue" routines" are used), and subsequently returns the result. In order to determine the proper argument types for any given trap, you should refer to the Pascal definition of the function in **Inside Macintosh**, and the Pascal/C argument correspondence defined in Appendix C of this manual. While Mac C cannot check the actual type of all arguments passed to Macintosh traps, it does check the number of arguments, and type converts arithmetic values to the precision required by the trap.

Macintosh trap routines require strings to be in Pascal format (which is a count followed by text) rather than C format (text terminated by 0). C strings must be changed to Pascal format before being passed as arguments to Macintosh traps. Two Mac C library functions facilitate this conversion: **CtoPstr** and **PtoCstr**. These perform an in-place conversion of C-to-Pascal and vice versa. They are declared in **stdio.h**, and defined in **MacCLib** and **StdLib** (the source files containing the actual code are MacCLib.ASM and StdCLib.ASM).

If **stdio.h** is not being included in the source file, **PtoCStr** and **CtoPStr** must be declared in your source file before use:

```
extern char *PtoCStr();
extern P_Str *CtoPstr();

where P_Str is defined as:
    typedef struct {char count; char contents[255];} P_Str;
```

Care should be taken when using the **PtoCstr** and **CtoPStr** functions. They perform an in-place conversion of the string, and ***if they are called with the address of a constant string, that constant is permamently converted.*** It is a good idea to follow any call to **CtoPstr** with a call to **PtoCStr** unless permanent conversion is desired. Mac C Toolkit users may want to use the routine **tempMacStr** instead of **CtoPstr**, since it does not change the original string.

Mac C does not use "glue" routines to convert strings for those traps which use them (there are about 20) because it would be inconsistent with Mac C philosophy to have such

hidden routines, and because this kind of automatic conversion cannot be perpetuated uniformly through all calls, results, packages, and future routines. A "glue" routine that performs string conversion can always be written by the programmer.

The low level I/O routines described in the operating system section of **Inside Macintosh** are generally not implemented in favor of their "Parameter Block" equivalents, which are identical except that they accept a second argument which is a boolean governing whether the call is to be executed synchronously (zero) or asynchronously (non-zero). The name of a "Parameter Block" routine is the same as the corresponding trap name preceded by "PB", e.g. **Open** becomes **PBOpen**, **close** becomes **PBClose**, and so on. To use the Parameter Block routines, the file **pbDefs.h** must be included at the start of your source file.

Inside Macintosh describes some routines which are not traps, but are instead "Pascal Only" procedures or "packages." These routines are not supported directly by Mac C. In many cases, there are direct trap calls which provide the same function. In some cases, they require "glue" routines, which are easy to write in Mac C (see Appendix C). Many such "glue" routines are included in a collection of Mac C Examples available from Consulair Corporation.

MACINTOSH HEADER FILES

Most system values and structures are defined in the collection of ".h" files. These files are grouped according to the **Inside Macintosh** divisions for traps (e.g., **quickdraw.h, font.h**). Include the header file for the functions you want in your source file and it will in turn include any header files which it depends on.

Unless the system in use is a Lisa with MacWorks, a Macintosh XL, or a 512 K Macintosh, not all of the ".h" files will fit in the symbol table space available for a compilation. Normally, this is not a problem because you need to include only those files actually required by the source file. If space is a limitation, it is safe to create ".h" files containing only those definitions required by the source program, or you can change the symbol table allocation (See The Mac C Compiler Options, Chapter 1).

Appendix E provides, in alphabetical order by trap name, the correct name and spelling, argument type, function result, and trap number for each Macintosh trap used by Mac C. A trap that returns a result is indicated by preceding the trap name with the result's type.

All traps that return pointers are typed as "char *". The appropriate ".h" file will cast this function to its proper Macintosh type.

Argument types are designated as **char** (8 bits), **short** (16 bits), **Point** (the address of a Macintosh **Point**), or **Long**. **Long** arguments are either 32-bit integers or pointers, as

required by the particular trap.
Some Macintosh traps accept their arguments in registers instead of on the stack. The register values D0, A0, and A1 designate the formal argument order for those traps (e.g., BlockMove). These arguments are untyped, and the program must determine whether they are appropriate values. With BlockMove (**Inside Macintosh**), for example, the first argument is (A0 = sourcePtr), the next is (A1 = destPtr), and the last is (D0 = nbytes). A call would look like this:

```
short result;
char *source, *dest;
int bytes;
result = BlockMove(source, dest, nbytes);
```

SIGNALS

Mac C implements three routines as an aid to handling run-time errors in an application or program: **CatchSignal**, **Signal**, and **LocalSignal**.

In a recursive language, it is normal to nest procedure calls many levels deep. Since functions often build contexts which must be destroyed before exiting, global processing of error conditions is difficult, if not impossible. Programs are frequently constructed so that each function returns an error status in order to solve this problem. This may interfere with the logic of the program, making it cumbersome and more difficult to understand, especially where multiple error conditions may exist.

Mac C uses the concept of a **signal** to handle errors and increase a program's simplicity and robustness. This concept works as follows: any procedure can call the function **CatchSignal**. This function, which takes no arguments, saves the program context, and returns a value of 0. The calling function may then call another function, which may call another function, which calls another, and so on. If any function calls the **Signal** function (which takes a single non-zero argument), control is immediately transferred to the location and context of the last **CatchSignal** call executed in a function below the current one in the calling sequence. At this point, the program behaves as though the original call on **CatchSignal** returned the argument passed to **Signal** as its result.

Here is an example:

```
p()
{
 char *ptr;
 char *error;
 ptr = malloc(100);
 if (error = CatchSignal())
          {
          printf("\rError Reported: %s", error);
          free(ptr);
          return;
          }

 p1(ptr);
}
p1(ptr)
 char *ptr;
 {
 p2();
 }

p2()
 {
  if (errorcondition) Signal("Some Error");
}
```

When function **p** first calls **CatchSignal**, a value of 0 is returned, and the "true" block of the conditional is not executed. When, and if, **p2** calls **Signal**, the stack is cut back and control is transferred so that it is just as if the original call to **CatchSignal** in function **p** returned a pointer to the string "Some Error". In this case, **p** prints the error and returns. A more general form of **p** would not return (which destroys the signal), but would pass the signal on down the stack:

```
p()
  {
  char *ptr;
  char *error;
  ptr = malloc(100);
  if (error = CatchSignal())
          {
          free(ptr);
          Signal(error); /* Let the procedure(s) which called me know */
          }
  p1(ptr);
  }
```

The function **LocalSignal** works just like **Signal**, except that it is "Caught" by an active **CatchSignal** in the same function. Suppose, in the earlier example, that **p2** opened a file which needed closing after an error. **LocalSignal** could be used to address that problem:

```
p2()
{
 FILE *file;
 file = NULL;
 if (error = CatchSignal())
          {
          if (file != NULL) fclose(file);
          Signal(error); /* Let the procedure(s) which called me know */
          }
 .
 file = fopen("SomeFile", "r");
 .
 .
 if (errorcondition) LocalSignal("Some Error");
 .
 .
 .
}
```

There is one restriction in the use of these functions: only one **CatchSignal** may be active in each instantiation of a function. There may, however, be multiple calls to **CatchSignal**. The last **CatchSignal** call actually executed in a function at run time is the active **CatchSignal** for that function, regardless of its lexical scope.

CHAPTER 4:
THE MAC C TOOLKIT

INTRODUCTION

The Mac C Toolkit is a collection of routines and techniques that can form the foundation for almost any application. These routines can be used in their present form, or modified for a particular program. While the Toolkit sometimes lacks the strict cohesiveness and uniformity characteristic of a library it contains powerful routines that do more than provide the functions typical of a library. There is a utilitarian rationale to the Toolkit functions. Routines which don't help are generally missing, and many of the routines ignore some of the less used features of the Macintosh.

Don't expect to use all of the Toolkit routines. Many of the routines will not not be applicable to your programs, but are documented for the sake of completeness. The names of the routines you are most likely to use are underlined. The Toolkit is meant to be responsive to the job at hand. Use it by extracting those routines which seem useful. Change the ones which don't suit the needs of a particular application. Eliminate those which are unnecessary.

The Mac C Toolkit package contains, in addition to the source and binary files for the Mac C Toolkit routines, the source files for the "standard" C library, and a program (TestLib.c) for testing the "standard" C library.

The "standard" C library source files are:

StdLib.c (includes: StdCLib.Asm (includes: InitGlobals.Asm, Math.Asm) StdMem.Asm StdStr.Asm)
StdFileIO.c (includes: MacCDefs.h StdFileDefs.h)
StdIOPrim.C (includes: MacCDefs.h StdFileDefs.h StdTTY.Asm StdSIO.Asm StdMac.Asm)

The "standard" C library is actually built upon a variation of the Toolkit. Programs can be linked and run using the "standard" C library functions by simply **StdLib** with the Toolkit files place of the library files **StdfileIO** and **StdIOprim**. The file descriptors used within the "standard" C library are the same as those in the high-level I/O system of the ToolKit, and the "standard" C library and high-level I/O routines may be employed interchangeably. For example, the Toolkit routine **TKopen**, which allows multiple buffers to be set up for a disk file, can be used instead of the "standard" C Library functions **open** or **fopen**. Note, however, that the parameters and their meanings are different. After the file is opened, **printf** or any other "standard" C Library routine can be used. The file can be closed with **TKclose, close** or **fclose**.

THE TOOLKIT I/O SYSTEM

Mac C Toolkit I/O and file operations are done through a set of common routines that provide a high-level I/O system. This is similar in nature to the I/O system used in the "standard" C Library. In fact, the I/O system in the "standard" C library is simply a paraphrasing of the Toolkit I/O system.

The file system must be initialized by calling **initio** once at the start of a program (unless **Stdlib** has been linked). Files are assigned small numbers when opened, and all I/O is performed using these numbers to identify the files. The reserved file number "0" (defined as **tty** in **MacCdefs.h**) is set up by **initio** and represents a bidirectional channel for reading characters from the keyboard and writing them to the active teletype simulation window.

Once a file has been opened, the high-level I/O routines work uniformly for all devices (taking into account device-specific characteristics). This means, for example, that by changing an open call, output can be redirected from a file to the teletype simulation window.

There is a complete set of routines for performing generic file operations in the high-level I/O system. A low-level I/O system implements routines for doing disk file I/O, serial port I/O, teletype simulation, etc. Occasionally, these lew-level routines must be accessed directly. When this is the case ***care must be exercised or the high-level I/O system may be undermined.*** For example, the high-level I/O system maintains a single character buffer for each open file, which will sometimes contain the next character to be read from the file. If the high-level I/O routines are bypassed, it is possible to miss a character, or to get a character out of sequence the next time the high-level input routine is called.

MAC C TOOLKIT FILES

The Mac C Toolkit Source files are:

Header Files

MacCDefs.h
MacCIODefs.h
MacCMemDefs.h

Source Code Files

MacCMem.c
MacCFileIO.c
MacCIOPrim.c (Do not link if MacCMinIOPrim is linked)

MacCMinIOPrim.c	(Do not link if MacCIOPrim is linked)
MacCIO.c	
MacCIOSupp	
MacCUtil.c	
MacCStrings.c	
MacCStringUtil.c	
MacCLib.Asm	(Do not link if StdLib is linked)
MacCTTYSim.Asm	
MacCSIOLib.ASM	

MAC C TOOLKIT ROUTINES

The remainder of this chapter is divided into sections which functionally group the Mac C Toolkit routines into categories:

High-level I/O
String Processing
Miscellaneous
Memory Management
Teletype Simulation
Serial I/O and Keyboard Input
Disk File Management and I/O

Routines are listed alphabetically by name within each section. Each Toolkit routine is described in a standard format: the first line contains the actual name of the routine in bold, an expanded version of this name, and the name of the source file containing the definition of the routine enclosed by brackets. The most commonly used routines are underlined. The next lines contain the C declaration syntax for the routine. Finally, the arguments and function for each routine are briefly described. For more detailed information about an individual routine, refer to the source code.

INDEX TO MAC C TOOLKIT

Sections

Routines

THE HIGH-LEVEL I/O SYSTEM

The source files for the high-level I/O system are **MacCMinIOPrim.c, MacCIOPrim.c** and **MacCFileio.c.** The header file **MacCDefs.h** contains the following global definitions used in this section:

numchannels: The maximum number of files that can be open at one time. If this is changed, recompile **MacCIOPrim.c** and/or **MacCMinIOPrim.** Use **maxfiles** in **MacCIOdefs.h** to change the maximum number of open disk files (and recompile **MacCIO.c**).

Device types (passed to **TKopen**):

serialdevice: Serial I/O Port A or B.

diskdevice: A disk file.

I/O Modes

The I/O modes passed to **TKopen** are:

read =	1
write =	2
read/write =	3

read, write, and **read/write** are not defined in **MacCDefs.h** because the names are commonly used in programs. They may be defined by the programmer. To perform normal reads and writes from from a file it must be opened according to the appropriate mode. The resource fork of a file may be read as normal data (bypassing the Resource Manager) by opening the file and adding the value **resource_fork** (defined in **MacCDefs.h**) to the mode in the call to **TKopen.**

I/O Devices

Disk files are always identified by their names. A volume name must be included when a file resides on other than the current default volume. Constants are defined for other devices:

sioAout: Serial Port A output

sioAin: Serial Port A input

sioBout: Serial Port B output

sioBin: Serial Port B input

The High Level I/O Routines

The files **MacCIOprim.c** and **MacCMiniOprim.c** contain most of the high-level I/O routines. These two files are identical, except that **MacCMiniOprim.c** has teletype and serial I/O routines stripped out to save space. A program may be loaded with one or the other, but not both. **MacCFileIO.c** contains high-level routines for file positioning and block I/O.

bytelength - byte length [MacCFileio.c]

```
long bytelength(file)
    short file;
```

This routine returns the number of bytes in the file.

closeall - close all files [MacCIoPrim.c]

```
void closeall()
```

This routine closes all open files.

crlf - carriage return/line feed [MacIOPrim.c]

```
void crlf(file)
```

This routine writes a carriage return (and line feed if necessary), to start a new line on the designated file.

echo - echo [MacIOPrim.c]

```
char echo(file, c)
    short file;
    char c;
```

This routine writes the character to the output file as a guaranteed printable character. Characters are masked to seven bits. Characters in the range 0 to 0x1F have "@" added to their value, and are printed preceded by the character "^". The value 1, for example, prints as "^A".

eof - end of file [MacIOPrim.c]

```
char eof(file)
    short file;
```

For disk files, **eof** returns true if the file is positioned at its end. For serial I/O files and **tty**, **eof** returns true if there are no characters waiting. Non-blocking input can be implemented by the following:

```
if (leof(input)) /* then there is a character waiting */
      c = inchr(input);
```

Inblock - input bytes [MacFileIO.c]

```
void inblock(file, buffer, count)
    short file, count;
    char *buffer;
```

This routine reads **count** bytes from the file to the buffer, where **count** is a short integer.

Inchr - input character [MacCIOPrim.c]

```
char inchr(file)
    short file;
```

This routine reads the next character from the input and advances the file position one byte. If the file is **tty** and the global variable **echoflag** (type **char**) is true, the charcter is echoed to the active teletype window. Note the comments under **lookch** about end of file.

Initio - initialize I/O [MacCIOPrim.c]

```
void initio(windowname)
    char *windowname;
```

This routine should be called once at the beginning of a program. It must be called after **initmem**. If **windowname** is non-zero, it is assumed to be the address of the title for a teletype window. This causes the teletype simulation package to be initialized, and a teletype window to be opened with the designated title. The WindowPtr for the teletype window is stored in the global pointer variable **console**.

Inlong - input long word [MacCIOPrim.c]

```
long inlong(file)
    short file;
```

This routine reads the next long word from the input and advances the file position four bytes. Byte alignment of the file is insignificant. Note the comments under **lookch** about end of file.

Inwd - input word [MacCIOPrim.c]

```
short inwd(file)
    short file;
```

This routine reads the next word from the input and advances the file position two bytes. Byte alignment of the file is insignificant. Note the comments under **lookch** about end of file.

Ioerror - I/O error [MacCIOPrim.c]

```
short ioerror(file)
    short file;
```

For serial devices, this routine returns the error status of the port (the cumulative error word, byte 28 of the parameter block). Briefly, the error warnings are:

B8	(Soft overrun) Buffer overflow
B12	Parity error
B13	Hard overrun
B14	Framing error

If this procedure is declared as **char ***, it returns the address of the I/O Parameter Block from the status call. (See the section on the Serial Driver in the **Inside Macintosh** manual for more information.)

linblock - long input block [MacCFileIO.c]

```
void linblock(file, buffer, count)
    short file;
    char *buffer;
    long count;
```

This routine reads **count** bytes from the file to the buffer, where **count** is a long integer.

lookch - look at character [MacIOPrim.c]

```
char lookch(file)
    short file;
```

This routine reads the next character from the input, but does not flush it. Successive calls to lookch always return the same charcter until it is read by an input routine, or the file position is changed. If a file is at
the end of file, the character 0xFF is returned. This is not the same as the standard C/UNIX convention of returning an **int** value from **getc**, which is -1 at end of file. Use the separate function **eof** to determine if a file is at its end.

loutblock - long output block [MacCFileIO.c]

```
void loutblock(file, buffer, count)
    short file;
    char *buffer;
    long count;
```

This routine writes **count** characters from the buffer to **file**, where **count** is a long.

loutdec - output long decimal [MacCIOPrim.c]

```
void loutdec(file, value)
    short file;
    long value;
```

This routine formats **value** into a signed decimal number, and writes it to the designated file.

loutnum - output number [MacCIOPrim.c]

```
void loutnum(file, value, radix)
    short file;
    long value;
    short radix;
```

This routine formats **value** into a number in base **radix**, and writes it to the designated file. If **radix** < 0, then **value** is formatted as a signed number where the radix = Abs(radix).

outblock - output block [MacCFileIO.c]

```
void outblock(file, buffer, count)
    short file;
    char *buffer;
    short count;
```

This routine writes **count** number of bytes from the buffer to the file, where **count** is a **short**.

outch - output character [MacCIOPrim.c]

```
char outch(file, value)
    short file;
    char value;
```

This routine writes **value** to the output file as a single binary byte. It returns **value** as its result. Note that the arguments are in the reverse order of those in **putc**. Use a macro to switch them if necessary.

outdec - output decimal [MacCIOPrim.c]

```
void outdec(file, value)
    short file, value;
```

This routine formats **value** into a signed decimal number, and writes it to the designated file.

outhexbyte - output hex byte [MacCIOPrim.c]

```
void outhexbyte(file, value)
     short file;
      char value;
```

This routine writes **value** as two ASCII hexadecimal digits onto **file.**

outhexlong - output hex long [MacCIOPrim.c]

```
void outhexlong(file, value)
    short file;
    long value;
```

This routine writes value as eight ASCII hexadecimal digits onto **file.**

outhexword - output hex word [MacCIOPrim.c]

```
void outhexword(file, value)
    short file;
    short value;
```

This routine writes **value** as four ASCII hexadecimal digits onto **file.**

outlong - output long [MacCIOPrim.c]

```
long outlong(file, value)
      short file;
      long value;
```

This writes **value** to the output file as four binary bytes. It returns **value** as a result. Note that the arguments are in the reverse order of those in **putl.** Use a macro to switch them if necessary.

outnum - output number [MacCIOPrim.c]

```
void outnum(file, value, radix)
    short file;
    short value;
    short radix;
```

This routine formats **value** into a number in base **radix**, and writes it to the designated file. If **radix** < 0, then **value** is formatted as a signed number where: radix = Abs(radix).

outstr - output string [MacCIOPrim.c]

```
void outstr(file, str)
    short file;
    char *str;
```

This routine writes the C string **str** to **file**. Note that the arguments are in the reverse order of those in **puts**. Use a macro to switch them if necessary.

outwd - output word [MacCIOPrim.c]

```
short outwd(file, value)
    short file;
    short value;
```

This writes **value** to the output file as two binary bytes. Returns **value** as its result. Note that the arguments are in the reverse order of those in **putw**. Use a macro to switch them if necessary.

readbytepos - read byte position [MacCFileIO.c]

```
long readbytepos(file)
    short file;
```

This routine returns the current byte position of **file** relative to its start. The first position is 0.

readline - read line [MacCIOPrim.c]

```
char *readline(file, buffer, size)
    short file;
    char *buffer;
    short size;
```

This routine reads data from the file until a newline character or EOF is encountered, and stores it, terminated by a **NULL**, into the buffer. If **file** is **tty**, then the characters are echoed to the TTY window as they are read, and backspace causes the previous character to be erased and not saved in the buffer. **size** is the maximum number of characters which may be stored in the buffer, although more than this may be read from **file**. Excess characters are discarded.

setbytepos - set byte position [MacCIOPrim.c]

```
setbytepos(file, position)
    short file;
    long position;
```

This routine sets the position of the file so that the next byte read will be **position** number of bytes from the start of the file. The position of the first byte in a file is 0. If **position** is past the end of file, the file position will be set to the end of file if it is opened for read access only. If the file is open for write access or read/write access, the file position will be extended to **position**.

seteof - set end of file [MacCIOPrim.c]

```
short seteof(file)
    short file;
```

This routine sets the end of file position for file to the current position. Only disk files open for write or read/write can use this routine. It returns **file** as its result.

seteof can be used with open to initialize an existing file:

```
#define write 2
    file = seteof(open(diskdevice, "filename", write));
```

TKclose - close a file [MacCIOPrim.c]

```
short TKclose(file)
    short file;
```

This routine closes the designated file and flushes its buffers to disk if necessary. The file number is invalid after a close. TKclose always returns 0.

TKopen - open a file [MacCIOPrim.c]

```
short TKopen(deviceType, name, mode)
    char deviceType;
    char *name;
    long mode;
```

This is the generic open routine. It returns a small positive number (the file number) if it succeeds, and a 0 if it fails. Use the file number as an argument to all other high-level I/O calls.

A file's characteristics are set up at open time. Since devices vary in their characteristic definition, the exact interpretation of the paramaters **name** and **mode** depend on **deviceType**.

When the device is a disk (**deviceType = diskdevice**), **name** is a pointer to a C string which is the file name. The file name must include the volume name if the file is not on the default volume.

If **mode** is in the range 0 to 255, it is an access mode as follows:

1 = read
2 = write
3 = read/write

Add the constant **resource_fork** to the access mode to open the resource fork of a file rather than the data fork. If **mode** is not in the range 0 to 255, it is assumed to be the address of the structure:

```
struct {
  char filler[3];
  char mode;                    /*access mode*/
  long buffersizeA;    /*number of bytes in buffer A*/
  long buffersizeB;             /*number of bytes in buffer B*/
```

This causes the disk file to be opened with two buffers whose sizes are specified by **buffesrsizeA** and **buffersizeB**. If **buffersizeB** is 0, the file is opened with a single buffer. To be efficient, buffer sizes should be multiples of 512. If two buffers are specified, disk I/O to and from the file uses the asynchronous I/O option of the Macintosh, which allows simultaneous processing and I/O.

When the device is a serial port (**deviceType = serialdevice**), **name** is a constant defining the SIO port to be opened (**sioAin, SioAout**, etc.) It is important to always open a serial port in the output direction, since that is when the Macintosh operating system initializes it. You need open it in the input direction only if you wish to input data from it. ***Do not close a serial port once it has been opened.*** Use the **PBControl** trap to change the baud rate after it has been opened (the Toolkit routine **InitSIO** gives an example of how to do this.)
The lower 16 bits of **mode** are the Macintosh SIO configuration word (defined in the Serial Driver Section of the **Inside Macintosh** manual). This word allows the baud rate, number of stop bits, parity, and number of data bits to be set. A port is initially opened with XON/XOFF disabled, CTS disabled, and with all errors reported. Use the **PBControl** trap to change this setting (again, **InitSIO** can be used as an example.)

A serial port cannot be opened for different baud rates on input and output. The configuration used to open a port is valid until it is changed. If mode = 0, the configuration is assumed to be 0xCC0A (9600 baud, 8 data bits, 2 stop bits, no parity).

If you have a hard disk, be careful! It may use a serial port, so do not open it.

uinchr - upper case input character [MacCIOPrim.c]

```
char uinchr(file)
    short file;
```

This routine is just like **inchr**, except that if the character read is lower case, **inchr** forces the character to be upper case. Note the comments under **lookch** about end of file.

String Routines

All of these routines operate on C strings except where noted. They make use of a global variable named **temp_str**, which is used as a holding area for temporary strings. Temporary strings are allocated by some of these routines. A temporary string exists only until the next routine requiring use of **temp_str** is called. The program is responsible for saving copies of temporary strings.

Storage for strings is allocated by calling **getmem**. The caller of the string routine is responsible for deallocating string storage by calling **retmem** when he is finished using them.

Pascal strings are described in this documentation by:

```
typedef struct {char count; char contents[255];} P_Str;
```

Most of these routines are written in assembly languge (for speed), so a large overhead is not incurred by using them.

apchr - append character [MacCStrings.c]

```
char apchr(str, c)
char *str;
char c;
```

This routine appends **c** followed by a **NULL** to the end of **str**. It returns **c** as its result.

aplong - append formatted long [MacCStrings.c]

```
void aplong(str, number, radix)
char *str;
short radix;
long number;
```

This routine converts long integers into ASCII text and appends it to **str**. If **radix** < 0, then a signed conversion is performed, and the **radix** is **-radix**. A **radix** of 16 produces a hexadecimal number.

apnum - append formatted short [MacCStrings.c]

```
void apnum(str, number, radix)
    char *str;
    short radix;
    short number;
```

This routine converts 16-bit **short** integers into ASCII text, and appends it to **str**. If **radix** < 0, then a signed conversion is done, and the **radix** is **-radix**. A **radix** of 16 results in a hexadecimal number.

apstr - append string [MacCStrings.c]

```
char *apstr(str, str1)
char *str, *str1;
```

This routine appends the string addressed by **str1** followed by a **NULL** onto the the end of string **str**. The **str** parameter is returned.

CloneString - Make a copy of a string [MacCStringUtil.c]

```
char *CloneString(str)
char *str;
```

This routine makes a copy of **str**, and returns its address. **CloneString** calls **getmem** to allocate space for the copy, so **retmem** should be used to release this space after the caller is through with the result.

comparestr - compare string [MacCStrings.c]

```
char comparestr(str1, str2)
char *str1, *str2;
```

This routine compares the string addressed by **str1** to the string addressed by **str2** according to their sort sequence. **comparestr** returns a non-zero result if **str1** < **str2**. If **str1** is shorter than **str2** and they are equal up to the length of **str2**, it returns a non-zero result. Otherwise it returns a result of zero.

ConcatString - concatenate string [MacCStringUtil.c]

```
char *ConcatString(str1, str2)
    char *str1, *str2;
```

This routine allocates memory for a new string, and fills it with the contents of **str1** followed by the contents of **str2** (and terminated with a **NULL**). The address of the new string is returned. Since **ConcatString** calls **getmem** to allocate space for the copy, **retmem** should be used to release this space after the caller is through with the result.

copystring - copy string [MacCStrings.c]

```
char *copystring(str1, str2)
    char *str1, *str2;
```

This routine copies the string addressed by **str1** to the area pointed to by **str2** (including the terminating **NULL**). The area addressed by **str2** must have been previously allocated (be sure it is large enough to hold **str1**). The parameter **str2** is returned.

equalstr - equal string [MacCStrings.c]

```
char equalstr(str1, str2)
    char *str1, *str2;
```

This routine compares the string addressed by **str1** to the string addressed by **str2**,and returns a non-zero result if the contents are identical. Otherwise it returns zero.

findchar - find character [MacCStrings.c]

```
short findchar(str, c)
    char *str;
    char c;
```

This routine returns the index+1 of the first occurrence of **c** in **str**. If **c** is not found, 0 is returned.

LC - lower case [MacCStrings.c]

```
char LC(c)
    char c;
```

If **c** is an upper case character, this routine returns its lower case counterpart. Otherwise, **c** is returned.

length - length [MacCStrings.c]

```
int length(str)
    char *str;
```

This routine returns the number of bytes in **str** preceding the first **NULL** byte.

MacString - convert a C string to a Pascal String [MacCStringUtil.c]

```
P_Str *MacString(str)
    char *str
```

This routine allocates space for a string and converts **str** into it in Pascal string format. This is useful for calling Macintosh system routines which require Pascal format strings as arguments. It calls **getmem** to obtain storage, and expects the programmer to deallocate it with **retmem**. Its result is the address of the converted string.

setlowercase - set string lower case [MacCStrings.c]

```
char *setlowercase(str)
    char *str;
```

This routine converts upper case characters in **str** to lower case. It returns **str**.

setuppercase - set string upper case [MacCStrings.c]

```
char *setuppercase(str)
    char *str;
```

This routine converts lower case characters in **str** to upper case. It returns **str**.

strtolong - string to long [MacCStrings.c]

```
long strtolong(str, radix)
    char *str;
    short radix;
```

This routine converts the ASCII number in **str** into a **long** binary value and returns this value as its result. Legal radices are 2 through 10, and 16 (for hexadecimal values). Conversion stops at the first non-numeric character for the given radix. Overflow is lost.

strtonum - str to number [MacCStrings.c]

```
short strtonum(str, radix)
char *str;
short radix;
```

Identical to **strtolong**, except that the result is a 16 bit short value.

tempCStr - Convert Pascal string to C string [MacCStringUtil.c]

```
char *tempCStr(MacStr)
P_str * MacStr;
```

This routine converts **MacStr** into C string format. This is useful for converting results from Macintosh system calls into C string format. It uses temporary storage, so do not deallocate the memory when you are through. To save the string, make a copy of it (using **CloneString**). This routine returns the address of the temporary string as its result.

tempMacStr - Convert C string to Pascal string [MacCStringUtil.c]

```
P_Str *tempMacStr(str)
char *str;
```

This routine allocates space for a string and converts **str** into it in Pascal string format. This is useful for calling Macintosh system routines which take Pascal strings as arguments. Do not deallocate the memory when you are through with **str**. The system will do that on the next call to a routine that uses a temporary string. Use **MacString** if

you want a string which can be saved. This routine returns the address of the temporary string as its result.

UC - convert character to upper case [MacCStrings.c]

```
char UC(c)
char c;
```

If **c** is a lower case character, this routine returns its upper case counterpart. Otherwise, it returns **c** as its result.

Miscellaneous System Routines

This section contains miscellaneous routines.

Bounds - bound a value [MacCUtil.c]

```
short Bounds(value, lower, upper)
        short value, lower, upper;
```

If **lower <= value <= upper** then **value** is returned. Otherwise Bounds returns **lower** if **value < lower**, or **upper** if **value > upper**, i.e., it returns MIN(upper, MAX(value, lower)).

clearbytes - clear memory [MacCUtil.c]

```
clearbytes(address, nbytes)
        char *address;
        short nbytes;
```

This routine sets the value of **nbytes** of memory to 0 starting at **address**.

fillbytes - fill memory with value [MacCUtil.c]

```
fillbytes(value, address, nbytes)
        char *address;
        short nbytes;
        char value;
```

This routine sets the value of **nbytes** of memory to **value** starting at **address**.

Inrange - test in range [MacCUtil.c]

```
char inrange(value, lower, upper)
        short value, lower, upper;
```

This routine returns non-zero if **lower <= value <= upper**. It returns zero otherwise.

LBounds - bound a value [MacCUtil.c]

```
long LBounds(value, lower, upper)
        long value, lower, upper;
```

If **lower <=value <= upper**, this routine returns **value**. Otherwise, it returns **lower** if **value < lower**, or **upper** if **value > upper**. In other words, it returns LMIN(upper, LMAX(value, lower)).

letterdigit - test for letter or digit [MacCUtil.c]

```
char letterdigit(c)
        char c;
```

This routine returns non-zero if **c** is an alphabetic character or a digit.

Linrange - test in range [MacCUtil.c]

```
char Linrange(value, lower, upper)
        long value, lower, upper;
```

This routine returns non-zero if **lower <= value <= upper**; it returns zero otherwise.

LMAX - long maximum [MacCUtil.c]

```
long LMAX(v1, v2)
    long v1, v2;
```

This routine returns the signed arithmetic maximum of **v1** and **v2**.

LMIN - long minimum [MacCUtil.c]

```
long LMIN(v1, v2)
    long v1, v2;
```

This routine returns the signed arithmetic minimum of **v1** and **v2**.

MAX - short maximum [MacCUtil.c]

```
short MAX(v1, v2)
    short v1, v2;
```

This routine returns the signed arithmetic maximum of **v1** and **v2**.

MIN - short minimum [MacCUtil.c]

```
short MIN(v1, v2)
    short v1, v2;
```

This routine returns the signed arithmetic minimum of **v1** and **v2**.

movebytes - move bytes [MacCUtil.c]

```
movebytes(nbytes, dest, source)
    short nbytes;
    char *dest, *source;
```

This routine moves **nbytes** from **source** to **dest** using the **BlockMove** trap. If **source** or **dest** is 0, it does nothing.

SwapLong - swap long values [MacCUtil.c]

```
SwapLong(ptrA, ptrB)
    long *ptrA, *ptrB;
```

Exchanges the long values addressed by **ptrA** and **ptrB**.

UBounds - bound an unsigned value [MacCUtil.c]

```
unsigned long UBounds(value, lower, upper)
    unsigned long value, lower, upper;
```

If **lower <= value <= upper**, then **UBounds** returns **value**. Otherwise, it returns **lower** if **value < lower**, or **upper** if **value > upper**. In other words, it returns UMIN(upper, UMAX(value, lower))

UMAX - unsigned maximum [MacCUtil.c]

```
unsigned long UMAX(v1, v2)
    unsigned long v1, v2;
```

This routine returns the unsigned arithmetic maximum of **v1** and **v2**.

UMIN - unsigned minimum [MacCUtil.c]

```
unsigned long UMIN(v1, v2)
    unsigned long v1, v2;
```

This routine returns the unsigned arithmetic minimum of **v1** and **v2**.

wait - wait for time [MacCUtil.c]

```
wait(ms)
    int ms;
```

This routine waits **ms** milliseconds and returns. Accuracy depends on the tick rate (1/60 second).

Memory Management Routines

In general, these routines use the Macintosh memory manager to allocate memory. Memory is always allocated as non-relocatable blocks that begin on word boundaries, are an even number of bytes in length, and are less than 32,768 bytes long. These blocks are referenced by the address of the first byte.

The memory management routines act as a level of insulation between the user and the Macintosh memory manager. This buffering provides transportability and the option of using a more sophisticated or alternate allocator. The **allocate** routines (**allocate**, **initallocate**, and **resetarea**) are particularly space efficient in that they can allocate memory with no overhead bytes (although all such allocated areas must be returned at the same time or in the reverse order of allocation). This capability is useful for structures such as symbol tables which have the characteristic of growing monotonically and then being eliminated as a single entity.

allocate - allocate region [MacCMem.c]

```
char *allocate(area, Size);
    struct AREA *area;
    short Size;
```

This routine allocates a region **Size** bytes long from **area** using a simpleton allocator. **area** must have previously been obtained by a call on **initallocate**.

getmem - get memory [MacCMem.c]

```
char *getmem(size)
    short size;
```

This routine allocates the indicated number of bytes of memory and returns a pointer to the first byte. It calls **Signal** with the string "Out of Memory" if it can't accommodate a request.

getmemz - get memory [MacCMem.c]

```
char *getmemz(size)
    short size;
```

This routine is identical to **getmem**, except that it sets the allocated memory to 0 before returning.

Initallocate - initialize for allocate [MacCMem.c]

```
struct AREA *initallocate(Size)
    short Size;
```

This routine allocates a block of memory using **getmem** and prepares it for use by the simpleton allocator **allocate**. The structure **AREA** is declared in the file **MacCmemdefs.h**. The total space allocated to the area for **allocate** to use is **Size** bytes (forced to be even, i.e., (size+1)&-2).

Initmem - initalize memory [MacCMem.c]

```
void initmem()
```

This routine initializes **getmem**, and sets up the Macintosh application heap so that there is a null **GrowZone** Procedure. The application heap limit is set to 8K bytes below the current top of stack. Call this once at the start of a program.

MemAvail - maximum available memory block [MacCMem.c]

```
long MemAvail()
```

Just like the Macintosh trap **MaxMem**, except that this one returns either the largest block available or the space left to grow, whichever is greater.

resetarea - reset area [MacCMem.c]

```
void resetarea(area)
    struct AREA *area;
```

This routine returns all allocated space from **area**, and prepares it for future **allocate** calls.

retmem - free memory [MacCMem.c]

```
char *retmem(memory)
    char *memory;
```

This routine returns the area addressed by **memory** (which must have been previously allocated by **getmem**) to the free list. If the pointer **memory** is equal to 0, **retmem** does nothing. **retmem** always returns a result of 0.

THE LOW LEVEL I/O SYSTEM

Teletype Simulation

Any window can be a teletype simulation window. The **CreateTTY** function, or the **NewWindow** or **GetNewWindow** traps can be used to create such a window. Only one teletype window can be active at one time, and any output directed to **tty** (file 0) appears in that window. The **SetTTY** routine can be used to change windows. Characters are displayed in the current font, style, etc., of the active teletype simulation window. The next character will be displayed at the pen position of the window's port.

BackupTTY - backspace character [MacCTTY.Asm]

```
void BackupTTY(c)
    char c;
```

This routine backs up the character position and erases **c**, which is normally the last character displayed.

ClearTTY - clear teletype window [MacCTTY.Asm]

```
void ClearTTY(window)
    WindowPtr window;
```

This routine erases the entire contents of **window** and places the pen at the upper left hand corner of the window.

CreateTTY - create teletype window [MacCTTY.Asm]

WindowPtr CreateTTY(x, y, width, height, title, goawayFlag)

```
short x, y;
short width, height;
char *title;
char goawayFlag;
```

This routine creates a window and sets it up as the active teletype window. **x** and **y** locate the upper left hand corner, **title** is a **P-Str**, and **goawayFlag** is the window goawayflag. **width** and **height** are the width and height of the window in screen dots.

Font_Info - font information [MacCTTY.Asm]

FontInfo Font_Info()

This routine returns the address of the **FontInfo** record for the current font of the current port. The **FontInfo** structure is defined in **Inside Macintosh**, and in the header file **font.h**. The result of this function is saved in temporary storage. Do not dispose of it. Copy the structure if you wish to save it.

Line_Height - line height [MacCTTY.Asm]

short Line_Height()

This routine returns the line height in screen dots for the current teletype window. This is the sum of the leading, ascent, and descent.

SetTTY - set TTY [MacCTTY.Asm]

```
void SetTTY(window)
    WindowPtr window;
```

This routine makes **window** the active teletype simulation window. Characters are displayed from the current pen position.

TTYChar - display character in teletype window [MacCTTY.Asm]

```
void TTYChar(c)
    char c;
```

This routine displays **c** as the next character in the active teletype window.

TTYPort - teletype window pointer [MacCTTY.Asm]

```
WindowPtr TTYPort;
```

This global variable contains the WindowPtr for the active teletype window.

Serial I/O and Keyboard Input

The low-level I/O routines mentioned here take a CONSOLE or SIO device as an argument. This is not the device used for normal Toolkit I/O; rather, it is an internal Toolkit value. To call these routines directly, use the value saved in **channelvec[file]**, where **file** is the parameter returned by **TKopen**, and the type of **channelvec** is **int**. Do not use these routines for disk I/O.

EventReady - event ready [MacCSIOLib.Asm]

```
char EventReady(mask)
    short mask;
```

This routine returns non-zero if there is an event of the type specified by **mask** waiting in the event queue. **mask** is the event mask as described in **Inside Macintosh**.

FlushKey - flush key [MacCSIOLib.Asm]

```
void FlushKey()
```

This routine flushes all **keydown** and **autokey** events from the event queue.

InEvent - input Event [MacCSIOLib.Asm]

```
EventRecord *inEvent(mask, Event)
    short mask;
    EventRecord *Event;
```

This routine reads the next event of the type specified by **mask** into **Event**, and removes it from the event queue. If there are no such events in the event queue, the routine waits for one. Its result is **Event**.

InitSIO - initalize SIO port [MacCSIOLib.Asm]

```
void InitSIO(device, configuration)
    short device, configuration;
```

This routine initializes the SIO port. If **configuration** is 0, the port is initialized as follows: 2 stop bits, 8 data bits, no parity, 9600 baud. Otherwise, **configuration** is passed to the Macintosh operating system as the configuration parameter to a **PBControl** trap (see the Serial Driver section in **Inside Macintosh**). **InitSIO** is normally called by **TKopen**. ***If you have a hard disk, be careful! It may use a serial port, so do not initialize it.***

InKey - input key event [MacCSIOLib.Asm]

```
char inKey()
```

This routine reads the next **keydown** or **autokey** event from the event queue, and returns its character value. The Command key is treated as a control shift, producing characters in the range 0 through 0x1F. If the keyboard event queue is empty, **InKey** waits for a key event.

InSIO - input SIO character [MacCSIOLib.Asm]

```
char inSIO(device)
    short device;
```

This routine reads a single character from **device** (CONSOLE or an SIO port) and returns it. ***Do not use this directly and expect the Toolkit I/O Routines to work with the same device.***

KeyReady - key ready [MacCSIOLib.Asm]

char KeyReady()

This routine returns non-zero if there is a character in the input buffer for the keyboard.

SIOReady - SIO ready [MacCSIOLib.Asm]

char SIOReady(device)
short device;

This routine returns non-zero if there is a character in the input buffer for **device**, where **device** may be the CONSOLE or an SIO port.

Disk I/O Routines

These routines form the basis of the Mac C Toolkit disk I/O system. They interface to the Macintosh I/O system at the block file I/O level, maintaining their own file position and end-of-file information for open files. They allow either synchronous or asynchronous (multiply buffered) I/O, and provide a complete set of primitives for reading, writing, and positioning files. Particular care has been taken to make them efficient at the character I/O level so that applications can directly call the I/O routines without a serious loss of efficiency. These routines are used for all I/O done in the Mac C Compiler. **Most of these routines are not useful for normal I/O**. Use the high-level I/O routines instead.

Files at this level are identified by a File information Block (**FIB**). This structure (which is declared in **MacCIOdefs.h**) contains all the information that is known about an open file, and is passed as an argument to most of the routines.

The high-level I/O system (which uses **TKopen, TKclose, Inchr, outch**, etc.) saves the address of the **FIB** in **channelvec,** which is in turn indexed by the file number used in the high-level I/O routines.

A File Control Block (**FCB**) data structure (which is actually a Macintosh I/O Parameter Block) contains the information required by the Macintosh operating system about the file. It is used as a parameter to many low-level routines. The address of the **FCB** for a file is stored in its **FIB**.

The strings manipulated in these routines are all C strings unless otherwise noted. Conversion is performed by the routines as required for the Macintosh operating system calls.

I/O system errors are normally handled by calling **Signal** or by failing the operation. Failed operations return an invalid value andr store an error code in the global variable **lastIOError**. This is satisfactory for many applications, but not for all. For these latter cases the **FIB** contains a field named **IOErrorLoc**. If the address of a function is stored in this field (by calling **setIOError**), that function will be called when a serious error (such as diskette write protected or full) is encountered that cannot be handled by Toolkit routines. The function is called with arguments as though it were declared:

```
IOError(fib, result)
  struct FIB *fib;
  short result;
```

result is the result code from the Macintosh file system trap. Returning from the **IOError** function causes the error to be ignored by the I/O system.

callIOError - call I/O error [MacCIO.c]

```
void callIOError(fib, errorcode)
    struct FIB *fib;
    short errorcode;
```

This routine calls the error function for **fib** if there is one. **errorcode** contains the Macintosh trap result code.

deletefile - delete file [MacCIO.c]

```
short deletefile(filename)
    char *filename
```

This function deletes the named file from the disk if it exists and is not open. Otherwise, **deletefile** does nothing. Returns the result from the Macintosh Delete trap as its result (zero if everything was OK).

eoffile - test end of file [MacCIO.c]

```
char eoffile(fib)
    struct FIB *fib;
```

eoffile returns non-zero if the designated file is positioned at the end.

filelength - file length [MacCIO.c]

```
long filelength(fib)
    struct FIB *fib;
```

This routine returns the length in bytes of the designated file.

findfile - find file [MacCIO.c]

```
char findfile(filename)
    char *filename
```

This routine returns non-zero if the named file exists on disk.

initfileio - initialize file I/O [MacCIO.c]

```
void initfileio()
```

This routine is called by **initio** to initialize the disk file system.

readchar - read character [MacCIO.c]

```
char readchar(fib)
    struct FIB *fib;
```

This routine reads and returns the next character from the designated file. If the file is at the end, or if it is not open for read access, the value 0xFF is returned.

setfileposition - set file position [MacCIO.c]

```
long setfileposition(fib, pos)
    struct FIB *fib;
    long pos;
```

This routine sets the byte position of the designated file to **pos**. The first position in the file is 0. If **pos** is past the current end of file, the position is set to end of file for read only access files, and the file is extended to **pos** for write access or read/write access files.

setIOError - set I/O Error [MacCIO.c]

```
setIOError(fib, errorproc)
    struct FIB *fib;
    int (*errorproc)();
```

This routine makes **errorproc** the function called when serious I/O errors occur on the file designated by **fib**.

Define **errorproc** as:

```
errorproc(fib, result)
    struct FIB *fib;
    short result; // result is the error code returned by the Macintosh file system.
```

Call **setIOError** with the name of the **errorproc**:

```
setIOError(fib, errorproc);
```

This supercedes any previous calls to **setIOError** for **fib**.

truncatefile - truncate file [MacCIO.c]

```
struct FIB *truncatefile(fib)
    struct FIB *fib;
```

If the designated file is open for write or read/write access, its end of file pointer is set equal to the current position. Otherwise this routine does nothing. It returns **fib** as its result.

writechar - write chracter [MacCIO.c]

```
char writechar(fib, c)
    struct FIB *fib;
    char c;
```

This routine writes **c** onto the file designated by **fib** if it is open for write or read/write access. If the file is at end of file, it is extended to accommodate the character. If the file is not opened for write or read/write access, this routine does nothing and a byte with a value of 0xFF is returned as the result. Otherwise **c** is returned.

Routines Which Directly Call Macintosh I/O Traps

GetFileType - get file type [MacCIOSupp.c]

```
char GetFileType(filename, typePtr )
    char *filename;
    long *typePtr;
```

This routine returns the file type ('TEXT', 'APPL', etc.) of the file identified by **filename** in the **long** addressed by **typePtr**. It returns non-zero as its result if the file exists, and zero otherwise.

IOCall - I/O Call [MacCIOSupp.c]

```
short IOCall(trap, parmBlock)
    short trap;
    char *parmBlock;
```

This routine performs a generic call to the Macintosh I/O system. The number of the trap to be called is passed as **trap**, and **parmBlock** is a pointer to the I/O parameter block to be passed to the trap in A0. **IOCall** normally returns the result of the trap as its result. It stores any Macintosh error code in the global variable **lastIOError**. If an asynchronous trap results in an "Unmounted volume" I/O error, **IOCall** retries the trap as a synchronous call. If the retry fails, it calls **Signal** with the string "Could Not Re-Mount Volume".

MACbytelo - Mac byte I/O [MacCIOSupp.c]

```
long MACbytelo(inst, fcb, addr, position, nbytes)
    short inst;
    struct FCB *fcb;
    char *addr;
    long position;
    short nbytes;
```

This routine performs a read or write operation on the indicated file, and returns the actual number of bytes written or read. **Inst** contains the value of either a read or write trap and is passed directly to **IOCall**.

MACgeteof - get EOF [MacCIOSupp.c]

```
long MACgeteof(fcb)
    struct FCB *fcb;
```

This routine returns either the logical end of file for the indicated file, or zero if the file doesn't exist or if there is an error.

MACseteof - set EOF [MacCIOSupp.c]

```
short MACseteof(fcb, position)
    struct FCB *fcb;
    long position;
```

This routine sets the logical end of file of the indicated file to **position**. It returns the result code from the Macintosh **SetEOF** trap as its result.

OSclose - close file [MacCIOSupp.c]

```
short OSclose(fcb)
    struct FCB *fcb;
```

This routine closes the indicated file. It returns the result code from the Macintosh **Close** trap as its result.

OScreate - O.S. create file [MacCIOSupp.c]

```
char OScreate(filename)
    char *filename;
```

This routine creates a file in the directory with the name in **filename**. It returns non-zero if the operation is successful, zero if it is not.

OSdelete - O.S. delete file [MacCIOSupp.c]

```
char OSdelete(filename)
    char *filename;
```

This routine deletes the file named by **filename** from the directory. It returns 0 if the operation is successful, an error code from the Macintosh I/O system if it is not.

OSfind - O.S. find file [MacCIOSupp.c]

```
char OSfind(filename)
    char *filename;
```

This routine returns a non-zero result if the file named by **filename** exists in the directory. It returns false otherwise.

OSflush - O.S. flush [MacCIOSupp.c]

```
short OSflush(fcb)
    struct FCB *fcb;
```

This routine ensures that all file buffers of the indicated file are flushed to disk. (It should not be necessary to call **OSflush**, since the file system does so at the appropriate times.) It returns the result code from the Macintosh **FlushFile** trap.

OSopen - O.S. open file [MacCIOSupp.c]

```
short OSopen(fcb, filename, access, resourceflag)
    struct FCB *fcb;
    char *filename;
    short access;
    char resourceflag;
```

This routine opens the file indicated by **filename** (a C string) in the mode indicated by **access** (0 = read/write if allowed, 1 = read only, 2 = write only, 3 = read/write). If **resourceflag** is non-zero, the resource fork of the file is opened instead of the data fork. The result of the Macintosh **Open** trap (zero = operation is successful, non-zero = error code) is returned.

OSreset - O.S. reset [MacCIOSupp.c]

```
short OSreset()
```

This routine ensures that all file buffers and directories are flushed to disk on any mounted volumes. Call this before leaving a program (unless you call the "standard" C library routine **exit** or **_exit**). Note that this routine does not close files. The high-level I/O System function **closeall** can be used to close any open files. **OSreset** returns the result code from the Macintosh **FlushVol** trap as its result.

SetFileSignature - set file signature [MacCIOSupp.c]

```
void SetFileSignature(filename, Signature)
    char *filename;
    long Signature;
```

This routine sets the creator of the file named by **filename** to **Signature**.

SetFileType - file type [MacCIOSupp.c]

```
void SetFileType(filename, Type)
    char *filename;
    long Type;
```

This routine sets the type of the file named by **filename** to **Type**.

UserWords - get user words [MacCIOSupp.c]

```
char *UserWords(filename, IOParmBlock)
    char *filename;
    char *IOParmBlock;
```

Given name pointer **filename** and the address of an I/O Parameter Block in **IOParmBlock**, this routine fills in the I/O Parameter Block by calling **GetFileInfo**, and returns the address of the userwords field within the I/O Parameter block as its result. The area pointed to by **IOParmBlock** must be at least 80 bytes long. **UserWords** returns 0 if the file does not exist.

CHAPTER 5:
FLOATING POINT

INTRODUCTION

The floating point package used by Mac C is based on the IEEE floating point package implemented in the Standard Apple Numerics Environment (SANE). It is a sophisticated and highly accurate package using 80 bit operands. In the Mac C implementation, which conforms to an Apple specification for a C implementation of SANE, the normal C floating point data types **float** and **double** are augmented by two new data types, **extended**, which is a full precision 80 bit floating point value, and **comp**, which is a 64 bit integer type implemented by the SANE package. This chapter discusses the features of the Mac C implementation of SANE where they differ from (or are extensions to) the standard C floating point implementation. ***Programs written according to the Kernighan and Ritchie description of floating point will run, unmodified, in the Mac C floating point implementation.*** You should be familiar with the section of Inside Macintosh which discusses floating point and SANE. A special thanks is due to the Numerics group at Apple Computers, Inc., for their assistance and guidance in implementing the SANE C Numerics.

FLOATING POINT TYPES

Four data types are encompassed by the Mac C floating point implementation: **float** (32 bits), **double** (64 bits), **comp** (64 bits), and **extended** (80 bits). All floating point computations are done in **extended** precision. The best performance will be obtained by using **extended** precision operands, since any other type will require conversion before use.

NaNs AND INF

The IEEE floating point standard defines two special quantities, **NAN**s and **INF**. **INF** stands for infinity, and can be signed (i.e. +INF, -INF). **NaN** stands for "Not a Number". This is normally the result of an operation which yields an undefined result, e.g. sqrt (-1). **NaN**s have a value associated with them, which identify their origin. The defined **NaN** codes and their origins are:

NaN	Origin
NaN(1)	Invalid Square Root
NaN(2)	Invalid Addition
NaN(4)	Invalid Division
NaN(8)	Invalid Multiplication

NaN	Origin
NaN(9)	Invalid Remainder
NaN(17)	Conversion of invalid ASCII String
NaN(20)	Conversion of **comp** NaN to extended
NaN(21)	Attempt to create **NaN(0)**
NaN(33)	Invalid Argument to **sin**, **cos**, or **tan**
NaN(34)	Invalid Argument to **atan**
NaN(36)	Invalid Argument to log function
NaN(37)	Invalid Argument to exp function
NaN(36)	Invalid Argument to financial function

CONSTANTS

Floating point constants are converted to binary at compile-time. Integer constants which are greater than the maximum long integer value are read as floating integers for use as **comp** values. The maximum integer is now 9,223,372,036,854,775,807. Constant expressions, including negated numbers, are evaluated at run-time. **INF** and **NaNs** cannot be input as constants (since they would be interpreted by the compiler as identifiers and function calls). The floating point library functions **nan()** and **Inf()** produce these special values. Integer constants may be used in floating point expressions, and they may be assigned to floating point values. This is not always efficient, since they will normally be compiled as integer constants, and then converted at run-time to their floating point forms.

VARIABLE INITIALIZATION

Floating point variables may be initialized when defined just like integer variables, except that static and external variables may only be initialized to floating constants. Expressions are illegal, and so are integer constants. Automatic floating point variables may be initialized to anything.

OPERAND CLASSES

Any floating point operand (whether **extended**, **float**, **double**, or **comp**) belongs to one of the following classes: Signalling NaN, quiet NaN, infinite, zero, normalized, or denormalized. Floating point library routines allow the user to determine the class of a floating point operand. See Inside Macintosh for a precise explanation of the meaning of the classes.

OPERATORS

Floating point operands are valid with the operators in table 5.1. All of the operators retain their customary meanings except that the relational operators have an extra twist. Given two values, A and B, we normally can say that either A > B, A < B, or A = B. Since we have introduced floating point values, **NaNs**, which do not represent numbers, we have another possibility. If either A or B is a **NaN**, we say that A and B are ***unordered***. All ***unordered*** comparisons are false except for not equal. This means, for example, that if the relation A < B is false, it is not necessarily true that A >= B. A and B may be ***unordered***.

Table 5-1. C Floating Point Operators

Arithmetic Operators		Unary Operators	
+	Addition	&	Address of Operand
-	Subtraction	-	Arithmetic Negate
*	Multiplication	++	Increment
/	Division	--	Decrement
%	Modulus	(type)	Cast (Type Conversion)
		sizeof	Size of Object (bytes)
Relational Operators		**Assignment Operators**	
>	Greater Than	=	Simple Assignment
>=	Greater Than or Equal To	+=	Add, then Assign
<	Less Than	-=	Subtract, then Assign
<=	Less Than or Equal To	*=	Multiply, then Assign
==	Equal To	/=	Divide, then Assign
!=	Not Equal To	%=	Modulus, then Assign

THE ENVIRONMENT

Although this is discussed in detail in Inside Macintosh, a brief description is appropriate here. Three things are addressed by the environment: Rounding control, exception reporting, and halts. The sane library routines **getenvironment** and **setenvironment** are generic routines for controlling the environment.

Rounding direction can be set as being to nearest (the default), upward, downward, or toward zero. Rounding precision is normally set for extended operands, but can be set for float or double operands if required by an algorithm. The sane library routines for controlling rounding are **setround** and **getround**, **getprecision**, and **setprecision**.

An exception occurs when an anomolous condition arises in a computation, and it sets an exception flag. Reported exceptions are invalid (invalid operand(s)), underflow, overflow, divide by zero, and inexact (indicating rounding error). Exceptions are sensed and controlled by the sane library routines **testexception** and **setexception.**

There is a capability to treat an exception like a software interrupt, and cause an "exception" handling function to be called when the exception occurs. This is called a halt, and is controlled by the sane library routines **testhalt, sethalt, gethaltvector,** and **sethaltvector.**

Scanf

In order to allow the input of the new types and values, the syntax of floating point values acceptable to **scanf** has been changed to allow **NaNs** and **INF**:

```
<infinity>   ::= INF
<NaN>        ::= NaN (<Empty> | '(' <digits> ')')
<digits>     ::= <Empty> | ('0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9') <digits>
```

Both "NaN" and "INF" are case insensitive, and the NaN number is decimal.

New size specifiers (m and n) have been added to the format string to allow the input of types **comp** and **extended**. To summarize floating input:

Format Specifier	Type of argument
e	float
f	float
g	float
le	double
lf	double
lg	double
ne	extended
nf	extended
ng	extended
me	comp
mf	comp
mg	comp

Printf

Corresponding to the scanf changes, **INF** and **NaN** values are written as "NaN(d)", where d is a decimal number between 1 and 255, and "INF". Exponents in the e, E, g, or G formats may be up to four digits wide, and will always be printed as at least two digits (with a leading 0 added if necessary). See **C: A Reference Manual**, by Harbison and Steele for a precise specification of **printf** and **scanf**.

THE FLOATING POINT LIBRARY

The floating point library is grouped into four sections. The first section contains the numeric functions which do standard sorts of things with floating numbers (sin, cos, exp, etc.) The second section contains functions which return specific values, perform miscellaneous operations on floating point numbers, or obtain information about floating point numbers. The third section contains routines for interrogating and controlling the floating point environment, and the fourth section contains formatting and conversion functions. Remember that in C, all floating function arguments are forced to **extended.** Library functions whose operations or results are obvious are not documented beyond their calling sequences and/or result types. Arguments named **x** or **y** are taken to be type **extended**, those named **i** are **int,** and those named **n** are short.

All types and constants for the floating point library are defined in the file "sane.h", which you should **include** in any source file using the floating point library. The code for the library is in the file "sanelib.c", so your link control file should contain "sanelib" as an entry.

Floating Point Numeric Functions

Name	Declaration	Meaning
ANNUITY:	extended annuity(x, y)	(1 - (1+x)^(-y))/x. x = periodic rate. y = number of periods.
ATAN:	extended atan(x)	arctangent.
COMPOUND:	extended compound(x, y)	(1+x)^y. x = periodic rate. y = number of periods.
COS:	extended cos(x)	

Name	Declaration	Meaning
EXP:	extended exp(x)	exponential base-e.
EXP1:	extended exp1(x)	exponential base-e (x)-1.
EXP2:	extended exp2(x)	exponential base-2.
EXP21:	extended exp21(x)	exponential base-2(x)-1.
FABS:	extended fabs(x)	absolute value.
IPOWER:	extended ipower(x, n)	x^n.
LOG:	extended log(x)	log base-e.
LOG1:	extended log1(x)	log base-e (1+x).
LOG2:	extended log2(x)	log base-2.
LOG21:	extended log21(x)	log base-2(1+x).
LOGB:	extended logb(x)	binary exponent of normalized x.
POWER:	extended ipower(x, y)	x^y
RANDOMX:	extended randomx(seed) extended *seed;	gets random value; updates seed.
REMAINDER:	extended remainder(x, y, quo) short *quo;	return x % y; *quo is assigned the low order 7 bits of the quotient as a signed value, i.e. -127 <= *quo <= 127.
SIN:	extended sin(x)	
SQRT:	extended sqrt(x)	
TAN:	extended tan(x)	
SCALB:	extended scalb(n, x)	x * 2^n.

Miscellaneous Functions

Name	Declaration	Meaning
CLASSCOMP:	numclass classcomp(c) comp c;	abs(result) is class of c. result < 0 if c < 0.
CLASSDOUBLE:	numclass classdouble(d) double d;	abs(result) is class of d. result < 0 if d < 0.
CLASSEXTENDED:	numclass classextended(x);	abs(result) is class of x. result < 0 if x < 0.
CLASSFLOAT:	numclass classfloat(f) float f;	abs(result) is class of f . result < 0 if f < 0.
COPYSIGN	extended copysign(x, y)	returns y with sign of x.
INF	extended inf()	returns INF.
NAN	extended nan(v) unsigned char v;	returns NaN(v).
NEXTCOMP	extended nextcomp(x, y)	returns next value after x in the direction of y, comp precision.
NEXTDOUBLE	extended nextdouble(x, y)	returns next value after x in the direction of y , double precision.
NEXTEXTENDED	extended nextextended(x, y)	returns next value after x in the direction of y, extended precision.
NEXTFLOAT	extended nextfloat(x, y)	returns next value after x in the direction of y, float precision.
PI	extended pi()	returns value of pi.
RELATION	relop relation(x, y)	returns 0 if x > y, 1 if x<y, 2 if x=y, 3 if unordered.
RINT:	extended rint(x)	rounds x to integral value.
SIGNNUM	int signnum(x)	returns 0 if x >= +0., 1 if x <= -0.
TINT:	extended tint(x)	truncates x to integral value.

Environment Control

Name	Declaration	Meaning
GETENVIRONMENT	void getenvironment(e) environment *e;	*e = environment.
GETHALTVECTOR	haltvector gethaltvector()	return halt vector.
GETPRECISION	roundpre getprecision()	return rounding precision.
GETROUND	rounddir getround()	return rounding direction.
PROCENTRY	void procentry(e) environment *e;	*e = environment environment = IEEE default.

PROCEXIT	void procexit(e) environment e;	temp = exceptions;
		environment = e;
		signal exceptions in temp.

SETENVIRONMENT	void setenvironment(e) environment e;	environment = e.
SETEXCEPTION	void setexception(e,n) exception e;	set flags indicated in mask e if n != 0, reset if n=0. May cause halt.
SETHALT	void sethalt(e,n) exception e;	enable halts indicated in mask e if n != 0, disable if n=0.
SETHALTVECTOR	void sethaltvector(h) haltvector h;	Set halt vector to h.
SETPRECISION	void setprecision(p); roundpre p;	set rounding precision to p.
SETROUND	void setround(r); rounddir r;	set rounding direction to r.
TESTHALT	int testhalt(e) exception e;	return 1 if any halt indicated by mask e is enabled, 0 if not.
TESTEXCEPTION	int testexception(e) exception e;	return 1 if any flag indicated by mask e is enabled, 0 if not.

Formatting and Conversion

Formatting and conversion of floating point values are normally done using **printf** and **scanf**. There may be instances, however, where it is necessary for a program to do its own formatting and conversion. The Mac C floating point library provides the necessary low level routines for doing this. Using them requires a thorough understanding of the DECIMAL RECORD format as described in Inside Macintosh.

A floating point decimal number may be represented as an ASCII string, a binary value, or as a value encoded as a mixture of binary and ASCII values known as a DECIMAL RECORD, or **decimal** for short. The format of a **decimal** is defined in "sane.h". In the

SANE package, ASCII strings are always converted to decimal format, and vice versa. Only **decimal** format operands can be converted to and from the binary format. Conversion from binary to **decimal** is controlled by a format record called a decimal format or **decform**. Binary values may be converted to decimal records with **num2dec**, and **decimal** records to binary with **dec2num. decimal** records may be converted to and from string formats with **dec2str** and **str2dec**. These routines are in the file floatconv, so you must include it in your link control file if you use these.

Name	Declaration	Meaning
DEC2NUM	extended dec2num(d) decimal *d;	convert the decimal record at d to binary and return value.
NUM2DEC	void num2dec(f,x,d) decform *f; decimal *d;	Convert x to decimal record and store into *d.
DEC2STR	void dec2str(f,d,s) decform *f; decimal *d; char *s;	Convert number in decimal record *d into an ASCII string in *s.
STR2DEC	void str2dec(s,inx,d,valid) char *s;short *inx, *valid; decimal *d;	Converts ASCII string at s[*inx] into decimal record *d. *inx is updated to point past last character used from s. *valid is non-0 if a null byte terminated the string.

APPENDIX A:
DIFFERENCES BETWEEN MAC C AND STANDARD C

INTRODUCTION

Mac C differs in several ways from standard C as defined in **The C Programming Language**, by Brian W. Kernighan and Dennis M. Ritchie. These differences, explained in the following text, fall into three categories: extensions, interpretations, and restrictions.

EXTENSIONS

These extensions represent enhancements of the Kernighan and Ritchie C.

1. All characters of an identifier are significant.

2. All field names are local in scope to the structure in which they are declared. Mac C enforces strict member specification rules on the structure memory reference operators ". " and "->."

3. Inline assembly code may be bracketed by **#asm** and **#endasm** lines.

4 Type and field checking is carried out through all levels of indirection.

5. All basic types (**char, short, int, long**) may be unsigned.

6. Character literals may be from one to four characters in length.

7. The '#' character may appear in any column, so long as it is preceded only by white space.

8. The character pair "//" outside of strings causes the remainder of the line to be a comment.

9. Automatic (local) declarations may appear anywhere before they are are first used. They do not need to appear immediately after the "{."

10. The address of a procedure p() may be taken either with the C standard:

```
ptr = p;
```

or, alternatively:

```
ptr = &p;
```

11. The floating point implemented in Mac C 2.0 is based on the Apple IEEE/SANE Numerics package, and features 80 bit precision with the added data types **ext** and **comp.** Arguments and intermediate results are 80 bit values.

12. The C extensions: structure assignment, passing structures by value, and functions returning structures are implemented in Mac C.

13. The C extension of enumerated types is implemented. The Mac C implementation is based on the "pointer model" suggested in **C: A Reference Manual** by Harbison and Steele.

14. Bitfields may be specified as **unsigned char, unsigned short, unsigned int,** or **unsigned long.** The basic type identifies the "unit" size in which the bit field is to be imbedded. Address alignment will be done before the first bitfield of a group according to the basic type, and storage will be allocated for bitfields in multiples of the the basic type size. Bits are numbered from the MSB (0) to the LSB (n), which is the opposite of hardware numbering. Bit fields may not be initialized.

15. Type **void** is allowed.

INTERPRETATIONS

These interpretations represent implementation specifics.

1. **char** and **short** values are only converted to integers when necessary., e.g. , given:

```
char c1,c2,c3;
c1 = c2+c3;
```

c2 and c3 are NOT converted to **int** before the addition.

2. The programmer may select a 16- or 32-bit integer size. 32 bits is standard.

3. The >> operator shifts 0 into the high order bits.

4. Case sensitivity is not carried outside of the source file.

5. '\n' is interpreted as a return, since linefeed is not recognized as a newline character on the Macintosh.

6. Mac C uses the MDS Assembler to process its output, and the Assembler has certain reserved words which cannot be used as labels (Register names and instruction mnemonics). These words (such as SP, ST, SF, MOVE, etc.) cannot be used as global or static variable or function names in your C program. You can use them as local names, structure and field names, and typedefs.

7. There are some differences in identifier scope:

   a. The scope of structures is always from the point of definition to the end of the source file, even if the declaration is in a function.

   b. Local variables may not be redefined in a subordinate block, i.e.

```
p()
{ int i;
    { int i;
    }
 }
```

   is illegal.

   c. The scope of labels is logically the procedure in which they are declared, but this is not checked by Mac C.

An Important Note.

C allows a variable number of arguments to be passed to a function as a non-portable construct. Mac C adds a feature to the C language to specify that a function is to be called with a variable number of arguments. This serves as both an implementation and a documentation aid (it flags the use of a non-portable feature.) To specify a variable number of arguments, the last element in the function argument list must be "...", and external declarations of the function should simply include the three dots between the () in the declaration. **"printf"**, for example, is defined as:

```
int printf(format, args, ...)
```

and declared external as:

```
extern int printf(...);
```

APPENDIX B:
USING ASSEMBLY CODE WITH MAC C

INTRODUCTION

Since Mac C is, in fact, a translator from C to Assembler, using assembly code with a Mac C program is easy and convenient. There are two ways to use assembly code with Mac C: the first is inline, and the second is via linked files. Both methods are explained in this appendix. For information about the Macintosh Assembler and its use, consult the **Macintosh 68000 Development System Manual.**

Inline Assembly Code

The most convenient way to use assembly code is inline. Mac C recognizes the commands **#asm** and **#endasm**, as delimiting an assembly code routine, and passes any text between them to the intermediate output (assembly code) file on a line-by-line basis. The **#asm** and **#endasm** commands must appear on separate lines at the beginning and end of the inline assembly code as the following example illustrates:

```
swap(ptrA, ptrB)
    long *ptrA, *ptrB;
    {
    /*
        long temp;
        temp = *ptrA;
        *ptrA = *ptrB;
        *ptrB = temp;
    */
    #asm
        MOVE.L D0,A0        ; ptrA
        MOVE.L D1,A1        ; ptr B
        MOVE.L (A0),D2
        MOVE.L (A1),(A0)
        MOVE.L D2,(A1)
    #endasm
    };
```

When the compiler encounters such inline assembly procedures, it does not generate the normal procedure prologue and epilogue. This allows fast and compact routines to be imdedded on a procedure-by-procedure basis in a program.

To use the inline assembly feature effectively, both the Mac C procedure call conventions, and the way Mac C references global and local storage must be understood. See Chapter 2: Compiler Code Generation for more information.

Because the assembly code labels have not been declared to Mac C for typing and scope purposes, they must be declared as normal external variables in the C source file. This does not apply to routines such as **swap** in the above example, since the routine is actually defined as a C function.

Some assembly code labels are not referenced in the C source code file in which they are declared, but are referenced in another source file. In this case, an assembly **XDEF** directive must be provided.

Linking Assembly Files

Linking a Mac C program with assembly files is easy to do. Simply include the name of the assembly file in the Linker control file. Be sure that global variables are properly declared, and that you understand how Mac C calls functions and references variable names. Note that registers A5, A6, and A7 must be preserved. The first seven arguments to a function are passed in registers D0-D6, and function results are returned in register D0 for values and A0 for pointers.

APPENDIX C:
CALLING PASCAL PROCEDURES
FROM MAC C

INTRODUCTION

Combining C and Pascal procedures (for instance, library routines) may be very desirable. Mac C allows procedures generated by Apple Lisa Pascal to be called from Mac C by writing a short "glue" routine. For example, for the following Pascal procedure:

```
FUNCTION PascalFunc(boolParm: BOOLEAN; intParm: INTEGER;
        longParm: LongInt; structParm: SomeStruct; VAR varParm: INTEGER):
        INTEGER;
```

a glue routine could be written:

```
short CPascalFunct(boolParm, intParm, longParm, structParm, varParm)
    char boolParm;              /* BOOLEANs are bytes */
    short intParm;              /* INTEGERS are short (16 bits) */
    long longParm;
    char *structParm;           /* any pointer will do */
    short *varParm;             /* VARs are pointers */
    {
   #asm
    CLR         -(SP)           ; SPACE FOR 16-BIT RESULT
    MOVE.B      D0,-(SP)        ; boolParm
    MOVE        D1,-(SP)        ; intParm
    MOVE.L      D2,-(SP)        ; longParm
    MOVE.L      D3,-(SP)        ; structParm
    MOVE.L      D4,-(SP)        ; VAR varParm
    JSR PascalFunc
    MOVE (SP)+,D0               ; INTEGER RESULT
    #endasm
    };
```

Executing this routine enables the Pascal function to be referenced from a C program where desired. The example illustrates the basic characteristics of the Pascal calling conventions for Mac C, which include the following:

- Types BOOLEAN and CHAR are 8 bits.
- Type INTEGER is 16 bits.
- Types LONG INT, VAR references, and pointers are 32 bits.
- All parameters are passed on the stack.
- Space for the result must be allocated on the stack before the first parameter.
- The result is left on the stack.
- Pascal procedures preserve registers D3-D7 and A3-A7.

It is important to note that Pascal does not always pass the address of a structure when the structure is passed as an argument. In particular, if the total size of the structure is <= 32 bits, it passes the VALUE and not the address.

Pascal uses Bit 0 of a char value for Boolean values. Be sure the value you pass has this bit set for true.

If a function or procedure with more than seven arguments is called, the arguments must be obtained from the stack. This can be accomplished either by saving a pointer to the top of the stack on procedure entry, or by calculating the offset of the desired parameter from the current top of the stack. Saving the stack pointer is simpler and much less likely to cause bugs.

The following illustrates this calling convention:

SFPGetFile(where, prompt, fileFilter, numTypes, typeList, dlgHook, reply, dlgID, filterProc)

```
Point *where;
P_Str *prompt;
int (*fileFilter)();
short numTypes;
SFTypeList *typeList;
int (*dlgHook)();
SFReply *reply;
  short dlgID;
int (*fileFilter)();
{
#asm
  ARG8              EQU       4
  ARG9              EQU       8
  MOVE.L            SP,A1                   ; Save original SP
  MOVE.L            D0,A0                   ; dereference point since structure size
                                            ; is smaller than 33 bits, and PASCAL
                                            ; expects the value and not the address
  MOVE.L            (A0),-(SP)              ; where
  MOVE.L            D1,-(SP)                ; prompt
  MOVE.L            D2,-(SP)                ; fileFilter
  MOVE.W            D3,-(SP)                ; numTypes
  MOVE.L            D4,-(SP)                ; typeList
  MOVE.L            D5,-(SP)                ; dlgHook
  MOVE.L            D6,-(SP)                ; reply
  MOVE.W            ARG8+2(A1),-(SP)        ; dlgID
  MOVE.L            ARG9(A1),-(SP)          ; filterProc
  MOVE.W            #4,-(SP)                ; Select SFPGetFile
DC.W  $A9EA                     ; _Pack3
#endasm
};
```

APPENDIX D
ERROR MESSAGES

This appendix documents compiler error messages. Assembler and linker errors messages can be found in the **Macintosh 68000 Development System Manual**. Run-time error reporting is the responsibility of the user program or the Macintosh operating system.

The compiler error messages are listed alphabetically, with the actual message indicated in bold, and an explanation (unless it is obvious) underneath.

ADDRESS LOAD ON ILLEGAL NODE TYPE
This is an internal system error.

ALREADY DEFINED
An identifier was encountered which had been previously defined in a similar context.

BREAK OUT OF CONTEXT
A break was encountered outside of a switch statement or a loop.

CODE GEN ERROR
This is a system error.

CONTINUE OUT OF CONTEXT
A continue statement was encountered outside of a loop.

COULD NOT OPEN FILE <FILENAME>
The Compiler was unable to open the indicated file.

DISK FULL

DISK I/O ERROR

DISKETTE WRITE PROTECTED

DIRECT LOAD ON ILLEGAL NODE TYPE
This is an internal system error.

DIVIDE BY 0
An attempt was made to divide by zero in a literal expression.

DUPLICATE CASE CONSTANT
The same case constant was used twice in a switch statement.

DUPLICATE DEFAULT IN SWITCH
Two or more default cases exist in the switch list.

DUPLICATE LABEL
A label was declared twice.

DUPLICATE MACRO DEFINITION
A macro was defined twice.

EXPRESSION TOO COMPLEX
The number of operators in a given expression has exceeded the limit. To eliminate this problem, break the expression into smaller units, using temps if necessary.

EXTRA OPERAND
An expression with an extra operand was encountered.

FILE LOCKED
One of Mac C's output files cannot be opened.

FILE NAME TOO LONG
A file name longer than 252 characters was declared.

FILE OPEN ERROR

FUNCTION TYPE DOES NOT MATCH PREVIOUS DECL OR USE
A function that has been referenced or declared is later defined, and the definition type does not match the original type of the reference or declaration. The default for undefined functions is type int.

ILLEGAL ARGUMENT CONVERSION
An attempt was made to pass an illegal value to a procedure.

ILLEGAL ARGUMENT NAME
An argument had an illegal name.

ILLEGAL ARRRAY TYPE
An attempt was made to have an array of functions.

ILLEGAL BOOLEAN EXPRESSION
The compiler failed in an attempt to turn a relational expression into a value. This should never happen.

ILLEGAL CASE CONSTANT
A value that is not a constant has been used as part of a case label.

ILLEGAL CONSTANT EXPRESSION
A non-constant expression was encountered where a constant expression was required.

ILLEGAL C$$ TOKEN
This is a system error.

ILLEGAL DECLARATION

ILLEGAL FORMAL ARGUMENT TYPE
An attempt was made to pass an structure or an array as formal argument.

ILLEGAL FORMAL IN MACRO CALL
An illegal formal parameter was encountered in a macro call.

ILLEGAL FUNCTION CALL
The compiler encountered what appeared to be a function call, but the function name was illegal.

ILLEGAL FUNCTION DECLARATION
A function was encountered in an illegal context (e.g., in the argument list for a procedure).

ILLEGAL FUNCTION TYPE
A function has been declared with the type array or function.

ILLEGAL INDEX
Either an illegal value was used an an index (i.e., a pointer cannot be used as an index value), or an expression that was not a type pointer was followed by an index expression.

ILLEGAL INITIALIZATION
A user tried to initialized an automatic variable that was either an array or a structure.

ILLEGAL INC/DEC
A "++" or "--" construct was encountered in an illegal context.

ILLEGAL INT CONVERSION
An attempt was made to convert an argument to an illegal form. This attempt could either be implied by the conversion rules of an expression, or in explicit type casting.

ILLEGAL LABEL
A label was used with the wrong syntax.

ILLEGAL LEFT HAND SIDE
An expression used as the destination of an assignment did not have a legal assignment value, or an attempt was made to take the address (either by context or explicitly with the & operator) of an illegal value.

ILLEGAL LONG CONVERSION
An attempt was made to convert an argument to an illegal form. This attempt could either be implied by the conversion rules of an expression, or in explicit type casting.

ILLEGAL MACRO DEFINITION
A syntax error was made in a macro definition.

ILLEGAL MACRO NAME
A macro was defined with an illegal format name.

ILLEGAL MEM OP NODES
A system error was encountered when parsing an expression of the form <ident> <op> = <exp>.

ILLEGAL NAME FOLLOWS #
Either an illegal preprocessor directive was encountered, or a trap name under the #NoTraps option was improperly spelled.

ILLEGAL NUMBER
A number with an illegal format was encountered (e.g., the digit "8" or "9" in an octal number).

ILLEGAL OPTION
An illegal character or syntax was found after # options.

ILLEGAL OPERATOR OR TYPE ERROR
The system has encountered an error of an unknown type in an expression.

ILLEGAL POINTER OPERATOR OR COMBINATION
Pointers were used in an illegal context (e.g., attempting to multiply a pointer by a number).

ILLEGAL PREPROCESSOR LINE

ILLEGAL RIGHT HAND SIDE
The right hand side of an assignment was made in an illegal fashion.

ILLEGAL SIZE OF ARG
Sizeof operator was used but was not followed by a legal operand for sizeof.

ILLEGAL STATEMENT
An input statement could not be parsed by the Compiler into a legal statement.

ILLEGAL STRUCTURE ELEMENT
An illegal element, e.g., a function, was encountered in a structure body.

ILLEGAL STRUCTURE REF
Either a structure reference was attempted to an entity that was not a structure, or the reference was of the wrong type (e.g., a "." rather than a "-->" reference was used).

ILLEGAL SYMBOL FOLLOWS #
A # has been followed by an unknown identifier.

ILLEGAL SYMBOL NAME
The Compiler expected the name of a symbol but encountered something else.

INCLUDE ERROR
An unspecified problem was encountered when trying to include a file.

INCOMPATIBLE POINTER ASSIGNMENT
An attempt to assign or operate on a pointer of one type by a pointer of another type has been made. (Pointers should be type cast so they are of the same type.)

INCOMPATIBLE POINTERS
Two pointers of different types were used in an expression, or an attempt to assign or operate on a pointer of one type by the pointer of the second type was made, and failed. Pointers should be type cast so they are of the same type.

INCOMPATIBLE TYPES IN A:B
A statement of the form test? A:B was written in which the types of the first and second values were incompatible. This might happen if pointers of different types were used. Use the type cast operator to create pointers of the same type.

I/O MEMORY ERROR
This is a direct error from the Macintosh I/O system.

LINE TOO LONG

MACRO TOO LONG
A text required to store a macro was more than 500 characters in length.

MISSING ":"
In an expression of the form value:value, the ":" was missing.

MISSING "}"
The program compiled, but when the compilation completed, the block count was not equal to zero. This indicates the number of right and left braces did not match.

MISSING "c"
The word "missing" followed by character indicates a syntax error.

MISSING FORMAL IN MACRO CALL
Either the syntax of the call indicated that a formal argument should be present or the definition of a macro required a formal argument. The argument, however, was not passed in the call.

MISSING FORMAL IN MACRO DEFINITION OR CALL
Either the syntax of the call indicated that a formal argument should be present or the definition of a macro required a formal argument. The argument, however, was not passed in the call.

MISSING LABEL
A GOTO was encountered that was not followed by a label.

MISSING OPERAND
The expression parser expected an operand but did not encounter one. This error message is normally issued only after an expression has been scanned and an anticipated operate and has not been found.

MISSING PROCEDURE BODY
A procedure without a body was encountered. Typically, this error is caused by putting a semi-colon after the argument list in a procedure declaration.

MISSING STRUCTURE BODY
A structure declaration was encountered without a structure body being defined.

MISSING "while"
A do clause appeared that was lacking a while clause.

MISSING ")" IN MACRO CALL
A macro call was made and the number of left and right parentheses do not match.

NEGATIVE ARRAY SIZE
A negative value was encountered as an array size in a declaration.

NO CASES IN SWITCH
A switch without any case labels was encountered.

NUMBER EXPECTED

NUMBER TOO LONG
A number longer than 252 characters was encountered.

OUT OF MEMORY
This message should only be encountered if an attempt is made to run Mac C with MacsBug or TermBug installed.

OUT OF REGISTERS
The Compiler attempted to evaluate an expression but could not find enough free registers. While this is unlikely to occur, the expression in question may be simplified, using temps if necessary.

OUT OF SPACE
This message indicates one of the Compiler's internal tables is full. This condition is normally reported by a more specific message, i.e., Symbol Table Full.

POINTER REQUIRED
A pointer was required where a non-pointer value was used.

POSSIBLE TYPE CAST ERROR
A pointer is cast to a short; that is, a 32-bit pointer is cast to a 16-bit value.

PUSH FZN REG
This is a system error. If it occurs, try to simplify the expression using temps.

STACK UNDERFLOW
This is a system error.

STRING TOO LONG
Either a single string was longer than 252 characters, or a collection of strings in an expression exceeded the available string storage. To eliminate this problem, break up the expression and store one string into a temp in another expression.

SYSTEM REGISTER ERROR (INTERNAL)
This is a system error.

SYMBOL TABLE FULL
This error indicates that one of the symbol tables has overflowed. The sizes of the internal symbol tables for the compilation are printed at the end of compilation, so it is possible to tell which table has overflowed by examining these values. Each table size can be set individually. (See Chapter 1 for compiler option details.)

SYMBOL TOO LONG
The symbol encountered was longer than 252 characters.

TOO MANY ARGUMENTS
A procedure was called with more than 50 arguments.

TOO MANY CASES
More than 100 cases in a switch statement were encountered. To eliminate this problem, break the switch statement up, nesting a second switch statement into the default of the first.

TOO MANY FORMALS IN MACRO
A macro was encountered with more than nine formal arguments.

TOO MANY LEVELS OF MACRO NESTING
Macros were nested more than 100 levels.

TOO MANY NESTED FILES
Includes were nested more than eight levels.

TOO MANY PROCEDURE ARGUMENTS
More than 50 arguments were found in a procedure declaration.

TYPE MISMATCH
An expression was evaluated that had two types in it that did not match.

TYPE SIZE LOOP (SYSTEM ERROR)
This is a system error.

TYPE STORAGE FULL
The type storage table has overflowed. The size of the type storage table for the compilation is printed at the end of the compilation. (See Chapter 1 for setting compiler options.)

TYPED ARG NOT DECLARED
An identifier was listed in the type list of a procedure declaration but the identifier did not appear in the argument list.

UNDEFINED VARIABLE
An identifier that had not been previously declared was encountered in an expression.

UNKNOWN I/O ERROR
This is a direct error from the Macintosh I/O system.

UNMATCHED ")"
A right parenthesis has been encountered without a previous left parenthesis.

VOLUME LOCKED
This is a direct error from the Macintosh I/O system.

WRONG NUMBER OF ARGS IN MAC TRAP CALL
A call was made to the Macintosh system trap with the wrong number of arguments.

WRONG OR MISSING FIELD
Either a structure was referenced with a field that does not belong to that structure, or a structure was referenced without a field where a field is required.

#N OPTION ILLEGAL AFTER TRAPS LOADED
The #N option was used too late in the source file. This option must be encountered before the first structure or identifier declaration.

APPENDIX E:
THE MACINTOSH TRAPS

Function Type	Trap Name and Argument	Trap I.D.
short	AddDrive(D0, A0), D0 = Result (short)	A04E
	AddPt(Point, Long)	A87E
	AddReference(Long, short, Long)	A9AC
	AddResMenu(Long, Long)	A94D
	AddResource(Long, Long, short, Long)	A9AB
short	Alert(short, Long)	A985
short	AngleFromSlope(Long)	A8C4
	AppendMenu(Long, Long)	A933
	BackColor(Long)	A863
	BackPat(Long)	A87C
	BeginUpdate(Long)	A922
int	BitAnd(Long, Long)	A858
	BitClr(Long, Long)	A85F
int	BitNot(Long)	A85A
int	BitOr(Long, Long)	A85B
	BitSet(Long, Long)	A85E
int	BitShift(Long, short)	A85C
char	BitTst(Long, Long)	A85D
int	BitXor(Long, Long)	A859
short	BlockMove(A0, A1, D0), D0 = Result (short)	A02E
	BringToFront(Long)	A920
char	Button()	A974
	CalcMenuSize(Long)	A948
	CalcVis(Long)	A909
	CalVisBehind(Long, Long)	A90A
short	CautionAlert(short, Long)	A988
	Chain(A0)	A9F3
	ChangedResData(Long)	A9AA
short	CharWidth(short)	A88D
	CheckItem(Long, short, char)	A945
char	CheckUpdate(Long)	A911
	ClearMenuBar()	A934
	ClipAbove(Long)	A90B
	ClipRect(Long)	A87B
	CloseDeskAcc(short)	A9B7
	CloseDialog(Long)	A982
	ClosePicture()	A8F4
	ClosePoly()	A8CC
	ClosePort(Long)	A87D
	CloseResFile(short)	A99A
	CloseRgn(Long)	A8DB
	CloseWindow(Long)	A92D

Function Type	Trap Name and Argument	Trap I.D.
	ColorBit(short)	A864
int	CompactMem(D0), D0 = Result (int)	A04C
	CopyBits(Long, Long, Long, Long, short, Long)	A8EC
	CopyRgn(Long, Long)	A8DC
	CouldAlert(short)	A989
	CouldDialog(short)	A979
short	CountMItems(Long)	A950
short	CountResources(Long)	A99C
short	CountTypes()	A99E
	CreateResFile(Long)	A9B1
short	CurResFile()	A994
int	Date2Secs(A0), D0 = Result (int)	A9C7
int	Delay(A0), D0 = Result (int)	A03B
	DeleteMenu(short)	A936
short	Dequeue(A0, A1), D0 = Result (short)	A96E
	DetachResource(Long)	A992
char	DialogSelect(Long, Long, Long)	A980
	DiffRgn(Long, Long, Long)	A8E6
	DisableItem(Long, short)	A93A
short	DisposHandle(A0), D0 = Result (short)	A023
	DisposeControl(Long)	A955
	DisposeDialog(Long)	A983
	DisposeMenu(Long)	A932
	DisposeRgn(Long)	A8D9
	DisposeWindow(Long)	A914
short	DisposPtr(A0), D0 = Result (short)	A01F
	DragControl(Long, Point, Long, Long, short)	A967
int	DragGrayRgn(Long, Point, Long, Long, short, Long)	A905
int	DragTheRgn(Long, Point, Long, Long, short, Long)	A926
	DragWindow(Long, Point, Long)	A925
	DrawChar(short)	A883
	DrawControls(Long)	A969
	DrawDialog(Long)	A981
	DrawGrowIcon(Long)	A904
	DrawMenuBar()	A937
	DrawNew(Long, char)	A90F
	DrawPicture(Long, Long)	A8F6
	DrawString(Long)	A884
	DrawText(Long, short, short)	A885
short	EmptyHandle(A0), D0 = Result (short)	A02B
char	EmptyRect(Long)	A8AE
char	EmptyRgn(Long)	A8E2
	EnableItem(Long, short)	A939
	EndUpdate(Long)	A923
char *	Enqueue(A0, A1), A0 = Result (char *)	A96F
char	EqualPt(Point, Point)	A881
char	EqualRect(Long, Long)	A8A6
char	EqualRgn(Long, Long)	A8E3
short	EqualString(A0, A1, D0), D0 = Result (short)	A03C
	EraseArc(Long, short, short)	A8C0
	EraseOval(Long)	A8B9

Function Type	Trap Name and Argument	Trap I.D.
	ErasePoly(Long)	A8C8
	EraseRect(Long)	A8A3
	EraseRgn(Long)	A8D4
	EraseRoundRect(Long, short, short)	A8B2
	ErrorSound(Long)	A98C
char	EventAvail(short, Long)	A971
	ExitToShell()	A9F4
	FillArc(Long, short, short, Long)	A8C2
	FillOval(Long, Long)	A8BB
	FillPoly(Long, Long)	A8CA
	FillRect(Long, Long)	A8A5
	FillRgn(Long, Long)	A8D6
	FillRoundRect(Long, short, short, Long)	A8B4
short	FindControl(Point, Long, Long)	A96C
short	FindWindow(Point, Long)	A92C
int	FixMul(Long, Long)	A868
int	FixRatio(short, short)	A869
short	FixRound(Long)	A86C
	FlashMenuBar(short)	A94C
	FlushEvents(D0)	A032
	ForeColor(Long)	A862
	FrameArc(Long, short, short)	A8BE
	FrameOval(Long)	A8B7
	FramePoly(Long)	A8C6
	FrameRect(Long)	A8A1
	FrameRgn(Long)	A8D2
	FrameRoundRect(Long, short, short)	A8B0
	FreeAlert(short)	A98A
	FreeDialog(short)	A97A
int	FreeMem(), D0 = Result (int)	A01C
char *	FrontWindow()	A924
	GetAppParms(Long, Long, Long)	A9F5
	GetClip(Long)	A87A
int	GetCRefCon(Long)	A95A
	GetCTitle(Long, Long)	A95E
char *	GetCtlAction(Long)	A96A
short	GetCtlMax(Long)	A962
short	GetCtlMin(Long)	A961
short	GetCtlValue(Long)	A960
char **	GetCursor(short)	A9B9
	GetDItem(Long, short, Long, Long, Long)	A98D
	GetFNum(Long, Long)	A900
	GetFontInfo(Long)	A88B
	GetFontName(short, Long)	A8FF
int	GetHandleSize(A0), D0 = Result (int)	A025
char *	GetIcon(short)	A9BB
char *	GetIndResource(Long, short)	A99D
	GetIndType(Long, short)	A99F
	GetItem(Long, short, Long)	A946
	GetItemIcon(Long, short, Long)	A93F
	GetItemMark(Long, short, Long)	A943

Function Type	Trap Name and Argument	Trap I.D.
	GetItemStyle(Long, short, Long)	A941
	GetIText(Long, Long)	A990
	GetKeys(Long)	A976
char *	GetMenu(short)	A9BF
char *	GetMenuBar()	A93B
char *	GetMHandle(short)	A949
	GetMouse(Long)	A972
char *	GetNamedResource(Long, Long)	A9A1
char *	GetNewControl(short, Long)	A9BE
char *	GetNewDialog(short, Long, Long)	A97C
char *	GetNewMBar(short)	A9C0
char *	GetNewWindow(short, Long, Long)	A9BD
char	GetNextEvent(short, Long)	A970
short	GetOSEvent(D0, A0), D0 = Result (short)	A031
char *	GetPattern(short)	A9B8
	GetPen(Long)	A89A
	GetPenState(Long)	A898
char *	GetPicture(short)	A9BC
char	GetPixel(short, short)	A865
	GetPort(Long)	A874
int	GetPtrSize(A0), D0 = Result (int)	A021
short	GetResAttrs(Long)	A9A6
short	GetResFileAttrs(short)	A9F6
	GetResInfo(Long, Long, Long, Long)	A9A8
char *	GetResource(Long, short)	A9A0
int	GetScrap(Long, Long, Long)	A9FD
char *	GetString(short)	A9BA
char *	GetTrapAddress(D0), A0 = Result (char *)	A146
char *	GetWindowPic(Long)	A92F
	GetWMgrPort(Long)	A910
int	GetWRefCon(Long)	A917
	GetWTitle(Long, Long)	A919
char *	GetZone(), A0 = Result (char *)	A11A
	GlobalToLocal(Long)	A871
	GrafDevice(short)	A872
int	GrowWindow(Long, Point, Long)	A92B
short	HandAndHand(A0, A1), D0 = Result (Short)	A934
char *	HandleZone(A0), A0 = Result (char*)	A126
char *	HandToHand(A0), A0 = Result (char *)	A9E1
	HideControl(Long)	A958
	HideCursor()	A852
	HidePen()	A896
	HideWindow(Long)	A916
	HiliteControl(Long, short)	A95D
	HiliteMenu(short)	A938
	HiliteWindow(Long, char)	A91C
short	HiWord(Long)	A86A
char *	HLock(A0), A0 = Result (char *)	A029
short	HNoPurge(A0), D0 = Result (short)	A04A
short	HomeResFile(Long)	A9A4
short	HPurge(A0), D0 = Result (short)	A049

Function Type	Trap Name and Argument	Trap I.D.
short	HUnlock(A0), D0 = Result (short)	A02A
char *	InfoScrap()	A9F9
short	InitApplZone(), D0 = Result (short)	A02C
	InitCursor()	A850
	InitDialogs(Long)	A97B
	InitFonts()	A8FE
	InitGraf(Long)	A86E
	InitMenus()	A930
	InitAllPacks()	A9E6
	InitPack(short)	A9E5
	InitPort(Long)	A86D
	InitQueue()	A016
short	InitResources()	A995
short	InitUtil(), D0 = Result (short)	A03F
	InitWindows()	A912
short	InitZone(A0), D0 = Result (short)	A019
	InsertMenu(Long, short)	A935
	InsertResMenu(Long, Long, short)	A951
	InsetRect(Long, short, short)	A8A9
	InsetRgn(Long, short, short)	A8E1
short	InstallDriver(A0, D0), D0 = Result (short)	A03D
	InvalRect(Long)	A928
	InvalRgn(Long)	A927
	InvertArc(Long, short, short)	A8C1
	InvertOval(Long)	A8BA
	InvertPoly(Long)	A8C9
	InvertRect(Long)	A8A4
	InvertRgn(Long)	A8D5
	InvertRoundRect(Long, short, short)	A8B3
char	IsDialogEvent(Long)	A97F
	KillControls(Long)	A956
	KillPicture(Long)	A8F5
	KillPoly(Long)	A8CD
	Launch(A0)	A9F2
	Line(short, short)	A892
	LineTo(short, short)	A891
	LoadResource(Long)	A9A2
int	LoadScrap()	A9FB
	LoadSeg(short)	A9F0
	LocalToGlobal(Long)	A870
	LongMul(Long, Long, Long)	A867
short	LoWord(Long)	A86B
	MapPoly(Long, Long, Long)	A8FC
	MapPt(Long, Long, Long)	A8F9
	MapRect(Long, Long, Long)	A8FA
	MapRgn(Long, Long, Long)	A8FB
int	MaxMem(), D0 = Result (int)	A01D
char *	MemLeft(), A0 = Result (char *)	A11D
int	MenuKey(short)	A93E
int	MenuSelect(Point)	A93D
	ModalDialog(Long, Long)	A991

Function Type	Trap Name and Argument	Trap I.D.
	MoreMasters()	A036
	Move(short, short)	A894
	MoveControl(Long, short, short)	A959
	MovePortTo(short, short)	A877
	MoveTo(short, short)	A893
	MoveWindow(Long, short, short, char)	A91B
int	Munger(Long, Long, Long, Long, Long, Long)	A9E0
char *	NewControl(Long, Long, Long, char, short, short, short, short, Long)	A954
char *	NewDialog(Long, Long, Long, char, short, Long, char, Long, Long)	A97D
char *	NewHandle(D0), A0 = Result (char *)	A122
char *	NewMenu(short, Long)	A931
char *	NewPtr(D0), A0 = Result (char *)	A11E
char *	NewRgn()	A8D8
char *	NewString(Long)	A906
char *	NewWindow(Long, Long, Long, char, short, Long, char, Long)	A913
short	NoteAlert(short, Long)	A987
	ObscureCursor()	A856
	OffsetPoly(Long, short, short)	A8CE
	OffsetRect(Long, short, short)	A8A8
	OffsetRgn(Long, short, short)	A8E0
short	OpenDeskAcc(Long)	A9B6
char *	OpenPicture(Long)	A8F3
char *	OpenPoly()	A8CB
	OpenPort(Long)	A86F
short	OpenResFile(Long)	A997
	OpenRgn()	A8DA
short	OSEventAvail(D0, A0), D0 = Result (short)	A030
	Pack2()	A9E9
	Pack3()	A9EA
	Pack4()	A9EB
	Pack5()	A9EC
	Pack6()	A9ED
	Pack7()	A9EE
	PackBits(Long, Long, short)	A8CF
	PaintArc(Long, short, short)	A8BF
	PaintBehind(Long, Long)	A90D
	PaintOne(Long, Long)	A90C
	PaintOval(Long)	A8B8
	PaintPoly(Long)	A8C7
	PaintRect(Long)	A8A2
	PaintRgn(Long)	A8D3
	PaintRoundRect(Long, short, short)	A8B1
	ParamText(Long, Long, Long, Long)	A98B
	PenMode(short)	A89C
	PenNormal()	A89E
	PenPat(Long)	A89D
	PenSize(short, short)	A89B
	PicComment(short, short, Long)	A8F2

Function Type	Trap Name and Argument	Trap I.D.
int	PinRect(Long, Point)	A94E
	PlotIcon(Long, Long)	A94B
	PortSize(short, short)	A876
short	PostEvent(A0, D0), D0 = Result (short)	A02F
	Pt2Rect(Point, Point, Long)	A8AC
char	PtInRect(Point, Long)	A8AD
char	PtInRegn(Point, Long)	A8E8
short	PtrAndHand(A0, A1, D0), D0 = Result (short)	A9EF
char *	PtrToHand(A0, D0), A0 = Result (char *)	A9E3
short	PtrToXHand(A0, A1, D0), D0 = Result (short)	A9E2
char *	PtrZone(A0), A0 = Result (char *)	A148
	PtToAngle(Long, Point, Long)	A8C3
char *	PurgeMem(D0), A0 = Result (char *)	A14D
int	PutScrap(Long, Long, Long)	A9FE
short	Random()	A861
	rDrvrInstall()	A04F
short	ReadDateTime(A0), D0 = Result (short)	A039
char	RealFont(short, short)	A902
short	ReallocHandle(A0, D0), D0 = Result (short)	A027
char *	RecoverHandle(A0), A0 = Result (char *)	A128
char	RectInRgn(Long, Long)	A8E9
	RectRgn(Long, Long)	A8DF
	ReleaseResource(Long)	A9A3
short	RemoveDriver(D0), D0 = Result (short)	A03E
short	ResError()	A9AF
char *	ResrvMem(D0), A0 = Result (char *)	A140
	RmveReference(Long)	A9AE
	RmveResource(Long)	A9AD
	RsrcZoneInit()	A996
	SaveOld(Long)	A90E
	ScalePt(Long, Long, Long)	A8F8
	ScrollRect(Long, short, short, Long)	A8EF
short	Secs2Date(D0, A0), D0 = Result (short)	A9C8
char	SectRect(Long, Long, Long)	A8AA
	SectRgn(Long, Long, Long)	A8E4
	SelectWindow(Long)	A91F
	SetIText(Long, short, short, short)	A97E
	SendBehind(Long, Long)	A921
short	SetApplBase(A0), D0 = Result (short)	A857
short	SetApplLimit(A0), D0 = Result (short)	A02D
	SetClip(Long)	A879
	SetCRefCon(Long, Long)	A95B
	SetCTitle(Long, Long)	A95F
	SetCtlAction(Long, Long)	A96B
	SetCtlMax(Long, short)	A965
	SetCtlMin(Long, short)	A964
	SetCtlValue(Long, short)	A963
	SetCursor(Long)	A851
short	SetDateTime(D0), D0 = Result (short)	A03A
	SetDItem(Long, short, short, Long, Long)	A98E
	SetEmptyRgn(Long)	A8DD

Function Type	Trap Name and Argument	Trap I.D.
	SetFontLock(char)	A903
short	SetGrowZone(A0), D0 = Result (short)	A04B
short	SetHandleSize(A0, D0), D0 = Result (short)	A024
	SetItem(Long, short, Long)	A947
	SetItemIcon(Long, short, short)	A940
	SetItemMark(Long, short, short)	A944
	SetItemStyle(Long, short, short)	A942
	SetIText(Long, Long)	A98F
	SetMenuBar(Long)	A93C
	SetMenuFlash(Long, short)	A94A
	SetOrigin(short, short)	A878
	SetPenState(Long)	A899
	SetPort(Long)	A873
	SetPortBits(Long)	A875
	SetPt(Long, short, short)	A880
short	SetPtrSize(A0, D0), D0 = Result (short)	A020
	SetRect(Long, short, short, short, short)	A8A7
	SetRectRgn(Long, short, short, short, short)	A8DE
	SetResAttrs(Long, short)	A9A7
	SetResFileAttrs(short, short)	A9F7
	SetResInfo(Long, short, Long)	A9A9
	SetResLoad(char)	A99B
	SetResPurge(char)	A993
	SetStdProcs(Long)	A8EA
	SetString(Long, Long)	A907
	SetTrapAddress(D0, A0)	A047
	SetWindowPic(Long, Long)	A92E
	SetWRefCon(Long, Long)	A918
	SetWTitle(Long, Long)	A91A
short	SetZone(A0), D0 = Result (short)	A01B
	ShieldCursor(Long, short, short)	A855
	ShowControl(Long)	A957
	ShowCursor()	A853
	ShowHide(Long, char)	A908
	ShowPen()	A897
	ShowWindow(Long)	A915
	SizeControl(Long, short, short)	A95C
Int	SizeResource(Long)	A9A5
	SizeWindow(Long, short, short, char)	A91D
int	SlopeFromAngle(short)	A8BC
	SpaceExtra(short)	A88E
	StdArc(Long, Long, short, short)	A8BD
	StdBits(Long, Long, Long, short, Long)	A8EB
	StdComment(short, short, Long)	A8F1
	StdGetPic(Long, short)	A8EE
	StdLine(Point)	A890
	StdOval(Long, Long)	A8B6
	StdPoly(Long, Long)	A8C5
	StdPutPic(Long, short)	A8F0
	StdRect(Long, Long)	A8A0
	StdRgn(Long, Long)	A8D1

Function Type	Trap Name and Argument	Trap I.D.
	StdRRect(Long, Long, short, short)	A8AF
short	StdTxMeas(short, Long, Long, Long, Long)	A8ED
	StdText(short, Long, Point, Point)	A882
char	StillDown()	A973
short	StopAlert(short, Long)	A986
short	StringWidth(Long)	A88C
	StuffHex(Long, Long)	A866
	SubPt(Point, Long)	A87F
char *	SwapFont(Long)	A901
	SysBeep(short)	A9C8
	SysError(D0)	A9C9
	SystemClick(Long, Long)	A9B3
char	SystemEdit(short)	A9C2
char	SystemEvent(Long)	A9B2
	SystemMenu(Long)	A9B5
	SystemTask()	A9B4
	TEActivate(Long)	A9D8
	TECalText(Long)	A9D0
	TEClick(Point, char, Long)	A9D4
	TECopy(Long)	A9D5
	TECut(Long)	A9D6
	TEDeactivate(Long)	A9D9
	TEDelete(Long)	A9D7
	TEDispose(Long)	A9CD
char *	TEGetText(Long)	A9CB
	TEIdle(Long)	A9DA
	TEInit()	A9CC
	TEInsert(Long, Long, Long)	A9DE
	TEKey(short, Long)	A9DC
char *	TENew(Long, Long)	A9D2
	TEPaste(Long)	A9DB
	TEScroll(short, short, Long)	A9DD
	TESetJust(short, Long)	A9DF
	TESetSelect(Long, Long, Long)	A9D1
	TESetText(Long, Long, Long)	A9CF
short	TestControl(Long, Point)	A966
	TEUpdate(Long, Long)	A9D3
	TextBox(Long, Long, Long, short)	A9CE
	TextFace(short)	A888
	TextFont(short)	A887
	TextMode(short)	A889
	TextSize(short)	A88A
short	TextWidth(Long, short, short)	A886
int	TickCount()	A975
short	TrackControl(Long, Point, Long)	A968
char	TrackGoAway(Long, Point)	A91E
	UnionRect(Long, Long, Long)	A8AB
	UnionRgn(Long, Long, Long)	A8E5
short	UniqueID(Long)	A9C1
int	UnloadScrap()	A9FA
	UnloadSeg(Long)	A9F1

Function Type	Trap Name and Argument	Trap I.D.
	UnPackBits(Long, Long, short)	A8D0
	UpdateResFile(short)	A999
char *	UprString(A0, D0), A0 = Result (char *)	A854
	UseResFile(short)	A998
	ValidRect(Long)	A92A
	ValidRgn(Long)	A929
short	VInstall(A0), D0 = Result (short)	A033
short	VRemove(A0), D0 = Result (short)	A034
char	WaitMouseUp()	A977
short	WriteParam(A0, D0), D0 = Result (short)	A038
	WriteResource(Long)	A9B0
	XOrRgn(Long, Long, Long)	A8E7
int	ZeroScrap()	A9FC

APPENDIX F:
SAMPLE PROGRAMS

INTRODUCTION

This appendix lists the sources for two test programs. Testlib.c and MacDemo.c. Testlib.c calls and tests the library functions in various ways. It is described in Chapter 3: The Mac C Run-Time Environment. MacDemo.c illustrates various Macintosh features.

Both these programs are referenced in the warranty. Consulair Corp warrants that Mac C will compile the programs listed in this appendix.

```
/*************************************************************/
/*                                                           */
/*                                                           */
/*                        testlib.c                          */
/*                                                           */
/*          Mac C Library Test Program.                      */
/*                                                           */
/*       Copyright (C) 1984 Consulair Corporation.           */
/*              All Rights Reserved                          */
/*                                                           */
/*                                                           */
/*                                                           */
/*************************************************************/


/* declarations */

 #include "stdio.h"

 #define CR '\r'

/* Code */

 #define testlibC(routine) testcf(routine, "routine"); //

 char pause(str)
  {
   char c;
   if (str)
   {
    puts("\r\rTest \"");
    puts(str);
    puts("\" ? ");
   };
   if ((c=getchar()) == '.')
   {
    unlink("testLibFile");
    exit(0);
   };
   if (str) printf("%s\r", (c == '\r')? "Yes":"No");
```

```
 else printf("\r");
 return(c);
}

testcf(routine, routinename)
 char (*routine)();
 char *routinename;
{
 char c;
 unsigned short i, j;

 if (pause(routinename) != '\r') return;
 printf("\r      ");
 for (i = 0; i < 16; i+=1) printf("  %2x", i);
  printf("\r");
  for (i = 0; i < 16; i+=1)
                                {
                                printf("\r  %2x:  ", i);
                                for (j = 0; j < 16; j+=1)
                                  printf("  %2x", (*routine)(i*16+j));
                                }
 if (pause(0) != '\r') return;
 printf("\r\r      ");
 for (i = 0; i < 16; i+=1) printf("%4x", i);
  printf("\r");
  for (i = 0; i < 16; i+=1)
                                {
                                printf("\r %2x:  ", i);
                                for (j = 0; j < 16; j+=1)
                                 {
                                 c = (*routine)(i*16+j);
                                 printf("   %c", ((c < ' ')? ((c == 0)? '.':'*'): (c > 0x7f)? '*':c));
                                 }
                                }
 printf("\r");
}

testStr(routine, routinename, pstr1, pstr2, count)
 char *(*routine)();
 char *routinename;
 char *pstr1, *pstr2;
 int count;
{
 char *result;
 if (pause(routinename) != '\r') return(0);
 printCount(count);
 printStr(pstr1);
 printStr(pstr2);
 if (routine == 0) return (1);
 result = (*routine)(pstr1, pstr2, count);
 printf("\r Rslt: %s", result);
}

printCount(count)
 int count;
 {
```

```
 if (count > 0) printf("\r Count: %d", count);
 };

printStr(str)
 char *str;
 {
 if (str) printf("\r Str: %s", str);
 };

main()
 {
 int c;
 char line[MAXLINE]
 char c1;
 short s;
 int i, x;
 FILE file;

 puts("This is the Test Program\rIt will echo characters: ");
 while ((c = getchar()) != '\r') putchar(c);
 putchar('\r');



  testlibC(isupper) char isupper(c) -- return non-zero if 'A' <= c <= 'Z'
  testlibC(islower) char islower(c) -- return non-zero if 'a' <= c <= 'z'
  testlibC(isalpha) char isalpha(c) -- return non-zero if 'a' <= c <= 'z' or 'A' <= c <= 'Z'
  testlibC(isdigit) char isdigit(c) -- return non-zero if '0' <= c <= '9'
  testlibC(isspace) char isspace(c) -- return non-zero if c = SPACE, TAB, or NL (LF)
  testlibC(toupper) char toupper(c) -- return c or upper case value of c
  testlibC(tolower)  char tolower(c) -- return c or upper case value of c


  // char *index(s, c) -- return 0 or pointer to first occurrence of c in s
                        // int Index(s, c) -- return -1 or index of first occurrence of c in s
                        // char *rindex(s, c) -- return 0 or pointer to last occurrence of c in s

                        // int RIndex(s, c) -- return -1 or index of last occurrence of c in s
                        {
                        char *str;
                        str = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyZ0123456789";
                        if (testStr(0, "index", str, 0, 0))
                         {
                         printf("\r base addr = %x", str);
                         printf("\r  A = %x, a = %x, 9 = %x, . = %x",
                           index(str, 'A'),
                           index(str, 'a'),
                           index(str, '9'),
                           index(str, '.'));
                         }

                        if (testStr(0, "Index", str, 0, 0))
                         {
                         printf("\r  A = %d,  a = %d, 9 = %d, . = %d",
                           Index(str, 'A'),
                           Index(str, 'a'),
```

```
                    Index(str, '9'),
                    Index(str, '.'));
                  }

                  if (testStr(0, "rindex", str, 0, 0))
                   {
                   printf("\r base addr = %x", str);
                   printf("\r  Z = %x, a = %x, 9 = %x, . = %x",
                     rindex(str, 'Z'),
                     rindex(str, 'a'),
                     rindex(str, '9'),
                     rindex(str, '.'));
                   }

                  if (testStr(0, "Rindex", str, 0, 0))
                   {
                   printf("\r  Z = %d,  a = %d, 9 = %d, . = %d",
                     RIndex(str, 'Z'),
                     RIndex(str, 'a'),
                     RIndex(str, '9'),
                     RIndex(str, '.'));
                   }
                  }

// char *strsave(s) -- returns address of a copy of s (uses malloc())
                  {
                  char *str;
                  str = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";
                  testStr(strsave, "strsave", str, 0, 0);
                  }

// char *strcat(s1, s2)    -- appends s2 onto s1
                  {
                  char *str1, *str2;
                  str1 = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
                  str2 = "abcdefghijklmnopqrstuvwxyz0123456789";
                  str1[26] = 0;
                  testStr(strcat, "strcat", str1, str2, 0);
                  str1[26] = 0;
                  testStr(strncat, "strncat", str1, str2, 26);
                  }


// char *strcmp(s1, s2) -- compares s1 to s2.

                  // returns 0 if equal, -1 if s1 < s2, 1 if s1 > s2.
                  // char *strncmp(s1, s2, n) -- like strcmp, but compares up to n characters
                  {
                  char *str1, *str2;
                  str1 = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
                  str2 = "ABCDEFGHIJklmnopqrstuvwxyz";
                  if (testStr(0, "strcmp", str1, str2, 0))
                   {
                   printf("\r result = %d\r", strcmp(str1, str2));
                   printStr(str1); printStr(str1);
                     printf("\r result = %d\r", strcmp(str1, str1));
```

```
                    printStr(str2); printStr(str1);
                     printf("\r result = %d\r", strcmp(str2, str1));
                    }

                   if (testStr(0, "strncmp", str1, str2, 1))
                    {
                    printf("\r result = %d\r", strncmp(str1, str2, 1));
                    printCount(10); printStr(str1); printStr(str2);
                     printf("\r result = %d\r", strncmp(str1, str2, 10));
                    printCount(11); printStr(str1); printStr(str2);
                     printf("\r result = %d\r", strcmp(str1, str2, 11));
                    printCount(1100); printStr(str1); printStr(str2);
                     printf("\r result = %d\r", strcmp(str1, str2, 1100));
                    }
                   }

// char *strcpy(s1, s2) -- copy str2 to str1.

                   {
                   // char *strncpy(s1, s2, n) -- copies n characters of s2 to s1.
                   // if length s2 < n, then s1 will  be null padded.
                   char *str1, *str2;
                   str1 = calloc(100, 1);
                   str2 = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
                   testStr(strcpy, "strcpy", str1, str2, 0);

                   free(str1);
                   str1 = calloc(100, 1);
                   if (testStr(strncpy, "strncpy", str1, str2, 1))

                    {
                    free(str1);
                    str1 = calloc(100, 1);
                    printf("\r Count = 0"); printStr(str1); printStr(str2);
                     printf("\r result = %s\r", strncpy(str1, str2, 0));

                    free(str1);
                    str1 = calloc(100, 1);
                    printCount(1); printStr(str1); printStr(str2);
                     printf("\r result = %s\r", strncpy(str1, str2, 1));

                    free(str1);
                    str1 = calloc(100, 1);
                    printCount(10); printStr(str1); printStr(str2);
                     printf("\r result = %s", strncpy(str1, str2, 10));
                    };


                   free(str1);
                   }

// char *strlen(s) -- returns length of s
                   {
                   char *str;
                   str = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
                   if (testStr(0, "strlen", str, 0, 0))
```

```
                        {
                        printf("\r result = %d\r", strlen(str));
                        printf("\r strlen of EMPTY string = %d", strlen(""));
                        printf("\r strlen of NULL string = %d", strlen(0));
                        };
                       }

// CtoPstr and PtoCstr
                       if (pause("String Conversion") == CR)
                        {
                        char *Cstr;
                        struct P_Str
                         {
                          char count;
                          char contents[255];
                         } *Pstr;

                        Cstr = "ABCDEFG";
                        printf("\rC String = %s", Cstr);
                        Pstr = (struct P_Str *)CtoPstr(Cstr);
                        printf("\rP String = ");
                        for (i = 0; i < Pstr->count; ++i) putchar(Pstr->contents[i]);
                        printf("\rC String = %s", Cstr = PtoCstr(Pstr));
                        };


// int atoi(str) -- returns numeric conversion of number in str, radix = 10.
                       // long atol(str) -- like atoi but returns long
                       {
                       char *str1, *str2;
                       if (pause("atoi") == CR)
                        {
                        str1 = "0";
                        printf("\r\ratoi");
                        printf("\r  0 = %d", atoi("0"));
                        printf("\r  1 = %d", atoi("1"));
                        printf("\r  10 = %d", atoi("10"));
                        printf("\r  10000 = %d", atoi("10000"));
                        printf("\r  1000000 = %d", atoi("1000000"));
                        }
                       }


// scanf(format [, pointer]...) -- formatted input (see Kernighan and Ritchie)
                       // sscanf(s, format [, pointer]...) -- formatted input (see Kernighan and Ritchie)
                       // fscanf(file, format [, pointer]...) -- formatted input (see Kernighan and Ritchie)
                       {
                       char *str;
                       if (pause("scanf") == CR)
                        {
                        puts("\r  enter \"56789 0123 45a72\": ");

                          i = 0; x = 0; line[0] = 0; c1 = 0; s = 0;
                          scanf("%2d%3ld%*d%2s%c%2h", &i, &x, line, &c1, &s);
                          puts("\r should print:  ");
                          puts("\"i = 56, x = 789, str = 45 c = a, s = 72\"");
```

```
  printf("\r              ");
  printf("i = %d, x = %ld, str = %s, c = %c, s = %u\r", i, x, line, c1, s);

 puts("\rsscanf test:  ");
 str = "56789 0123 45a72";

  i = 0; x = 0; line[0] = 0; c1 = 0; s = 0;
  sscanf(str, "%2d%3d%*d%2s%c%2h", &i, &x, line, &c1, &s);
  printf("  i = %d, x = %d, str = %s, c = %c, s = %u\r", i, x, line, c1, s);

 puts("\rfscanf test:  ");
 file = creat("testLibFile", 7);
 fputs(str, file);
 close(file);
 file = fopen("testLibFile", "r");

  i = 0; x = 0; line[0] = 0; c1 = 0; s = 0;
  fscanf(file, "%2d%3d%*d%2s%c%2h", &i, &x, line, &c1, &s);
  printf("  i = %d, x = %d, str = %s, c = %c, s = %u\r", i, x, line, c1, s);
 close(file);
 }
}


// sprintf(s, format [, arg]...) -- formatted output (see Kernighan and Ritchie)
{
char str[MAXLINE+1];
if (pause("Sprintf") == CR)
 {
 i = 56; x = 789; c1 = 'a'; s = 72;
 strcpy(line,"45");
 sprintf(str,"i = %d, x = %d, str = %s, c1 = %c, s = %u", i, x, line, c1, s);
 printf("\r result:  %s", str);
 }
}

// fprintf(file, format [, arg]...) -- formatted output (see Kernighan and Ritchie)
{
if (pause("fprintf") == CR)
 {
 i = 56; x = 789;  c1 = 'a'; s = 72;
 strcpy(line,"45");
 file = creat("testLibFile", 7);
 fprintf(file,"i = %d, x = %d, str = %s, c1 = %c, s = %u", i, x, line, c1, s);
 close(file);
 file = fopen("testLibFile", "r");
 printf("\r result:  ");
 while ((c = getc(file)) > 0) putchar(c);
 close(file);
 }
}


// int getw(file) -- returns next word from file, ignores end-of-file.
if (pause("getw") == CR)
{
```

```
                        file = open("testLibFile", 2);
                        putw(0, file); putw(1, file); putw(2, file); putw(3, file);
                        putw(500, file); putw(501, file); putw(502, file); putw(503, file);
                        lseek(file, 0, 0);
                        printf("This should print: 0, 1, 2, 3, 500, 501, 502, 503\r");
                        printf("              ");
                        printf(" %d,", getw(file));
                        printf(" %d,", getw(file));
                        printf(" %d,", getw(file));
                        printf(" %d,", getw(file));
                        printf(" %d,", getw(file));
                        printf(" %d,", getw(file));
                        printf(" %d,", getw(file));
                        printf(" %d,", getw(file));
                        close(file);
                        }

// int getl(file) -- returns next word from file, ignores end-of-file.
if (pause("getl") == CR)
                        {
                        file = open("testLibFile", 2);
                        putl(0, file); putl(1, file); putl(2, file); putl(3, file);
                        putl(500, file); putl(501, file); putl(100502, file); putl(500503, file);
                        lseek(file, 0, 0);
                        printf("This should print: 0, 1, 2, 3, 500, 501, 100502, 500503\r");
                        printf("              ");
                        printf(" %d,", getl(file));
                        printf(" %d,", getl(file));
                        printf(" %d,", getl(file));
                        printf(" %d,", getl(file));
                        printf(" %d,", getl(file));
                        printf(" %d,", getl(file));
                        printf(" %d,", getl(file));
                        printf(" %d,", getl(file));
                        close(file);
                        }

// int fgets(file).

if (pause("fgets") == CR)
                        {
                        char str[MAXLINE+1];
                        file = open("testLibFile", 2);
                        fputs("Jack and Jill went up the hill\rTo fetch a pail of water", file);
                        lseek(file, 0, 0);
                        printf("This should print: Jack and Jill went up the hill\r");
                        printf("              ");
                        printf(" %s", fgets(str, 100, file));
                        printf("\rThis should print: To fetch a pail\r");
                        printf("              ");
                        printf(" %s", fgets(str, 16, file));
                        printf("\rFile is %sat End of File", feof(file)? "":"Not ");
                        printf("\rThis should print:  of water\r");
                        printf("              ");
```

```
printf(" %s", fgets(str, 16, file));
printf("\rFile is %sat End of File", feof(file)? "":"Not ");
printf("\rnext call on fgets = %x", fgets(str, 16, file));
close(file);
}
```

// int ungetc(c, file) -- puts c back onto file (one character maximum).

```
if (pause("ungetc") == CR)
 {
 file = open("testLibFile", 2);
 write(file, "ABCDEFG", 7);
 seteof(file);
 lseek(file, 0, 0);
 printf("This should print: ABCDEFG\r");
 printf("              ");
 c = getc(file);
 do
  {
   ungetc(c, file);
   printf("%1c", getc(file));
   } while ((c = getc(file)) != EOF);
 close(file);
 }
```

// unlink(name) -- deletes file identified by 'name' from disk UNLESS IT IS OPEN.

```
if (pause("unlink") == CR)
 {
 unlink("testLibFile");
 file = open("testLibFile", 0);
 if (file >= 0)
  {
   printf(" -- Failed\r");
   close(file);
  }
 else printf(" -- OK\r");
 }
```

// long read(file, buffer, n) -- reads up to n bytes from file into buffer.

```
if (pause("read/write") == CR)
 {
 char *str;
 int count;
 str = calloc(100, 1);
 file = open("testLibFile", 2);
 write(file, "ABCDEFG", 7);
 seteof(file);
 lseek(file, 0, 0);
 printf("This should print: 7: ABCDEFG\r");
 printf("              ");
 printf("%d", read(file, str, 100));
 printf(": %s", str);
```

```
printf("\r\rfread, fwrite\r");
free(str);
str = calloc(100, 1);
lseek(file, 0, 0);
printf("This should print: 3: ABCDEF\r");
printf("              ");
count = fread(str, 2, 4, file);
str[count*2] = 0;
printf("%d", count);
printf(": %s\r", str);
seteof(file);

lseek(file, 0, 0);
fwrite("ABCDEFGH", 2, 4, file);
lseek(file, 0, 0);
printf("This should print: 4: ABCDEFGH\r");
printf("              ");
printf("%d", fread(str, 2, 6, file));
printf(": %s", str);
close(file);
}
```

// long lseek(file, offset, mode) -- positions file according to mode:

```
if (pause("seek/tell") == CR)
 {
 char *str;
 file = open("testLibFile", 2);
 fwrite("ABCDEFGH", 2, 4, file);
 seteof(file);
 lseek(file, 0, 0);
 printf("This should print: 8, 0, 4, 4\r");
 printf("              ");
 lseek(file, 0, 2);
   printf("%d", tell(file));
 lseek(file, 0, 0);
   printf(", %d", tell(file));
 lseek(file, 4, 1);
   printf(", %d", tell(file));
 lseek(file, 0, 2);
   lseek(file, -4, 1);
   printf(", %d", tell(file));
 close(file);
 }
```

// swap(a, b) -- exchanges the contents of the long locations addressed by a and b.

```
if (pause("swap") == CR)
{
char *ptra, *ptrb;
ptra = (char *)&ptrb;
ptrb = (char *)&ptra;
printf("Pointers are:      %x, %x\r", ptra, ptrb);
swap(&ptra, &ptrb);
printf("Swapped Pointers are: %x, %x", ptra, ptrb);
}
```

```
if (pause("printf") == CR)
                        {
                        printf("\rThe following line pairs should match:");
                        printf("\r11 1 0 1");
                        x = -3+4*5-6;                   printf("\r%d ", x);
                        x = 3+4%5-6;                    printf("%d ", x);
                        x = -3*4%-6/5;                  printf("%d ", x);
                        x = (7+6)%5/2;                  printf("%d", x);

                        printf("\r\r1 2 3  4  5   6 000000780000000 9");
                        printf("\r%1d%2d%3d%4d%5d%6d %07d%-08d %-9d",1,   2,3,4,5,6,7,8,9,10);

                        printf("\r\r:hello, world:");
                        printf("\r:%10s:", "hello, world");

                        printf("\r\r(Strike a key for more)");

                        pause(0);

                        printf("\r\r:hello, world:");
                        printf("\r:%-10s:", "hello, world");

                        printf("\r\r:hello, world:");
                        printf("\r:%20s:", "hello, world");

                        printf("\r\r:hello, world:");
                        printf("\r:%-20s:", "hello, world");

                        printf("\r\r:       hello, wor:");
                        printf("\r:%20.10s:", "hello, world");

                        printf("\r\r:hello, wor       :");
                        printf("\r:%-20.10s:", "hello, world");

                        printf("\r\r:hello, wor:");
                        printf("\r:%.10s:", "hello, world");

                        }

printf("\r\r""End of Test"""\r(Strike a key to stop)");
pause(0);
unlink("testLibFile");
exit(0);
}
```

```
/*************************************************************/
/*                                                           */
/*                        MacDemo.c                          */
/*                                                           */
/*                Copyright 1984 Consulair Corporation       */
/*                        All Rights Reserved                */
/*                                                           */
/*                                                           */
/*                                                           */
/*************************************************************/

/*
This is a general Macintosh demonstration program.  It opens two windows, puts up menus, dispatches on events,
uses Text Edit, and handles desk accessories.  Windows can be grown, moved, and closed.  It is intended to
demontrate the flavor of Mac C in the Macintosh environment.

The program has a single conditional compilation flag;  If you allow UseWithStdLib to be defined, you must load
MacDemo with the Standard Library (or StdLib and the Mac C Toolkit).  If you don't define it, you only need to load
"MacCLib" with MacDemo.  This small version (use the link file smallDemo.link) doesn't print a TTY message out.
 */

 #define UseWithStdLib

/* Declarations */

 #include "MacCDefs.h"
 #ifdef UseWithStdLib
  #include "Stdio.h"
 #else
  extern struct P_Str *CtoPstr();
  int strlen(str) char *str;
   {int i=0; while (str[i++]); return i-1;}
 #endif
 #include "Window.h"
 #include "Events.h"
 #include "TextEdit.h"
 #include "Menu.h"

 /* Declared Here */

  MenuHandle DeskMenu;
  MenuHandle EditMenu;
  MenuHandle Menu;

  #define Desk_ID 200
  #define Edit_ID 201
  #define Menu_ID 202

  Rect screenRect  = {0, 0, 384, 512};
  Rect windowRectA                  = {50, 50, 200, 400};
  Rect windowRectB                  = {60, 60, 210, 410};

  WindowPtr openWindow();

  #define False 0
  #define True 0xFF
```

```
/* Declared Elsewhere */

  #ifdef UseWithStdLib
    extern WindowPtr console; /* Std Lib TTY Window */
  #endif

/* Code */

 Init()
  {
   InitDialogs(0);
   TEInit();
   InitMenus();

   /* Desk Accessory menu */
    DeskMenu = NewMenu(Desk_ID,CtoPstr("\024"));
    AddResMenu(DeskMenu, 'DRVR');
    InsertMenu(DeskMenu, 0);

   /* Edit menu */
    EditMenu = NewMenu(Edit_ID, CtoPstr("Edit"));
    AppendMenu(EditMenu,
      CtoPstr("Undo;(-;Cut/X;Copy/C;Paste/V;Clear"));
    InsertMenu(EditMenu, 0);
    DisableItem(EditMenu, 0);

   /* "Menu" menu */
    Menu = NewMenu(Menu_ID, CtoPstr("Menu"));
    AppendMenu(Menu,
      CtoPstr("Item 1;(Dimmed Item 2;Item 3;(-;Item 5/5;Quit/."));
    InsertMenu(Menu, 0);

   DrawMenuBar();
  }


 main()
  {
  char c, *str;
  short windowcode;
  long menuResult;

  EventRecord event;
  TEHandle hTE;
  WindowPtr mouseWindow, window, windowA, windowB;

  if (CatchSignal()) ExitToShell();

  Init();

  #ifdef UseWithStdLib

    printf("\r\r      Mac C Demo\rCopyright Consulair Corporation 1984\rAll Rights Reserved\r\r");
    printf("\rThis is a simple demonstration program with two windows.\r");
    printf("\rWhen you strike a key, this TTY window will be erased, and");
```

```
  printf("\rTwo windows will be displayed.  Typed text will go into the front");
  printf("\rone, and you can switch between the two with a mouse click.");
  printf("\rSelect Quit from the menu or hit a command period to stop.");

  getchar();
  DisposeWindow(console);

#endif

windowB =
  openWindow(&windowRectB, "Demo Window B",
    "\rMac C Demo\r\251 Consulair Corporation 1984\r All Rights Reserved\r\rThis is Window B");

windowA =
  openWindow(&windowRectA, "Demo Window A",
    "\rMac C Demo\r\251 Consulair Corporation 1984\r All Rights Reserved\r\rThis is Window A");

hTE = 0;
InitCursor();
FlushEvents(-1);
SelectWindow(windowA); /* Generate an activate event for window A */

while (True)
 {
  SystemTask();
  if (hTE)  TEIdle(hTE);

  if (GetNextEvent(everyEvent, &event))
   {
    switch ( event.what )
     {
      case autoKey:
      case keyDown:
       {
        c = event.message;
        if ((event.modifiers & cmdKey))
          DoMenu(MenuKey(c));
         else TEKey(c, hTE);
        break;
       }


     case mouseDown:
       {
        windowcode = FindWindow(&event.where, &mouseWindow);

        if (FrontWindow() != mouseWindow)
          if (mouseWindow != 0)
                    {
                    SelectWindow(mouseWindow);
                    break;
                    }

        if ((window != 0) && (window == mouseWindow))
```

```
{
if (mouseWindow != 0)
 {
 SetPort(mouseWindow);
 switch ( windowcode )
  {
  case inContent:
   {
   GlobalToLocal(&event.where);
   TEClick(&event.where,
                        (event.modifiers & shiftKey)? True:False, hTE);
   break;
   }
  case inDrag:
   {
   DragWindow(mouseWindow, &event.where, &screenRect);
   break;
   }
  case inGrow:
   {
   long growResult;
   short vert, horiz;
   growResult =
                        GrowWindow(mouseWindow, &event.where, &screenRect);
   horiz = growResult;
   vert = HiWord(growResult);
   SizeWindow(mouseWindow, horiz, vert, True);
   EraseRect(&mouseWindow->portRect);
   InvalRect(&mouseWindow->portRect);
   SizeTE(mouseWindow);
   DrawGrowIcon(mouseWindow);
   break;
   }
  case inGoAway:
   {
   if (TrackGoAway(window, &event.where))
                        {
                        TEDispose(hTE);
                        hTE = 0;
                        DisposeWindow(window);
                        window = 0;
                        }
   break;
   }
  }
 }
}
else
{
switch ( windowcode )
 {
 case inMenuBar:
  {

  DoMenu(MenuSelect(&event.where));
  break;
```

```
                    }
                    case inSysWindow:
                    {
                     SystemClick(&event, mouseWindow);
                     break;
                    }
                    case inDrag:
                    {
                     DragWindow(mouseWindow, &event.where, &screenRect);
                     break;
                    }
                    case inGoAway:
                    {
                     break;
                    }
                   }
                  }

    break;
    }
   case updateEvt:
    {
    TEHandle temp_hTE;
    WindowPtr tempWindow;

    SetPort(tempWindow = (WindowPtr)event.message);
    BeginUpdate(tempWindow);
    temp_hTE = (TEHandle)GetWRefCon(tempWindow);
    TEUpdate(&tempWindow->portRect, temp_hTE);
    DrawGrowIcon(tempWindow);
    EndUpdate(tempWindow);
    break;
    }

  case activateEvt:
    {
    TEHandle temp_hTE;
    WindowPtr tempWindow;

    SetPort(tempWindow = (WindowPtr)event.message);
    temp_hTE = (TEHandle)GetWRefCon(tempWindow);
    if ((event.modifiers & activeFlag))
                    {
                    window = tempWindow;
                    TEActivate(hTE = temp_hTE)
                    }
      else TEDeactivate(temp_hTE);
    DrawGrowIcon(tempWindow);
    break;
    }
   }
  }

 }
}
```

```
DoMenu(menuresult)
 long menuresult;
 {
 short menuID, itemNumber;

 menuID = HiWord(menuresult);
 itemNumber = menuresult;

 switch ( menuID )
  {
   case Menu_ID:
     {
     switch ( itemNumber )
      {
       case 1: break;                                   /* item 1 */
       case 2: break;                                   /* item 2 */
       case 3: break;                                   /* item 3 */
       case 5: break;                                   /* item 5 */
       case 6:                                          /* item 6 is quit */
        Signal("All Done");
      }
     break;
     }
   case Desk_ID:
     {
     struct P_Str AccessoryName;
     GetItem(DeskMenu, itemNumber, &AccessoryName);
     OpenDeskAcc(&AccessoryName);
     EnableItem(EditMenu, 0);
     DrawMenuBar();
     break;
     }
   case Edit_ID:
     {
     SystemEdit(itemNumber-1);
     break;
     }
  }
 HiliteMenu(0);
 }


Rect *TERect(window, rect)
 WindowPtr window;
 Rect *rect;
 {
 BlockMove(&window->portRect, rect, sizeof(Rect));
 rect->right -= 16;  /* Make room for scroll bar */
 rect->bottom -= 16;                    /* Make room for scroll bar */
 }

TEHandle openTE(window)
 WindowPtr window;
 {
 Rect destRect, viewRect;
```

```
 TEHandle hTE;

 TERect(window, &viewRect);
 TERect(window, &destRect);
 destRect.left += 4;                                    /* indent a bit */
 return TENew(&destRect, &viewRect);
 }

WindowPtr openWindow(rect, title, str)
 Rect *rect;
 char *title, *str;
 {
 WindowPtr window;
 TEHandle hTE;
 window = NewWindow(0, rect, CtoPstr(title), True, 0, -1, True, 0);
 SetPort(window);
 hTE = openTE(window);
 SetWRefCon(window, hTE);
 TESetText(str, strlen(str), hTE);
 TEUpdate(&(*hTE)->viewRect, hTE);
 return window;
 }

sizeTE(window)
 WindowPtr window;
 {
 Rect rect;
 TEHandle hTE;

 hTE = (TEHandle)GetWRefCon(window);
 TERect(window, &rect);
 BlockMove(&rect, &(*hTE)->viewRect, sizeof(Rect));
 rect.left += 4;                                /* indent a bit */
 BlockMove(&rect, &(*hTE)->destRect, sizeof(Rect));
 TECalText(hTE);
 }
```

William S. Duvall has been programming in the computer business for 20 years. He has written at least one of almost everything! In 1976, he founded Consulair Corp. after working and consulting for many companies including SRI International, Xerox PARC, and Apple Computer. Consulair's first commercial products are Mac C and Mac C Toolkit.

Bill's loves other than programming include his wife, Ann (Vice President of Consulair), his three children, bicycle riding, restoring old English sports cars, hiking and flying.

Consulair Corp 140 Campo Drive, Portola Valley, CA 94025