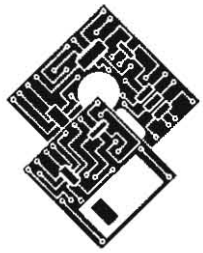


APPLE
PROGRAMMER'S
AND DEVELOPER'S
ASSOCIATION

290 SW 43rd. Street
Renton, WA 98055
206-251-6548

APW C Language Reference

APDA Draft
March 9, 1987

APDA# K2SAPC

Apple IIGS Programmer's Workshop

C Reference

APDA Draft
9 March 1987

This document contains preliminary information. It does not include

- ***final editorial corrections***
- ***final art work***
- ***an index***

It may not include final technical changes.

Apple Technical Publications, MS 22-K

Engineering Part Number: 030–3133
Marketing Part Number: A2L6003

Simultaneously published in the United States and Canada.

Apple IIGS Programmer's Workshop

C Language Reference

Contents

Chapter 2: Using the APW C Compiler

Chapter 3: Sample Program

Part II: Language Reference

Chapter 4: The APW C Language

Chapter 5: The Standard C Library

Chapter 6: Shell Calls

Appendix A: Calling Conventions

Appendix B: Files supplied with APW C

Appendix C: Comparison with Macintosh Workshop C

Appendix D: Library Index

(Contains an index entry for every define, type, enumeration literal, global variable, and function defined in the Standard C Library and the APW Shell.)

Glossary

About This Manual

This manual contains the information about Apple IIGS™ Programmer's Workshop C that you need when writing C programs for the Apple IIGS. It assumes that most readers already know the C programming language, as defined in Kernighan and Ritchie's *The C Programming Language*. For this reason, it does not repeat their definition of the C language, but instead defines the differences between APW C and "K and R" C. However, this manual can also be used by those learning C for the first time. The introductory chapters tell how to write, compile, link, and run a simple C program. From there, one can follow Kernighan and Ritchie or any standard textbook on C.

Roadmap to the Apple IIGS Technical Manuals

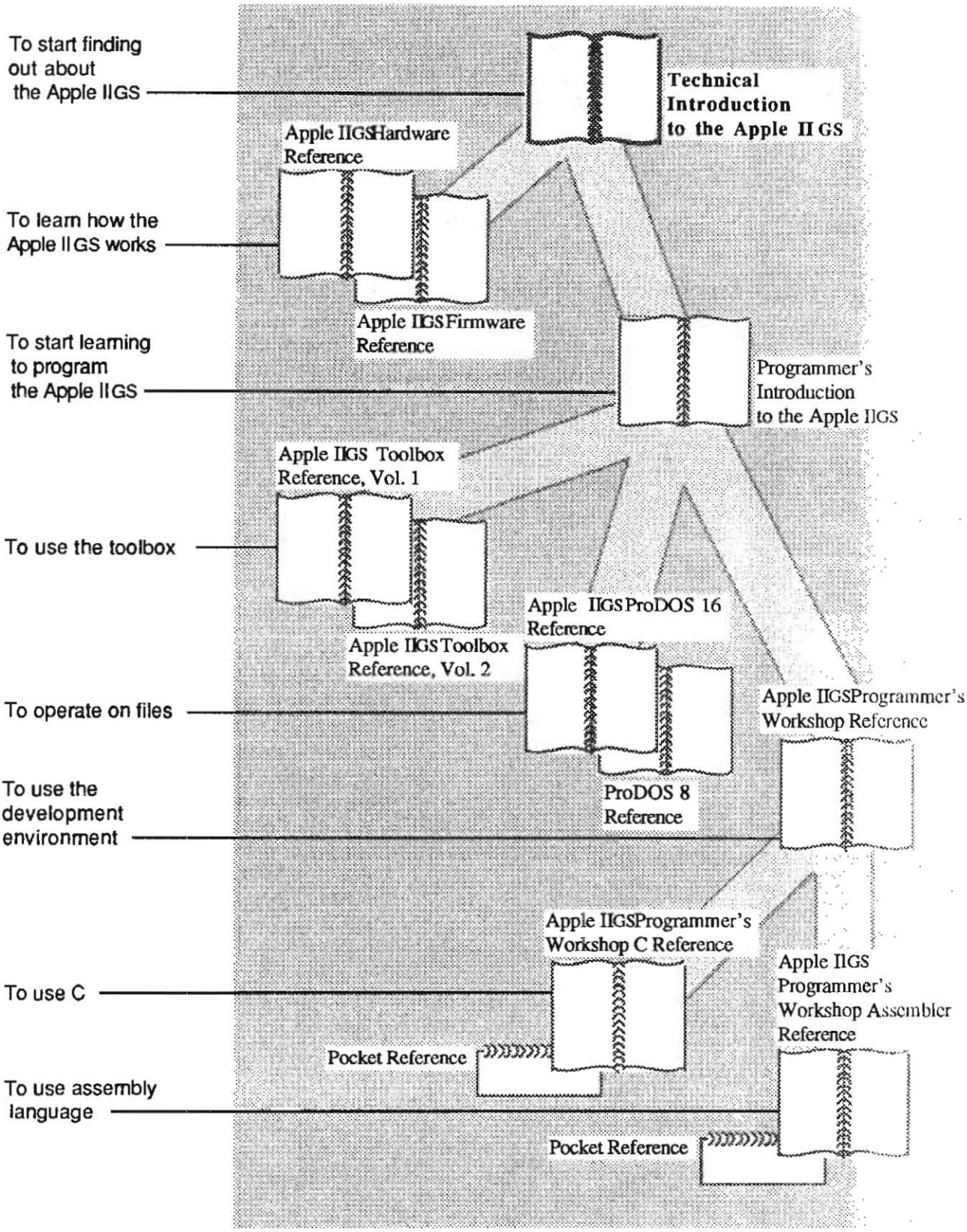
The Apple IIGS personal computer has many advanced features, making it more complex than earlier models of the Apple II™ computer. To describe the Apple IIGS fully, Apple has produced a suite of technical manuals. Depending on the way you intend to use the Apple IIGS, you may need to refer to a select few of the manuals, or you may need to refer to most of them.

The technical manuals are listed in Table P-1. Figure P-1 is a diagram showing the relationships between the different manuals.

Table P-1
The Apple IIGS technical manuals

Title	Subject
Technical Introduction to the Apple IIGS	What the Apple IIGS is
Apple IIGS Hardware Reference	Machine internals—hardware
Apple IIGS Firmware Reference	Machine internals—firmware
Programmer's Introduction to the Apple IIGS	Concepts and a sample program
Apple IIGS Toolbox Reference, Volume 1	How the tools work and some toolbox specifications
Apple IIGS Toolbox Reference, Volume 2	More toolbox specifications
Apple IIGS Programmer's Workshop Reference	The development environment
Apple IIGS Programmer's Workshop Assembler Reference	Using the APW Assembler
Apple IIGS Programmer's Workshop C Reference	Using C on the Apple IIGS
ProDOS 8 Reference	ProDOS for Apple II programs
Apple IIGS ProDOS 16 Reference	ProDOS and loader for Apple IIGS
Human Interface Guidelines: the Apple Desktop Interface	Guidelines for the desktop interface
Apple Numerics Manual	Numerics for all Apple computers

Figure P-1
Roadmap to the technical manuals



Note: Some of the book titles in the above diagram have changed since it was drawn. A corrected drawing is being prepared.

The following sections briefly describe the manuals listed in Table P-1 and Figure P-1.

Introductory manuals

These books are introductory manuals for developers, computer enthusiasts, and other Apple IIGS owners who need technical information. As introductory manuals, their purpose is to help the technical reader understand the features of the Apple IIGS, particularly the features that are different from other Apple computers. Having read the introductory manuals, the reader will refer to specific reference manuals for details about a particular aspect of the Apple IIGS.

The technical introduction

The *Technical Introduction to the Apple IIGS* is the first book in the suite of technical manuals about the Apple IIGS. It describes all aspects of the Apple IIGS, including its features and general design, the program environments, the toolbox, and the development environment.

Where the *Apple IIGS Owner's Guide* is an introduction from the point of view of the user, the *Technical Introduction* describes the Apple IIGS from the point of view of the program. In other words, it describes the things the programmer has to consider while designing a program, such as the operating features the program uses and the environment in which the program runs.

You should read the Technical Introduction no matter what kind of programming you indend to do, because it will help you understand the powers and limitations of the machine. If you are going to be doing assembly-language or system programming, this book is essential. To find out all about any one aspect of the Apple IIGS, you should read one of the following specific technical manuals.

The Programmer's Introduction

When you start writing programs that use the Apple IIGS user interface (with windows, menus, and the mouse), the *Programmer's Introduction to the Apple IIGS* provides the concepts and guidelines you need. It is not a complete course in programming, only a starting point for programmers writing applications for the Apple IIGS. It introduces the routines in the Apple IIGS Toolbox and the program environment they run under. It includes a sample **event-driven program** that demonstrates how a program uses the toolbox and the operating system.

Machine reference manuals

There are two reference manuals for the machine itself: the *Apple IIGS Hardware Reference* and the *Apple IIGS Firmware Reference*. These books contain detailed specifications for people who want to know exactly what's inside the machine.

If you are doing system programming or writing programs that are designed to recognize whether they are running on the Apple IIGS or older Apple II computers, these books are essential.

The hardware reference manual

The *Apple IIGS Hardware Reference* is required reading for hardware developers, and it will also be of interest to anyone else who wants to know how the machine works. Information for developers includes the mechanical and electrical specifications of all connectors, both internal and external. Information of general interest includes descriptions of the internal hardware, which provide a better understanding of the machine's features.

The firmware reference manual

The *Apple IIGS Firmware Reference* describes the programs and subroutines that are stored in the machine's read-only memory (ROM), with two significant exceptions: Applesoft BASIC and the toolbox, which have their own manuals. The *Firmware Reference* includes information about interrupt routines and low-level I/O subroutines for the serial ports, the disk port, and for the DeskTop Bus interface, which controls the keyboard and the mouse. The *Firmware Reference* also describes the Monitor, a low-level programming and debugging aid for assembly-language programs.

The toolbox manuals

Like the Macintosh, the Apple IIGS has a built-in toolbox. The *Apple IIGS Toolbox Reference,* Volume 1, introduces concepts and terminology and tells how to use some of the tools. The *Apple IIGS Toolbox Reference,* Volume 2, contains information about the rest of the tools. Volume 2 also tells how to write and install your own tool set.

Of course, you don't have to use the toolbox at all. If you only want to write simple programs that don't use the mouse, or windows, or menus, or other parts of the **desktop user interface,** then you can get along without the toolbox. However, if you are developing an application that uses the desktop interface, or if you want to use the Super Hi-Res graphics display, you'll find the toolbox to be indispensable.

The Programmer's Workshop manual

The development environment on the Apple IIGS is the Apple IIGS Programmer's Workshop (APW). APW is a set of programs that enable developers to create and debug application programs on the Apple IIGS. The *Apple IIGS Programmer's Workshop Reference* includes information about the parts of the workshop that all developers will use, regardless which programming language they use: the shell, the editor, the linker, the debugger, and the utilities. The manual also tells how to write other programs, such as custom utilities and compilers, to run under the APW Shell. (For brevity, we will usually refer to this as the *APW Reference.*)

The *APW reference* describes the way you use the workshop to create an application and includes a sample program to show how this is done.

Programming-language manuals

Apple is currently providing a 65816 assembler and a C compiler. Other compilers can be used with the workshop, provided that they follow the standards defined in the *APW Reference.*

There is a separate reference manual for each programming language on the Apple IIGS. Each manual includes the specifications of the language and of the Apple IIGS libraries for the language, and describes how to write a program in that language. The manuals for the languages Apple provides are the *Apple IIGS Programmer's Workshop Assembler Reference* and the *Apple IIGS Programmer's Workshop C Reference.*

Operating-system manuals

There are two operating systems that run on the Apple IIGS: ProDOS 16 and ProDOS 8. Each operating system is described in its own manual: *ProDOS 8 Reference* and *Apple IIGS ProDOS 16 Reference.* ProDOS 16 uses the full power of the Apple IIGS and is not compatible with earlier Apple II's. The ProDOS 16 manual includes information about the System Loader, which works closely with ProDOS 16. If you are writing programs for the Apple IIGS, whether as an application programmer or a system programmer, you are almost certain to need the *ProDOS 16 Reference.*

ProDOS 8, previously just called *ProDOS*, is compatible with the models of Apple II that use 8-bit CPUs. As a developer of Apple IIGS programs, you need to use ProDOS 8 only if you are developing programs to run on 8-bit Apple II's as well as on the Apple IIGS.

All-Apple manuals

In addition to the Apple IIGS manuals mentioned above, there are two manuals that apply to all Apple computers: *Human Interface Guidelines—The Apple Desktop Interface;* and the *Apple Numerics Manual.* If you develop programs for any Apple computer, you should know about those manuals.

The *Human Interface Guidelines* manual describes Apple's standards for the desktop interface to any program that runs on an Apple computer. If you are writing a commercial application for the Apple IIGS, you should be fully familiar with the contents of this manual.

The *Apple Numerics Manual* is the reference for the Standard Apple Numeric Environment (SANE™), a full implementation of the IEEE standard for floating-point arithmetic. The functions of the Apple IIGS SANE tool set match those of the Macintosh SANE packages and of the 6502 Assembly-Language SANE™ software. If your application requires accurate or robust arithmetic, you'll probably want to use the SANE routines in the Apple IIGS. The *Apple IIGS Toolbox Reference* tells how to use the SANE tool set routines in your programs. The *Apple Numerics Manual* is the comprehensive reference for the semantics of the SANE routines.

How to use this manual

If you are an experienced C programmer but have never written a program for the Apple IIGS, Chapters 1, 2, and 3 will give you enough information to get standard C programs running. (If you have written other programs for the Apple IIGS, Chapter 1 will be redundant.) The remaining chapters tell you what you need to write C programs that use the capabilities of the Apple IIGS.

If you are new to C, Chapter 1 will tell you what you need to go through a C textbook, such as Kernighan and Ritchie's. After you are familiar with C, you can learn about the capabilities of the C compiler and this particular implementation.

What this manual contains

This manual is divided into two major sections: Part I, " A Programmer's Guide," and Part II, "Language Reference."

Part I, "A Programmer's Guide," introduces you to APW C and its programming environment.

- Chapter 1, "Getting Started," introduces the environment in which you'll use the C compiler. It discusses the Apple IIGS Programmer's Workshop, ProDOS 16, the Apple IIGS Tools, and lists the hardware and software you'll need.
- Chapter 2, "Using the C Compiler," describes the compilation process, lists the Shell commands you'll need working with the compiler, and discusses the linker, the debugger, and other utilities.
- Chapter 3, "Sample Program," takes you step-by-step through the process of building a C program that has an assembly language subroutine.

Part II, "Language Reference," is a detailed description of the structure and components of the APW C and its libraries.

- Chapter 4, "The APW C Language," describes Apple extensions to C and clarifies aspects of the language definition as they apply to this implementation.
- Chapter 5, "The Standard C Library," documents functions for standard I/O, string manipulation, math routines, and other useful features not built into the language.
- Chapter 6, "The Shell Interface Library," lists the C interfaces to the APW Shell.
- Appendix A, "Calling Conventions," tells how to write calls between C and Pascal.
- Appendix B, "Files Supplied with APW C," contains a list of all the files that are supplied with this product.
- Appendix C, "Comparison with Macintosh Programmer's Workshop C," describes the differences between MPW C and APW C.
- Appendix D, "Library Index," is a combined index of identifiers in the Standard C Library and Apple IIGS Interface Libraries.

Visual cues

Certain conventions in this manual provide visual cues alerting you, for example, to the introduction of a new term and important or useful information. These are described in this section. Typographical conventions are described in the next section, "Language Notation."

New terms

When a new term is introduced, it is printed in **boldface** the first time it is used. This lets you know that the term has not been defined earlier and that there is an entry for it in the glossary.

Notes and warnings

Special messages of note are marked as such:

- **Note:** Text set off in this way presents sidelights or interesting points of information.
- **Important:** Text set off in this way presents important information or instructions that you should read before proceeding.
- **Warning!** A warning set off like this alerts you to something that cauld cause loss of data or damage to software.

Language notation

This manual uses certain conventions in common with other Apple IIGS language manuals.

- Words and symbols that are part of the C language, as well as anything that you type on the keyboard or that can appear on the screen, are presented in a monospace font:

```
int ndigit[10]
```

- Metalanguage expressions, which are used in syntax diagrams to indicate items that are replaced by C, are in italic:

```
else if (condition)
   statement
```

  Here *condition* and *statement* are expressions that are replaced by actual C expressions. The `else if` and the parentheses are C code.

In addition, the following conventions are observed:

Convention	Meaning
[ ]	Square brackets indicate that the enclosed item is optional.
...	A horizontal ellipsis indicates that the preceding item(s) can be repeated as necessary.
. . .	A vertical ellipsis indicates that not all of the statements in an example or figure are shown.

Other reference material you'll need

In order to write C programs for the Apple IIGS, you'll need to be familiar with these additional reference materials:

- *Apple IIGS Programmer's Workshop Reference.* This book describes the APW environment in which the C compiler operates, including the shell, editor, linker, debugger, and other important utilities.

- *The C Programming Language,* by Brian W. Kernighan and Dennis M. Ritchie (Prentice-Hall, 1978). This is a standard reference book for the C language in its original form. Appendix A of this book is a formal definition of "K & R" C.
- *C: A Reference Manual,* by Samuel P. Harbison and Guy L. Steele (Prentice-Hall, 1985). This is a complete reference book for standard C, as implemented by the Portable C Compiler, including the Western Electric extensions to K & R C.
- *Apple IIGS Toolbox Reference, Volumes I and II.* These books contain everything you need to program using the Apple IIGS ROM and associated RAM routines. The two volumes cover windows, alert boxes, menus, graphics, the SANE tool set, and much more.
- *Apple Numerics Manual.* This book describes in detail the floating-point arithmetic used in Apple computers. See the *Toolbox Reference* for a detailed description of the calling sequence for SANE routines.

Part I

Programmer's Guide

Chapter 1
Overview

This chapter introduces the Apple IIGS Programmer's Workshop (APW). The first section, "About the Apple IIGS Programmer's Workshop," describes the various parts of APW. The second section, "About Apple IIGS System Software," describes ProDOS 16, the System Loader, and the Memory Manager. The third section, "What You Need," describes the hardware and software you need to run APW C. The fourth section, "APW C Concepts," describes the relationships between source, object, load, and library files. The fifth section, "Program Interactions," describes the process of building a program.

About the Apple IIGS Programmers Workshop

The Apple IIGS Programmer's Workshop is a suite of software designed to assist developers in writing Apple IIGS applications programs. This development environment includes a **command interpreter**, known as the **shell**; a text **editor**; a **linker**; a **debugger**; and a set of utilities. APW supports C and 65816/65C02 assembly-language programming; other languages are planned. Further support for developers is provided by a comprehensive set of routines known as the **Apple IIGS Toolbox**. The toolbox routines are accessed from APW but are not considered part of APW. For a comprehensive description of APW, refer to the *Apple IIGS Programmer's Workshop Reference*. For detailed information on the Apple IIGS Toolbox, refer to the *Apple IIGS Toolbox Reference: Volumes I and II*.

The APW shell

The APW shell provides the interface that allows you to work with the C compiler and perform tasks such as file, directory, and disk management. The shell also acts as an extension to ProDOS 16, providing several functions that can be called by programs running under the shell. The C compiler can use a set of shell calls to perform the following functions:

- pass parameters and operations flags between the shell and APW programs
- read the current language number
- set the current language number
- return the address of the command table
- get filenames using wildcards

APW C provides C interfaces to the shell calls. The calls and their C interfaces are discussed in Chapter 6, "Shell Calls."

Commands most often used while working with the C compiler are described in Chapter 2, "Using the APW C Compiler." The APW shell is fully described in Chapters 2 and 3 of the *APW Reference*.

The APW editor

The APW editor is a full-screen text editor that operates under keyboard control.

You can send commands to the shell to perform tasks such as

- manipulating text
- searching for and replacing text strings
- moving your position in the file
- scrolling the screen
- setting and clearing tab stops
- defining and using keyboard macros

The APW editor is fully described in Chapters 2 and 4 of the *APW Reference*.

The APW linker

The APW linker takes the **object files** produced by the C compiler and generates load files that the System Loader can load into memory. Although the linker is a single program, conceptually there are two APW linuers:

1. Normally the linker is called by a shell command, such as LINK or CMPL (compile and link). These commands provided a limited set of options, setting other options to default values. This linker is referred to as the **standard linker.**

2. Alternatively, all funtcions of the APW Linker can be contorlled by compiling a file of linker commands. The linker command language, called LinkEd, allows you to do such things as place specific object-file segments in specific load-file segments, search specific libraries, and control linker printout. You can append the LinkEd file to your last source file, or you can compile and execute them separately by using the COMPILE or ALINK commands. The aspect of the linker controlled by LinkEd files is called the **advanced linker.**

About APW C

APW C is a complete implementation of the C programming language. It consists of a C compiler, the Standard C Library, the Apple IIGS Interface Libraries, and the C SANE Library.

The C Programming Language by Kernighan and Ritchie is an authoritative written definition of C in its original form: we refer to this C as *K & R C*. However, the language has changed in several ways since the book was written. In addition, numerous details of the language definition are open to interpretation, with the result that the de facto standard definition of C differs in several ways from the language originally defined by Kernighan and Ritchie. This de facto standard is loosely defined by the most widely used implementation of C, the Portable C Compiler (PCC).

In this manual, we use the term *Standard C* for C as defined and implemented by the Berkeley 4.2 BSD VAX implementation of PCC, including the documented Western

Electric extensions: type `void`, enumeration data types, and structures as function parameters and results. *C: A Reference Manual,* by Harbison and Steele, describes Standard C fully. APW C is based on this de facto standard and not on the proposed ANSI standard currently under development.

Apple has extended Standard C to facilitate writing programs for the Apple IIGS. In addition to the Western Electric extensions, APW C includes a function modifier that allows calls to and from Pascal programs and the Apple IIGS Interface Libraries. APW C also supports the Standard Apple Numeric Environment, described later in this chapter.

Mode of operation

The APW C compiler, and APW C itself, operates in the Apple IIGS's native mode. In **native mode,** the full instruction set of the 65816 processor is available. The 91 instructions combined with 25 addressing modes make 256 opcodes available to the compiler. The register set can be used for either 8- or 16-bit operations. The accumulator can be set to either a 16-bit register or two 8-bit registers. The advantage of using a processor with 16-bit registers as compared to one with 8-bit registers is that you can write shorter programs with more compact code and faster execution.

Standard Apple Numeric Environment

The APW C compiler provides full support for the Standard Apple Numeric Environment (SANE™). APW C together with the C SANE library compose a fully conforming implementation of extended-precision binary floating-point arithmetic as specified by IEEE Standard 754. This standard specifies data types, arithmetic, and conversions, as well as tools for handling exceptions such as overflow and division by zero. SANE supports all requirements of the IEEE standard and goes beyond the specifications of the standard by including a library of high-quality scientific and financial functions. Thus SANE provides a numeric environment sufficient for a wide range of applications.

Source programs using only the `float` and `double` types and standard C operations compile and run without modification.

Object module format

The **object module format** (OMF) on the Apple IIGS is the general format used in **object files, library files,** and **load files.** On the Apple IIc and IIe, there is only one loadable file format, called the binary file format, which consists of one absolute memory image along with its destination address. On the Apple IIGS, object module format allows dynamic loading and unloading of **load segments** containing program code and data while a program is running. Additionally, each APW language produces its object code in the object module format, allowing you to link together subroutines written in different languages.

About the Apple IIGS system software

System tasks are handled by ProDOS 16, the **System Loader**, and the **Memory Manager**. ProDOS 16 is the core, or kernel, of the Apple IIGS's operating system. It provides file management and input/output capability.

The System Loader works closely with ProDOS 16. It is responsible for loading all code and data into the Apple IIGS memory. It is capable of static and dynamic loading and relocating of load segments.

The Memory Manager is responsible for allocating memory. It provides space for load segments, tells the System Loader where to place them, and moves segments within memory when additional space is needed.

ProDOS 16 and the System Loader are documented in the *Apple IIGS ProDos 16 Reference*. The Memory Manager is documented in both the *Apple IIGS ProDos 16 Reference* and the *Apple IIGS Toolbox Reference: Volumes I and II.*

What you need

In order to use the Apple IIGS Programmer's Workshop, you must have the following hardware and software. A list of Apple IIGS manuals that you will find useful is given in the Preface.

- An Apple IIGS computer, or an Apple IIe computer with an installed Apple IIGS-upgrade, with 256K bytes of RAM.
- An installed Apple IIGS memory-expansion card with 512K bytes of RAM. With this card the Apple IIGS has 768K bytes of RAM.
- The 3.5-inch Apple IIGS System Disk.
- The two 3.5-inch APW disks.
- The 3.5-inch APW C disk, containing the files shown in Appendix B.
- Two 800K disk drives (one if you have a hard disk).
- Disks containing any other APW languages you intend to use with this system. The files on these disks must be installed on the Apple IIGS disk as described in the manuals that came with them.

The following hardware is highly recommended, especially if you intend to do multi-language development or develop large programs:

- An Apple IIGS memory-expansion card with one megabyte of RAM. With this card installed, the Apple IIGS has 1.25 Mbytes of RAM.
- A hard disk, such as the Apple HardDisk 20 SC, or a third 800K disk drive.

Many developers find that an additional Apple IIGS memory-expansion card is very useful. You can use the card for a large RAM disk on which you can place library files, compilers and assemblers, the linker, and utility programs. Since these programs are loaded into memory from disk each time they are used, placing them on a RAM disk can speed up the system's operation during program development.

> **Note:** If you haven't yet read "About this Manual," go back and read it now. In addition to providing a list of the manuals you'll need to develop programs for the Apple IIGS, it explains the layout of this book, the interrelationships of the books in the Apple IIGS Technical Library suite, and the conventions used to describe commands in this book.

The APW C disk contains the files shown in Appendix B. Use the index of this manual to get more information on any of these files. To examine the contents of your APWC disk, boot the disk, type `CAT` and press Return. To examine the contents of a subdirectory, include the pathname of the subdirectory; for example, to obtain a listing of the files in the subdirectory `/APWC/LIBRARIES`, use the following command:

```
CAT /APWC/LIBRARIES
```

To obtain a listing of all the files in the volume `/APWC`, use the command

```
FILES -L -R /APWC
```

This prints the contents of all the directories in the volume, the files in each directory indented below it, with information about each file.

APW C concepts

This section describes a variety of features and concepts that you must understand in order to write application programs for the Apple IIGS computer. While some of these concepts may be familiar to you from work with other computers, you must still be familiar with the way in which they are implemented on the Apple IIGS to get the most out of the Apple IIGS Programmer's Workshop and to use the operating system and the memory of the Apple IIGS effectively.

Relocatable load files

The Apple IIGS Programmer's Workshop deals with three fundamental types of files: **source files**, **object files**, and **load files**. Source files are ASCII files consisting of the text of your program, and follow the conventions of a particular programming language; object files and load files are binary files conforming to the Apple IIGS **object module format** (OMF) defined in Chapter 8 of the *APW Reference*.

A C source file consists of C statements, preprocessor directives, function definitions and declarations, and so forth, together with variable declarations, which may include initialized data. In the source code, specific functions, variables, and data structures are each labelled with a name. You can refer to the name in another part of the program: for example, you execute a function by using its name in a statement. A name or label of code or data used in this way is referred to as a **symbolic reference** (that is, a *symbol* that can be *referenced* or referred to). In high-level programming languages, like C, symbolic references are usually the only means available to jump from one place in a program to another.

C uses a special kind of source file—a **header** file, or **include** file—containing code shared by many programs: for instance, lists of constants or interfaces to libraries. The

header file is named in an `#include` statement in your source file, and the C compiler copies the header file in place of the `#include` statement before doing the actual compilation.

In assembly language it is possible to specify actual locations in the computer's memory to which you want the program to jump: that is, to write **absolute code.** The APW C compiler only produces **relocatable code segments:** that is, code that can be loaded into any location in memory. Note that such a program can be relocated only when it is loaded: once loaded, it can't be moved. (A program or block of code that can be moved from one location in memory to another while the program is running is called **position independent.**)

The Apple IIGS system software and APW are both designed to support relocatable code.

When a source program is compiled, the compiler converts the source code into 65816 machine-language instructions, data declarations, and symbolic references. Before the program is actually run, the symbolic references must be **resolved**; that is, the routine being referenced must be found, and the reference must be replaced with code that the loader can use to relocate the code at load time. The program that resolves the symbolic references is called the APW **Linker**. (The linker gets its name from the fact that it can combine, or link together, several object files and library files to create a single executable load file.)

The conversion of a source file into 65816 machine language and data resident in memory is done in several steps, as follows (see Figure 1.1, below):

1. The source code is compiled. The APW C compiler first executes preprocessor directives, such as inserting include files, before compiling the source code and writing out one or more object files. Object files, then, consist of machine-language instructions plus unresolved symbolic references to other routines.

   Your program can consist of several source files, and each source file can be in any of the APW programming languages. Each source file is converted into one or more object files by the APW assembler and compilers.

2. The object files are input to the APW Linker, which combines all of the object files into a single load file and resolves symbolic references. The linker verifies that every routine referenced is included in the load file. If there are any routines that the linker has not found when it has finished processing all of the object files, then it searches through any available library files for the missing routines. The linker removes symbolic references and replaces them with entries in special tables it creates called **relocation dictionaries**. The load file consists of blocks of machine-language code that can be loaded directly into memory (called **memory images**), plus relocation dictionaries that contain the information necessary to patch addresses into the memory images when the program is loaded into memory.

3. At program execution time, the load file is loaded into memory by the System Loader. The loader calls the Apple IIGS Memory Manager to request blocks of memory for the load file, loads the memory images, and uses the relocation dictionaries to patch the actual memory addresses into the machine-language code in memory. The entire load file is not necessarily loaded into memory at one time; all OMF files are divided into **segments**, which can be processed independently. OMF-file segmentation is a fundamental Apple IIGS concept, which we consider next.

The Memory Manager is the Apple IIGS toolset that allocates blocks of memory as needed, and keeps track of which blocks of memory are available.

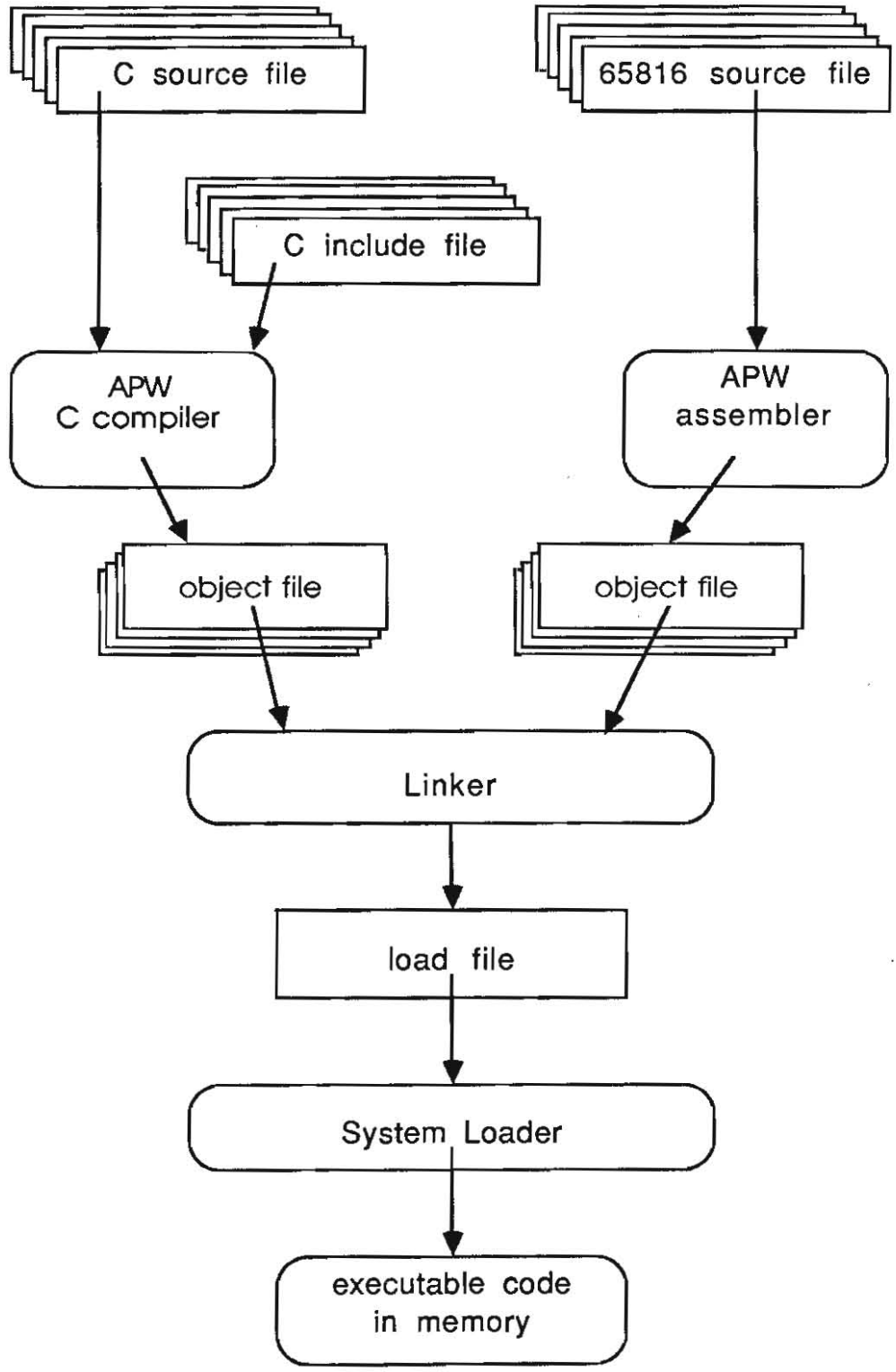


Figure 1.1. Creating an Executable C Program on the Apple IIGS

Program segmentation

In general, any computer program that consists of more than a few lines of code contains one or more subroutines; you may also choose to segregate large blocks of data into separate parts of the program.

In APW C, each subroutine (called a **function**) is translated into a segment in the source file: the function name is the segment name. As illustrated in Figure 1.2, when you compile the program, each source-code segment is translated into one **object segment**.

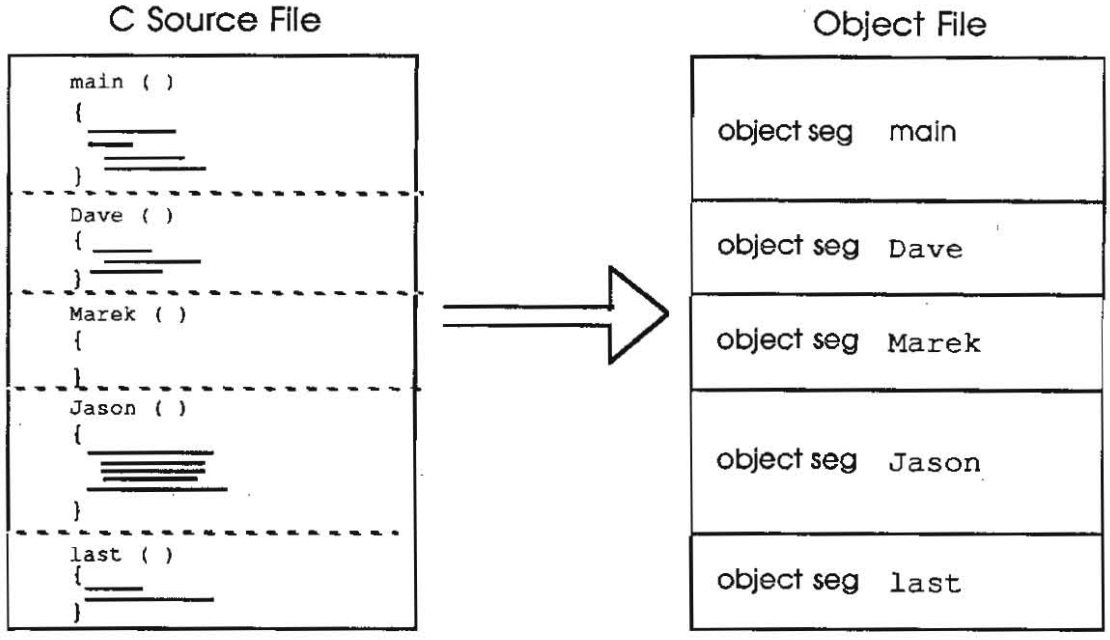


Figure 1.2. Creating Object Segments in Your Source Code

The object segment is the smallest linkable unit; for example, it can be selected from an object file for independent linking with a LinkEd command. It is also possible for a compiler to compile a segment (function) independently: a process called **partial compilation.**

Note: The APW C compiler does not perform partial compilation: if you request a partial compilation, the entire file will be compiled.

In addition to creating one code segment per function compiled, the APW C compiler also creates two **data segments** for each object file created (that is, for each source file compiled). These are used for storage of any global variables declared in the corresponding source file. Global scalar variables are stored in a segment called **`~globals`**, and global array and structure variables are stored in a segment called **`~arrays`**. Although this means that each file will therefore have the symbols `~arrays` and `~globals` defined, they are flagged as private symbols, meaning they can only be accessed from within the object module they are contained in. The symbols for the variables themselves contained with the segments, of course, are public. The compiler needs to generate two different data segments for the two different types of variables becauses it uses two different kinds of addressing—sixteen bit and twenty-four bit, respectively—to access them. The general implications of the code-generation memory

model are discussed in Chapter 4; the implications for use with the advanced linker are discussed in Chapter 4.

Apple IIGS load files are also segmented. Each load-file segment can incorporate any number of object-file segments. You can use a LinkEd command file to create load segments and to specify which object segments go in each load segment. Alternatively, APW C lets you specify load-segment names in the source code, by using the `segment` command. If you do not use a LinkEd file, all code segments with the same load-segment name are placed by the linker into the same load segment. The data segments ~globals and ~arrays are automatically identified as belonging to load segments of the same name; these must be collected into their own load segments so that the system loader can be assured of loading the ~globals segment within a single bank as required by the code-generation model, and so that the data segments can be re-loaded independently of the code when a program is re-started. Again, the linker does this automatically unless you use a LinkEd file to control your link. Use of source-file load-segment names are illustrated in Figure 1.3.

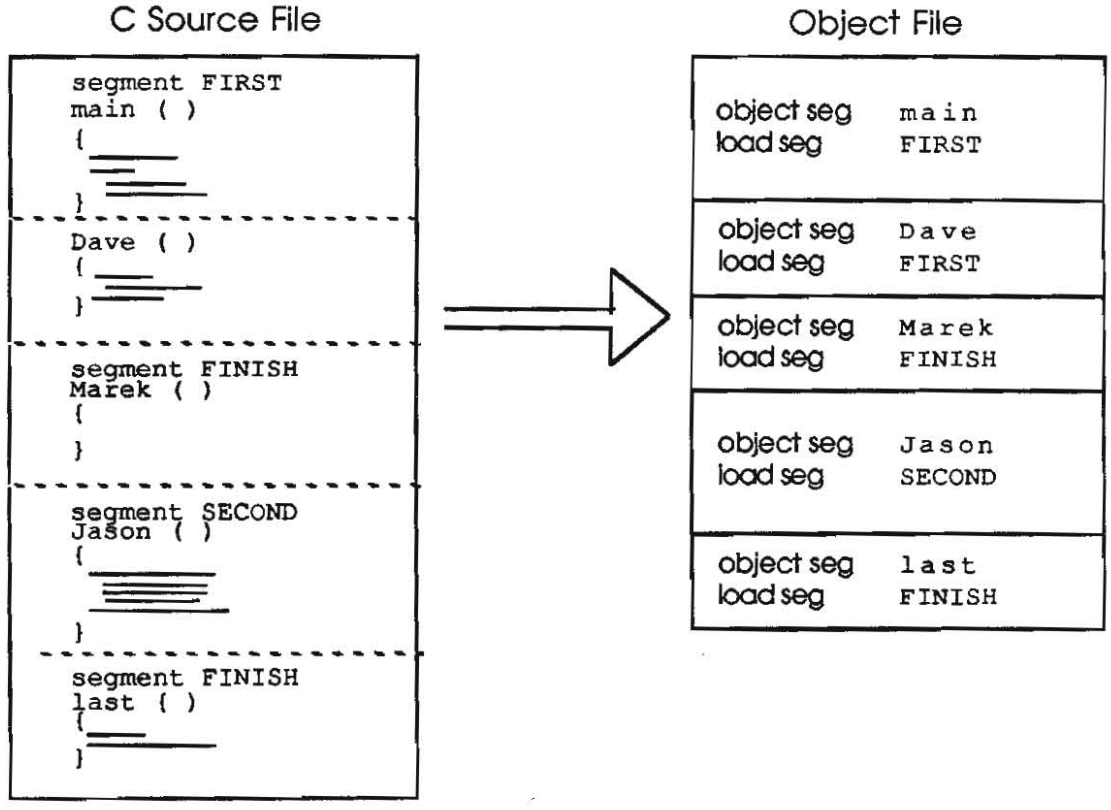


Figure 1.3. Assigning Load Segments in Your Source Code

The relationship of object segments to load segments is illustrated in Figure 1.4.

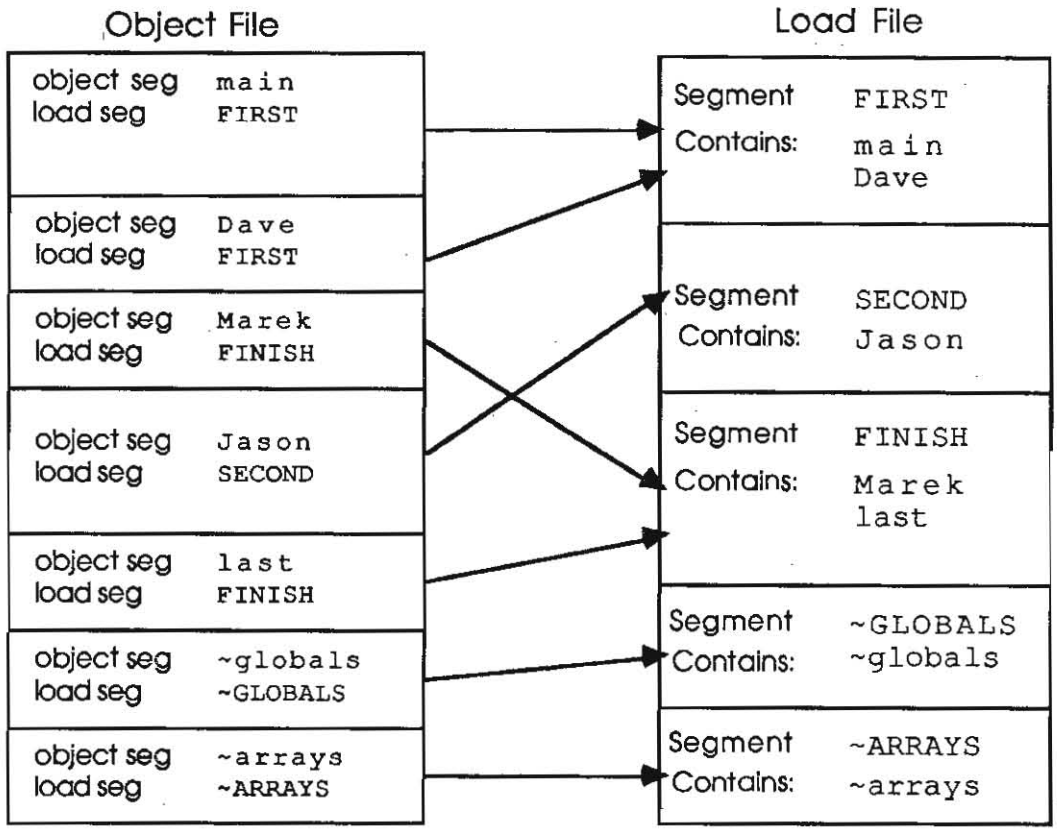


Figure 1.4. Relationship Between Object Segments and Load Segments

Every OMF file consists of one or more segments, each comprising a **segment header** and a **segment body**. The segment header is divided into fields described in the section "Segment Header" in Chapter 8 of the *APW Reference* .

The header of a load segment contains the name of the segment; the header of an object segment contains the name of the segment and the name of the load segment into which it goes. The name of the object segment is used by the linker in resolving function references; also, you specify the names of object segments when using the advanced linker to extract specific segments for linking (see the section "Using the Advanced Linker" in Chapter 5 of the *APW Reference*).

Each segment in a program must have a unique object-segment name: in APW C, each function is compiled to a separate object segment, whose name is the function name. Each object segment is also assigned a load-segment name. As illustrated in Figure 1.4, APW C lets you assign your own load-segment name to an object segment. Any number of object segments can have the same load-segment name. The standard linker places all object segments that share the same load-segment name into the same load segment (as long as they will fit into 64K).

For example, suppose your object file contains the following segments:

0. Object Segment Name: `main`
   Load Segment Name: `FIRST`

1. Object Segment Name: `Dave`
   Load Segment Name: `FIRST`
2. Object Segment Name: `Marek`
   Load Segment Name: `FINISH`
3. Object Segment Name: `Jason`
   Load Segment Name: `SECOND`
4. Object Segment Name: `last`
   Load Segment Name: `FINISH`
5. Object Segment Name: `~globals`
   Load Segment Name: `~globals`
6. Object Segment Name: `~arrays`
   Load Segment Name: `~arrays`

When the standard linker processes this file, object-segment names `main`, `Dave`, `Marek`, `Jason`, and `last` are treated as references that must be resolved. Object segments `main` and `Dave` are placed in the same load segment, named `FIRST`; object segments `Marek` and `last` are placed in the same load segment, named `FINISH`; and object segment `Jason` is placed in a separate load segment, named `SECOND`. Additionally, the object segment `~globals` is placed in the load segment `~globals`, and the object segment `~arrays` is placed in the load segment `~arrays`.

On the Apple IIGS computer, no single block of code can occupy more than 64 Kbytes of contiguous memory. To load a larger program than that, you must split it up into two or more load segments. When much of memory is already in use, it may be possible to load a program that is divided into several small load segments even if the same program in one or two load segments wouldn't fit. The Apple IIGS Memory Manager takes care of assigning each segment to a block of memory; the System Loader keeps track of where in memory the segment has been loaded, and patches intersegment calls in each segment as it is loaded.

Dynamic segments

On the Apple IIGS computer, the combination of load segments together with the System Loader and Memory Manager makes possible the creation of **dynamic segments**. A dynamic segment can be loaded automatically by the loader and Memory Manager during program execution simply by calling a function contained within the dynamic segment; if the segment is not currently in memory, the loader will load it automatically. A dynamic segment that is not needed at a given time can be removed, freeing the memory used to allow room in which to load another dynamic segment, or indeed, for any other purpose. Additionally, the loader and Memory Manager actually purge a dynamic segment from memory only if the memory is needed for something else; otherwise, the segment remains in memory and need not be reloaded the next time it is called, even if the user has "unloaded" it.

A segment that is not dynamic is **static**. A static segment is loaded at program boot time, and is not unloaded or moved during execution. The first segment of any program that is loaded is static; any other segments may be static, but (especially for large programs) the system will be more memory efficient if all infrequently-used segments are dynamic. These may make development of large applications for smaller memory configurations

practical. In order to specify that a load segment is dynamic, you must use a LinkEd command, or specify the **dynamic** option to the segment command.

Library files

Library files contain routines that are useful to many different programs. On the Apple IIGS, all library files are in object module format, regardless of the language of the source file. An Apple IIGS library file (ProDOS file type $B2) can therefore be used by a program written in any source language. Some languages, such as APW C, come with a set of library files used by that language. When the linker processes one or more object files and cannot resolve a symbolic reference, it assumes that it is a reference to a segment in a library file. If you use the standard linker, it automatically searches all the library files in the APW library prefix (`2/`). (If you use a LinkEd command file, then the advanced linker searches only the library files that you specify.) Unless you are using the advanced linker, you do not even need to know the names of the library files in order to use them: the standard linker automatically finds the files and extracts the segments it needs.

You can create your own library files from one or more object files by using the MakeLib APW utility program. Figure 1.5 illustrates the process of creating a library file. You specify one or more object files to be included in the library file. MakeLib concatenates the files and creates a special segment at the beginning of the file called the **library dictionary segment**. The library dictionary segment is the first segment of a library file; it contains the names and locations of all the **global symbols** in the file. (A global symbol is a label in one segment that can be referenced in another segment, as opposed to a **local symbol,** which can be used only within the segment in which it is defined.) The linker uses the library dictionary segment to find the segments it needs.

The library dictionary segment makes it possible for the linker to search a library file for global symbols much more rapidly than it can search an object file. Consequently, the linker will search a library dictionary segment multiple times if necessary to find segments referenced by other segments in the library file. The sequential order of the segments in a library file is therefore not important. If you were to use several library files, on the other hand, the order in which the files were searched *would* be important: if the linker first searched file `A` and then file `B`, for example, it could resolve a reference made in file `A` to a global symbol in file `B`, but could not resolve a reference made in file `B` to a symbol in file `A`. It is for that reason that MakeLib allows you to include several object files in a single library file.

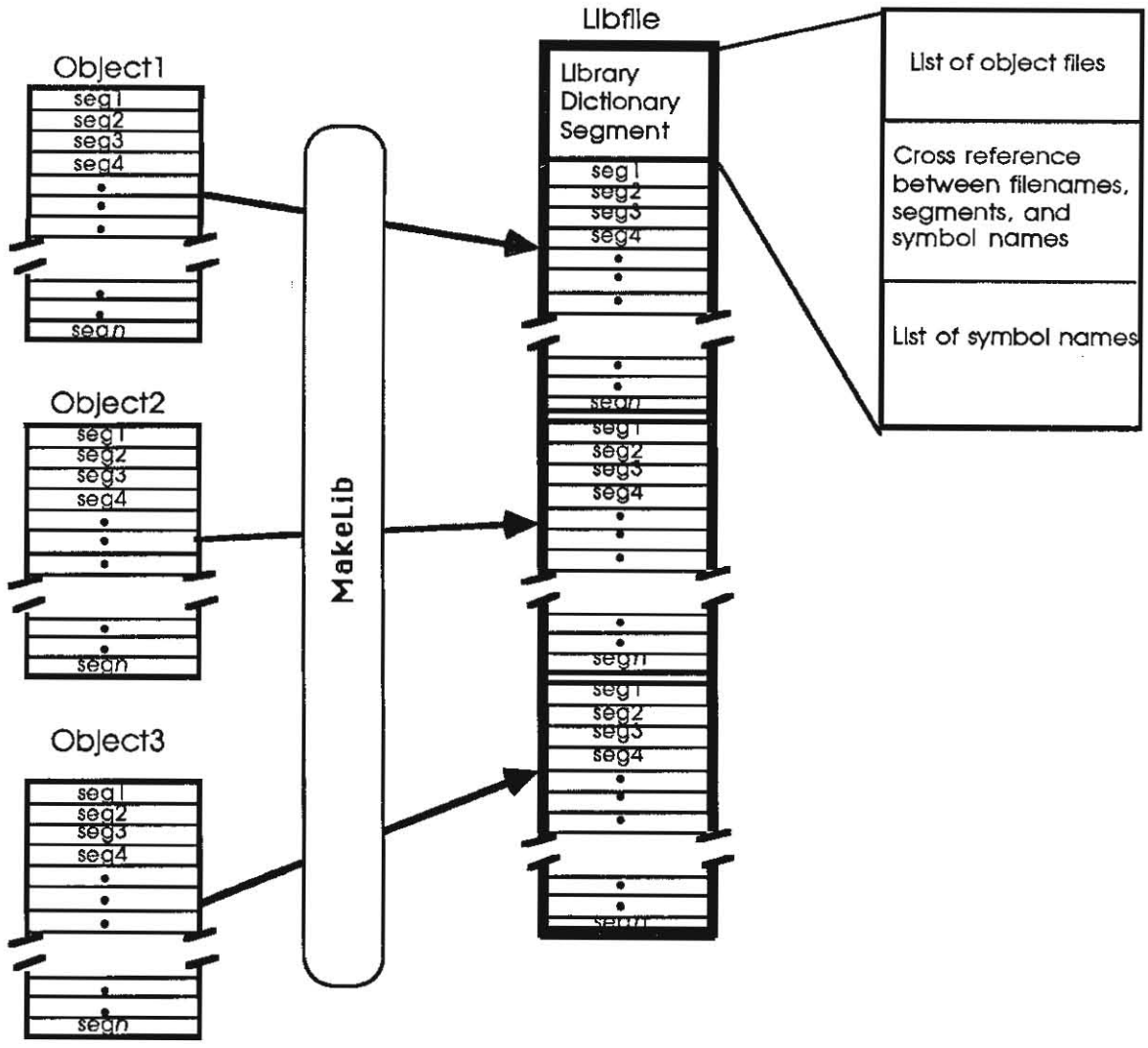


Figure 1.5. Relationship Between Object Files and Library Files

Program interactions

This section illustrates the interactions among the various programs in the Apple IIGS Programmer's Workshop by presenting a typical sequence of procedures and events. For this purpose, we assume that you are developing an application written mostly in C, with some routines written in 65816 assembly language. In this section, only the sequence of operations is listed; see Chapter 3 for an actual example of the sequence described here. The process described here is illustrated in Figure 1.6. See the *Apple IIGS ProDOS 16 Reference* manual for a complete description of the program load process.

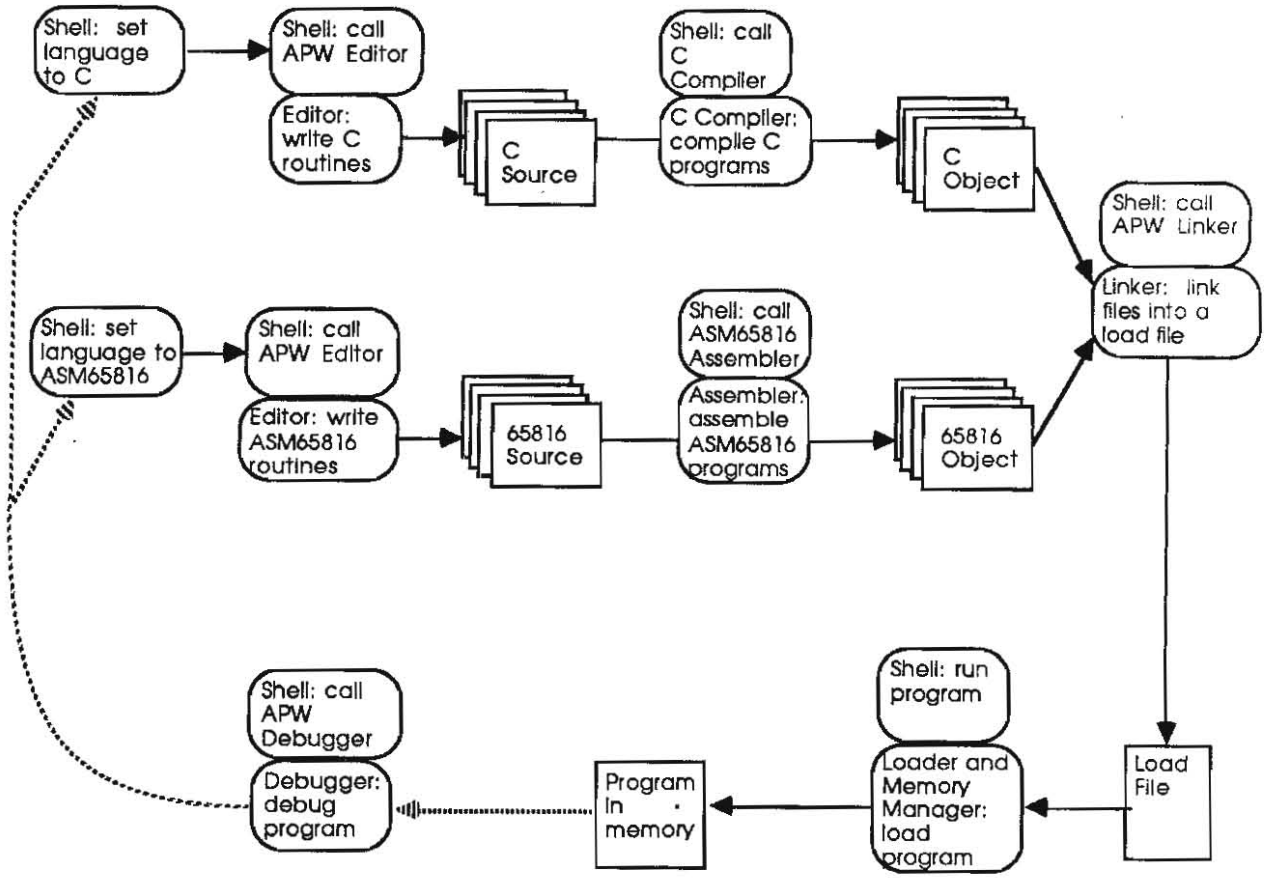


Figure 1.6. Program Interactions

1. Using an APW Shell command, set the **current language** for APW to `CC`. (Every APW file has an APW language type; if you open a new file, it is given the current APW language type.)
2. Call the APW Editor and open a new file.
3. Use the editor to write the C-language routines. You can divide the program among as many files as you wish. You do not have to return to the shell between files; you can save one file and open another within the editor. In APW C, you can use the `segment` command to specify which object segments go in which load segments. Until you use a shell command to change it, or open a non-C file, the current language remains `CC`.
4. Quit the editor, change the current language to `ASM65816`, call the editor, and open a new file. You can divide the 65816 assembly-language routines among as many files and as many segments per file as you wish. The APW Assember lets you specify which object segments go in which load segments. Make the assembly-language routines relocatable; that is, use no absolute addresses—use labels and relative addressing only.

   If you have used macros in your assembly language program, you can run the MacGen utility to generate a custom macro file for the program.

Until you use a shell command to change it, or open a non-assembly-language file, the current language remains ASM65816.

5. Quit the editor, call the APW Assembler to assemble the 65816 assembly-language routines, and call the APW C Compiler to compile the C routines. You can use the same command for both.

6a. Use the APW Linker to link the object files into a load file. Normally, you can use the standard linker to link the program. The standard linker places all object segments with the same load-segment name into a single load segment.

To compile and link the entire program in one operation, do the following:

a. Using the editor, tie all of your source files together by placing an `APPEND` directive (in assembly language) or a `#append` function (in C) at the end of each file but the last.

b. From the shell, execute the compile-and-link command (`CMPL`).

The shell checks the language type of the first file, and calls the C compiler. When the compiler gets to a 65816 file, it returns control to the shell, which calls the APW Assembler. When the assembler is finished, it returns control to the shell again, which calls the standard linker. The object files output from the C compiler and those output from the APW Assembler are all in the same format, and so are indistinguishable to the linker. The linker combines the object files, resolves references, writes the load file, and returns control to the shell.

6b. If you want to change load-segment assignments, or if you want to use dynamic load segments, you must use the advanced linker. Write a LinkEd file like a language source file: first set the system language to `LINKED`, then use the editor to write the file.

To compile and link the entire program in one operation, do the following:

a. Using the editor, tie all of your source files together by placing an `APPEND` directive (in assembly language) or a `#append` directive (in C) at the end of each file.

b. Put an `APPEND` or `#append` directive that references the LinkEd file at the end of the last file in the program.

c. In the shell, execute the `COMPILE` command.

The shell checks the language type of the first file, and calls the C compiler. When the compiler gets to a 65816 file, it returns control to the shell, which calls the assembler. When the assembler gets to the LinkEd file, it returns control to the shell again, which calls the advanced linker. The advanced linker, controlled by the commands in the LinkEd file, can do the following:

- combine the object files
- resolve references
- assign object segments to load segments
- label certain load segments as dynamic
- search libraries
- and write the load file.

When it is finished, the linker returns control to the shell.

7. Run the program by typing in the name of the load file and pressing the Return key. (You can also automatically execute a program after linking by using the `CMPLG` command.) When a program is run on the Apple IIGS, the following events occur:

a. The System Loader loads the first segment into memory (calling the Memory Manager to request the block of memory it needs). This segment is static; that is, it remains in memory during the execution of the program. The loader uses the relocation dictionary of the segment to relocate the code to its present location in memory.

b. The loader loads all other static segments into memory, relocating them as necessary.

c. The loader passes control of the system to the program, and the program begins to execute.

d. When a reference to a subroutine in a dynamic segment is encountered, control is returned to the System Loader through the jump table. If the segment is already in memory, the loader transfers control to the segment. If not, the loader uses the jump table to locate the load file, segment, and offset of the subroutine, loads the segment into memory, and transfers control to the segment. The System Loader creates and maintains a table (the **memory segment table**) to keep track of all the segments in memory.

8. If the program does not run correctly, you can use the APW Debugger to step through or trace the code, to insert breakpoints, to disassemble the machine code, and to examine the contents of registers and memory locations. You can modify the code in memory and rerun the program until the bug is fixed.
9. Correct the source code and recompile (or reassemble) the program.
10. Relink the program and rerun it.
11. When the program is completely debugged, you can use the CRUNCH command to compress the files created by partial assemblies into two object files, then link the program one last time. Using CRUNCH is optional: if you have performed several partial assemblies, compressing the object files speeds up the link process.

Using the APW C libraries

APW C programs can use the Standard C Library, The Apple IIgs Toolbox, the APW Shell, and ProDOS to talk to the Apple IIgs hardware. All of the interface code to make these calls is contained in the file CLIB which is installed in the APW library prefix (/2). (Any header files containing declarations needed needed to make the calls are installed in the CINCLUDES directory in the library prefix.) Figure 1-7 shows how these libraries interact. Your application can make calls to the Standard C Library, the APW Shell, the Apple IIgs Toolbox, or ProDOS. The Standard C Library contains a number of high-level routines familiar to C programmers, which deal with file handling, memory management, and so on. The Standard C Library in turn calls the Toolbox or ProDOS. You can also make calls to the APW Shell. The shell intercepts the call: if it is a ProDOS call, the shell passes it through unchanged; if it is a shell call, the shell makes ProDOS calls, or talks to the hardware directly, to execute it.

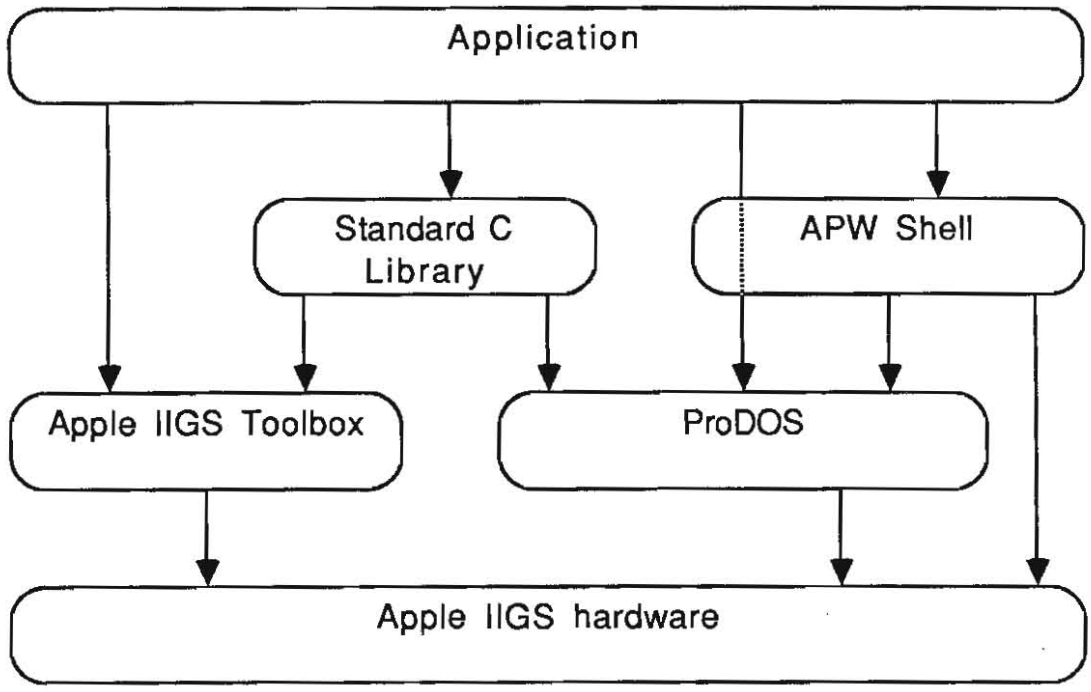


Figure 1-7. APW C Library Interactions

Chapter 2
Using the APW C Compiler

This chapter describes how to use the APW C compiler. The first section, "Installing APW C," tells you how to install APW C on your system. The second section, "Writing and running a sample program," leads you through a sample session, giving you a fast way to become acquainted with compiling, linking, and executing a program. The third section, "The APW C Compiler," discusses the compilation process. The fourth section, "C Compiler Shell Commands," describes the Shell commands you'll use when working with the C compiler. The fifth section, "Source Files, Object Files, and Listing Files," tells how to use the various files used in building a program.

Installing APW C

Before you can do any of the things described in this chapter, you must install APW C. First install APW and then install C, as described below:

Installing APW on a Hard Disk

Before doing anything else, make a backup copy of your APW disk and put the original in a vault.

We will assume your hard disk is called *harddisk*. First, insert the APW disk in a drive and start up APW, then type these commands:

```
COPY -C /APW/=  /harddisk/APW/
```

```
COPY -C /APW/SYSTEM/=  /harddisk/SYSTEM/
```

These steps take several minutes, as they involve copying hundreds of files.

Now insert the most recent Apple IIGS System Disk and type

```
COPY -C /SYSTEM.DISK/=  /harddisk/
```

This will take a few minutes.

When you are done, you have APW installed. It will work fine, but several files are duplicated: the copies in the directory */harddisk*/APW will never be used. You can save space by typing these commands:

```
DELETE /harddisk/APW/PRODOS
```

```
DELETE /harddisk/APW/SYSTEM/P16
```

```
DELETE -C /harddisk/APW/SYSTEM/SYSTEM.SETUP/=
```

```
DELETE /harddisk/APW/SYSTEM/SYSTEM.SETUP

DELETE -C /harddisk/APW/SYSTEM/DESK.ACCS/=

DELETE /harddisk/APW/SYSTEM/DESK.ACCS

DELETE -C /harddisk/APW/SYSTEM/TOOLS/=

DELETE /harddisk/APW/SYSTEM/TOOLS
```

Installing APW C on a Hard Disk

Before doing anything else, make a backup copy of your APW C disk and put the original in a vault. We will assume you have already installed APW in a directory called APW on a disk called *harddisk*.

First, launch APW from your hard disk. To install APW C, type these commands:

```
COPY -C /APWC/LANGUAGES/=  /harddisk/APW/LANGUAGES/

COPY -C /APWC/LIBRARIES/=  /harddisk/APW/LIBRARIES/
```

These steps also take some time.

Next, add this command to your LOGIN file:

```
CC
```

Installing APW C on two 3.5-inch disks

Before doing anything else, make a backup copy of your APW and APW C disks and put the originals in a vault. To install C, you will have to replace some of the files on the APW disk, and delete files not needed for C. (Two 3.5-inch disks will not hold all the files you need to program in both assembly language and C.) Type these commands:

```
COPY -C /APW/LANGUAGES/LINKED  /APWC/LANGUAGES/

DELETE -C /APW/LANGUAGES/=

DELETE /APW/LANGUAGES/

COPY -C /APW/LIBRARIES/=  /APWC/LIBRARIES/

DELETE -C /APW/LIBRARIES/=

DELETE /APW/LIBRARIES/
```

Next, add these commands to your LOGIN file:

```
PREFIX 2 /APWC/LIBRARIES/
```

```
PREFIX 5 /APWC/LANGUAGES/

CC
```

Writing and running a sample program

Here is how to write, compile, link, and run a trivial sample program.

Writing the sample program

First set the current language to C by typing `CC` and pressing Return. Now create a new file named `SHE.SELLS` by typing `EDIT SHE.SELLS` and pressing Return. You are now in the APW editor, so type a program: for example,

```
main()
{
      printf("She sells C shells by the C shore.\n");
}
```

Now press Apple-Q and then `S` to save the program, then press E et exit the editor.

Note that APW does not require the usual C filename extension "`.c`", because APW uses a unique file type for source files of each language. You can end a filename with "`.c`", but the APW C compiler regards the "`.c`" as part of the name, rather than as an extension. In particular, when forming an object filenames, the compiler appends an extension to the "`.c`", rather than replacing it. Using "`.c`" on a source filename can be confusing, as some object filenames have a "`.c`" extension.

Compiling and linking the sample program

To compile your program, use the `COMPILE` command; to compile and link it, use the CMPL command. This command takes the source file and load file names (KEEP) as arguments: they must be different, or your load file will overwrite your source file.

For example, to compile and link `SHE.SELLS`, creating an object file `C.SHELLS.ROOT` and a load file `C.SHELLS`, type the following, and then press Return:

```
CMPL SHE.SELLS KEEP=C.SHELLS
```

Note: If you get the error message `ProDOS: File not found`, make sure you've typed the command correctly. If you had typed `COMPL` rather than `CMPL`, for example, the APW shell would give you this message, because it knew no command named `COMPL` and couldn't find any file of that name. You could spend hours hunting for missing libraries and include files (described in the section "Files for Compiling and Linking" at the end of this chapter), when the real problem was a misspelled command.

Running the sample program

To run your program under the APW shell, type `C.SHELLS` and press Return. You will see

```
She sells C shells by the C shore.
```

on the screen.

A longer sample program

A more interesting sample program, written in both C and assembly-language, is in Chapter 3.

The APW C compiler

This section discusses the compilation process, how compilation is suspended or aborted, and error messages.

The compilation process

The APW C compiler is a one-pass compiler. In one pass, it resolves preprocessor macros, scans the source files, and generates code into a code buffer, and then writes the code out to an object file. Each C function is assigned to a separate object segment: the object-segment name is the function name. The default load-segment name is `MAIN`.

The `segment` command can be used to assign an object segment or group of segments to a load segment. The command

```
segment "segname"[, dynamic]
```

assigns all objects following it, up to the next `segment` command or the end of file, to the load segment named *segname*. (Note that the quotation marks are required.) By default, this command creates a static load segment. The `dynamic` option creates a dynamic segment.

No listing is printed. The compiler prints errors to the screen.

Object code output is in object module format (OMF). Each APW language outputs object code in object module format, allowing you to link together subroutines written in different languages. Object module format is discussed in detail in Chapter 8, "File Formats," of the *Apple IIGS Programmer's Workshop Reference.*

If there are no more subroutines to compile, the C compiler returns control to the Shell. Depending on the command you used to invoke the C compiler, the Shell either passes control to the linker or returns with the Shell prompt. If the linker is called, it uses the object modules produced by the C compiler as input. These are relocated and **global labels** are resolved, giving an executable binary file as output.

Suspending or aborting the compilation

You can suspend the compilation by pressing any key; pressing any key again causes compilation to resume. Note that you can suspend the compilation only while error messages are being printed. To abort the compilation, press Apple-period.

C compiler error messages

If the C compiler detects an error in the source code, an error message is printed on the screen: each error message includes the source file name, the line number, and the text of the offending line of code. In other cases, the compiler will print a warning message rather than an error. Error messages can be redirected, as explained in the section "Redirecting Input and Output" in Chapter 3 of the *APW Reference*. If no errors or warnings are detected, the compiler runs without comment.

C compiler shell commands

This section discusses the commands you'll use most often when working with the C compiler. With these commands, you can perform the following tasks:

- Edit new and existing files
- Compile, link and execute your program
- Make a library file
- Debug your program

Editing a source file

You will need three shell commands when you edit a new or existing source file:

`CC`	Change the default language to C
`EDIT` *filename*	Edit an new or existing file
`CHANGE` *filename* `CC`	Change the type of an existing file to C source file

The `CC` command sets the default language to C: any new files you create with the editor will automatically get the appropriate type for a C source file. The `EDIT` command edits an existing file or creates a new file. The `CHANGE` command changes the type of a file from one language to another: this is useful if you have imported an ASCII file from some other implementation of C, such as MPW, and the file type is not set for APW C, or if you had edited a C source file what the default language was not C.

Compiling a program

You'll need five commands when compiling, linking, and running your program:

COMPILE	Compile a program
CMPL	Compile and link a program
CMPLG	Compile, link, and execute a program
RUN	Compile, link, and execute a program
LINK	Link a program

In its simplest form, the COMPILE command compiles the source file, but saves no object file: it simply verifies the program's correctness. To create an object file, use the KEEP option or the KEEPNAME shell variable, both described below.

The COMPILE command is a synonym of the ASSEMBLE command. Either of these commands can be used interchangeably to compile or assemble programs. Similarly, CMPL is a synonym of ASML and CMPLG and RUN are synonyms of ASMLG. Synonymous commands have the same options, but one language processor may ignore options that another recognizes. For example, the C compiler ignores the +L|-L and +S|-S options.

Note: The CMPL, CMPLG, and RUN commands cannot be used if you're developing a program whose main entry point is not written in C.

Command Notation

The following notation is used to describe commands:

UPPERCASE	Uppercase letters indicate a command name or an option that must be spelled exactly as shown. The Shell is not case sensitive; that is, you can enter commands in any combination of uppercase and lowercase letters.
italics	Italics indicate a variable, such as a filename or address.
prefix	This parameter indicates any valid directory pathname or partial pathname. It does *not* include a filename. If the volume name is included, *prefix* must start with a slash (/); if *prefix* does not start with a slash, then the current prefix is assumed. For example, if you are copying a file to the subdirectory SUBDIRECTORY on the volume VOLUME, then the *prefix* parameter would be /VOLUME/SUBDIRECTORY/. If the current prefix were /VOLUME/, then you could use SUBDIRECTORY for *pathname* . The device numbers .D1, .D2, ...D*n* can be used for volume names; if you use a device number, do not precede it with a slash. For example, if the volume VOLUME in the above example were in disk drive .D1, then you could enter the *prefix* parameter as .D1/SUBDIRECTORY/.
filename	This parameter indicates a filename, *not* including the prefix. The unit names .CONSOLE and .PRINTER can be used as filenames.
pathname	This parameter indicates a full pathname, including the prefix and filename, or a partial pathename, in which the current prefix is assumed. For example, if a file is named FILE in the subdirectory DIRECTORY on the volume VOLUME, then the *pathname* parameter would be

/VOLUME/DIRECTORY/FILE. If the current prefix were /VOLUME/, then you could use DIRECTORY/FILE for *pathname* . A full pathname (including the volume name) must begin with a slash (/); do *not* precede *pathname* with a slash if you are using a partial pathname.

The unit names .CONSOLE and .PRINTER can be used as filenames; the device numbers .D1, .D2, ...D*n* can be used for volume names.

| | A vertical bar indicates a choice. For example, +L\|-L indicates that the command can be entered as either +L or as -L.

[ ] Parameters enclosed in square brackets are optional.

... Elipses indicate that a parameter or sequence of parameters can be repeated as many times as you wish.

The following pointers will help you use the APW Shell command interpreter:

- You must separate the command from its parameters by one or more blanks.
- You can use the right-arrow key to expand command names as described in the "Entering Commands" section in Chapter 2 of the *APW Reference,* you can use the Up- and Down-Arrow keys to scroll through previously-entered commands.
- There are no abbreviations for command names (unless you add aliases to the SYSCMND file).
- All commands and parameters (except for segment names) can be entered in any combination of uppercase and lowercase characters.
- For case-sensitive source languages, like C, segment names must be entered exactly as they appear in the source code.
- When a parameter in a command line conflicts with a source-code command, the command-line parameter takes precedence. When neither a source-code command nor a command-line parameter has been used, the default parameter is used.
- If you fail to enter a required parameter, you are prompted for it.
- Any of these commands can be placed in an Exec command file for automatic execution; Exec files are described in the section "Exec Files" in Chapter 3 of the *APW Reference*.

The APW Shell and C compiler recognize the following commands. The options for each command are described below it.

Cc

This command sets the shell default language to APW C. Any file created by the APW editor while this command is in effect will have the file type identifying it to APW as a C

source file. (This command is described in the section "Command Descriptions" in Chapter 3 of the *APW Reference*.)

CHANGE

`CHANGE` *filename* `CC` Change the type of an existing file to C source file

This command changes the file type of an existing file named *filename* so that APW will recognize it as a C source file. It is useful when you have imported a C source file from another development system, such as MPW, that does not identify the language of a source file by a unique file type. (This command is described in the section "Command Descriptions" in Chapter 3 of the *APW Reference*.)

CMPL

`CMPL [+L|-L] [+S|-S]` *file1* *[file2...]* `[KEEP=`*outfile*`] [NAMES=(`*seg1*`[ `*seg2*`[ ...]])]`
[language1=(option ...) [language2=(option ...) ...]]

This command compiles (or assembles) and links a source file or group of files. Its function is identical to that of the `ASML` command. The APW Shell checks the language of the source file and calls the appropriate compiler or assembler. If the maximum error level returned by the compiler or assembler is less than or equal to the maximum allowed (0 by default), then the resulting object module is linked, producing a load module with the filename *outfile*. The linker is described in Chapter 5 of the *APW Reference*.

Note: The commands `CMPL`, `CMPLG`, `ASML`, `ASMLG`, and `RUN` cannot be used if you're developing a program whose main entry point is not written in C. In this situation, you must use `COMPILE` or `ASSEMBLE`, then `LINK`.

The options peculiar to APW C are described fully below. The other options are described briefly: they are described fully in Chapter 3 of the APW Reference.

Important: If you are using a LinkEd file to take advantage of the advanced link-edit capabilities it provides, do *not* use the `CMPL` command. Instead, use the `COMPILE` command to compile your program. You can process the LinkEd file automatically by appending it to the end of your program with an `#append` directive (or the equivalent), or you can process it independently with the ALINK command.

> **Note:** You can use `#append` directives (or the equivalent) to tie together source files written in different computer languages; APW compilers and assemblers check the language type of each file and return control to the Shell when a different language must be called. See the section "Compiling (or Assembling) and Linking a Program" in Chapter 2 of the *APW Reference* for a description of the assembly and compilation process.

`+L|-L` (The APW C compiler ignores this option.)

`+S|-S` (The APW C compiler ignores this option.)

file1 file2... The full pathname or partial pathname (including the filename) of the source files to be compiled (or assembled). Multiple files (source, object, or library) can be listed, but at least one must be a source file.

KEEP=*outfile* You can use this parameter to specify the pathname or partial pathname (including the filename) of the object file to be produced. If this is a partial assembly or if several source files with different programming languages are being compiled, then other filename extensions may be used; see the section "Partial Assemblies or Compiles" in Chapter 3 of the *APW Reference*. If the assembly is followed by a successful link, then the load file is named *outfile*.

Important: Keep the following points in mind regarding the KEEP parameter:

- If you use neither the KEEP parameter nor the KeepNames shell variable, the object modules are not saved at all. In this case, the link cannot be performed, because there is no object module to link.
- The filename you specify in *outfile* must not be over 10 characters long. This is because the extension .ROOT is appended to the name, and ProDOS 16 does not allow filenames longer than 15 characters.
- If a load file named *outfile* or an object file with root filename *outfile* already exists, it is overwritten without a warning when this command is executed. If a source file named *outfile* exists, it will not be overwritten: the link will fail.

NAMES=(*seg1 seg2* ...) (The APW C compiler always compiles the whole source file.)

language1=(*option* ...) ... This parameter allows you to pass parameters directly to specific APW compilers or assemblers. For each compiler or assembler for which you want to specify options, type the name of the language (exactly as defined in the Command Table), an equal sign (=), and the string of options enclosed in parentheses. The contents and syntax of the options string is specified in the compiler or assembler reference manual; the APW Shell does no error checking on this string, but passes it through to the compiler or assembler. You can include option strings in the command line for as many languages as you wish; if that language compiler is not called, then the string is ignored.

Note: No blanks are permitted immediately before or after the equal sign in this parameter.

CC=(*option* ...) This is a special case of the *language1*=(*option* ...) option, defined above. This option's options are as follows:

-D*name*=*value* This parameter defines *name* as if a #define had occurred at the top of the file; *name* is given the value *value* if "=*value*" is present.

-I*path* This parameter adds *path* to the include-file path list. For example:

-I/APW/LIBRARIES/CINCLUDE/

Listings and error messages are sent to the screen unless you include a PRINTER ON directive (or equivalent) in the source file; or redirect output to a disk file or the printer.

Output redirection is described in the section "Redirecting Input and Output" in Chapter 3 of the *APW Reference*.

CMPLG

CMPLG [+L|-L] [+S|-S] *file1* [*file2*...] [KEEP=*outfile*] [NAMES=(*seg1*[*seg2*[...]])] [*language1*=(*option* ...) [*language2*=(*option* ...) ...]]

This internal command compiles (or assembles), links, and runs a source file or group of files. Its function is identical to that of the ASMLG command. See the CMPL command for a description of the parameters.

Note: The commands CMPL, CMPLG, ASML, ASMLG, and RUN cannot be used if you're developing a program whose main entry point is not written in C. In this situation, you must use COMPILE or ASSEMBLE, then LINK.

COMPILE

COMPILE [+L|-L] [+S|-S] *file1* [*file2*...] [KEEP=*outfile*] [NAMES=(*seg1*[*seg2*[...]])] [*language1*=(*option* ...) [*language2*=(*option* ...) ...]]

This internal command compiles a source file or group of files. Its function is identical to that of the ASSEMBLE command. See the CMPL command for a description of the parameters.

EDIT

EDIT *filename*

This command does one of two things. If a file named *filename* already exists, the command EDIT *filename* calls the editor and opens the file *filename*. The editor uses the language the file is already in. If a file named *filename* does not already exist, the command EDIT *filename* calls the editor and a new file called *filename*. The editor uses the default language established by the last language command (CC, ASM65816, or whatever).

LINK

LINK [+L|-L] [+S|-S] 2/START*file1* [*file2*...] [KEEP=*outfile*] [NAMES=(*seg1*[*seg2*[...]])] [*language1*=(*option* ...) [*language2*=(*option* ...) ...]]

This command links an object file or group of files. If you use a COMPILE command followed by a LINK command and if your main entry point is written in C, you must

include the pathname `2/START` in the `LINK` command. (The linker is described in Chapter 5 of the *APW Reference.)*

RUN

`RUN [+L|-L] [+S|-S]` *file1* *[file2...]* `[KEEP=`*outfile*`] [NAMES=(`*seg1*`[ `*seg2*`[ ...]]) ]`
[language1=(option ...) [language2=(option ...) ...]]

This internal command compiles (or assembles), links, and runs a source file or group of files. Its function is identical to that of the `CMPLG` command. See the `COMPILE` command for a description of the parameters.

Note: The commands `CMPL`, `CMPLG`, `ASML`, `ASMLG`, and `RUN` cannot be used if you're developing a program whose main entry point is not written in C. In this situation, you must use `COMPILE` or `ASSEMBLE`, then `LINK`.

Examples of these commands

The following command compiles and links a source file named `MYFILE`, and writes the load file to disk as the file `MYPROG`. No source listing or symbol table is produced unless called for by directives in `MYFILE`:

```
CMPL MYFILE KEEP=MYPROG
```

The following command compiles the segments `TOOLCALL` and `TEXT_OUT` in the source file named `MYFILE`, links the program, and writes the load file to disk as the file `MYPROG`.

```
CMPL MYFILE KEEP=MYPROG NAMES=(TOOLCALL TEXT_OUT)
```

The following command compiles the source file named `MYCFILE`.

```
CMPL MYCFILE KEEP=MYPROG CC=(-Ddebug -I/APW/MYINCLUDES)
```

Because `MYCFILE` is a C program, two C-compiler options are passed to the C compiler: the `-Ddebug` option defines a compiler flag that you can use to conditionally compile debugging code; and the `-I/APW/MYINCLUDES` option tells the compiler where to search for additional include files. After the program is assembled or compiled, it is linked and the load file is written to disk as the file `MYPROG`.

> **Note:** The `ASML`, `ASMLG`, `CMPL`, and `CMPLG` commands first assemble or compile the source file (or files), then send the object file specified in the `KEEP` parameter (or in a `KEEP` directive in the source file) to the linker as its only input. These commands cannot be used to send several object files with different root filenames to the linker. To link two or more object files, use the `LINK` command.

Appending files

When APW sees a `#append` directive in a file, it checks the language type of the appended file: if it is not CC, the compiler returns control to the shell, which brings in the appropriate compiler or assembler to open the file. If the appended file is in the same language, the effect is the same as if they had been concatenated into one file. If they are in different languages, APW begins a new assembly or compilation. This has curious effects, as we'll show.

Let's take three files, two in C and one in assembly language, each appended to the preceding file:

```
c1
c2
asm1
```

When you use the `COMPILE` command, `c1` and `c2` will be compiled together, then asm1 will be assembled. All symbols in `c1` will be available while `c2` is being compiled.

Something very different happens when we compile the same files, appended in a different order:

```
c1
asm1
c2
```

When you use the `COMPILE` command, `c1` is compiled, then `asm1` is assembled, then the C compiler is called afresh to compile `c2`. Since the compilations were separate, the compiler knows nothing about symbols in `c1` when compiling `c2`.

Partial compilation or assembly

Program development can often be speeded by compiling or assembling only the part of a program that you have changed most recently. The APW assembler has an option NAMES (to the `ASSEMBLE`, `ASML`, `ASMLG`, `COMPILE`, `CMPL`, `CMPLG`, and `RUN` commands) that lets you do partial assemblies, and future APW compilers may also support this option. APW C does not support partial compilation. The compiler will execute a `COMPILE` command with the `NAMES` option, but it will compile the entire source file, as if you had omitted the `NAMES` option.

The linker

The linker takes object files and file segments created by the C compiler and generates load files. The linker resolves external references and creates relocation dictionaries which allow the system loader to relocate code at load time. The linker supports **data, code, dynamic,** and **static segments**, and library files.

Normally, the linker is called by the Shell command `LINK` which provides a limited number of options. Additionally, you can control all functions of the linker by using a language-like set of commands called **LinkEd**. LinkEd is for advanced programmers who require maximum flexibility from the system; for most purposes, the ordinary Link commands are adequate. LinkEd commands are described in Chapter 5 of the *APW Reference;* other APW commands are in Chapter 5 of that book.

When you use CMPL to compile and link a series of files in different languages, the last file in the append sequence must be a C file. The files under the library prefix (prefix 2) are searched for unresolved references.

To link manually and search all libraries, use this command:

LINK 2/START *objectfilename* KEEP=*loadfilename*

The *objectfilename* parameters do not have .ROOT extensions. For example, the command

```
LINK 2/START FILE1 FILE2 FILE KEEP=LOADNAME
```

links the files FILE1.ROOT, FILE2.ROOT, and FILE3.ROOT with the file 2/START.ROOT.

The linker searches every library (file of filetype LIB) in the library prefix (/2).

Making a library

The MAKELIB utility allows you to make a **library file.** Libraries are useful for storing often-used code, as the linker can search a library much faster that an ordinary object file. The *APW Reference* explains how to use MAKELIB.

Files for compiling and linking

To create a program from source files, the compiler usually needs include files and the linker usually needs libraries. Include files, or header files, must be named in #include statements in the source files. Library files are either searched implicitly or can be named in LINK statements or in LinkEd files.

Include-file search rules

Appendix B, "Files Supplied with APW C," contains a list of include files to be used with APW C. If the include-file name is a full pathname, the compiler uses that name. A full pathname begins with a slash (/) and contains at least one embedded slash. A partial pathname does not begin with a slash. (For more information about pathname syntax, refer to the *Apple IIGS Programmer's Workshop Reference* and the *Apple IIGS ProDOS 16 Reference.)*

If the include-file name is a partial pathname, the compiler searches for include files using the rules shown in Table 2-2. The first file successfully opened using these rules is included.

Table 2-2. Include-file search rules

Include-File Name	Example	Search for Partial Pathname
In double quotes.	"CONSTANTS.H"	Look in the following directories:

		(1) The directory of the source file that contains the include statement.
		(2) The current prefix (`0/`) at the time the compiler was invoked.
		(3) Directories specified by the `-I` option, in the order given.
		(4) 2/CINCLUDE/
In angle brackets.	`<CTYPE.H>`	Look in the directories described under (3), or (4) if there is no `-I` option.

Note that ProDOS filenames are not case-sensitive. By convention, filenames and pathnames are notated in uppercase.

Library files

Appendix B, "Files Supplied with APW C," contains a list of library files to be used with C. (If you use the `CMPL` or `CMPLG` command, the files under the library prefix are searched, and you can't specify any others). For more information on linking C programs, refer to Chapter 5, "The Linker" of the *APW Reference*.

You can control which library files are to be searched by using a LinkEd script. If you specify library files, you will usually want to specify

- all the Standard C Library files listed in Appendix B
- only the particular Toolbox files you refer to in your program.

Chapter 3
Sample Program

This chapter provides a tutorial example that illustrates the creation of a program in the APW environment. The program includes a main routine in C and a subroutine in assembly language. You are shown how to use the APW Editor to create source files in both languages, as well as how to compile, assemble, link, and run the program.

The purpose of this chapter is to give you a tutorial introduction to compiling and linking a simple multilanguage program in the APW environment. This example is placed in the *APW C Reference,* rather than in the *APW Reference,* because both APW and APW C are needed to run the example, and only owners of APW C can be assumed to have both.

> **Note:** The instructions in this chapter assume that you have both the APW Assembler and the APW C compiler installed in your system. Assembly language is included on your APW disks; the C compiler is on the APW C disk. See Chapter 2 for instructions on installing APW and APW C in your system.
>
> If you have a hard disk, the instructions in this chapter are straightforward. If you have two 3.5" drives, you may have to do some disk swapping, and tweaking of prefixes, to follow these instructions.

General procedure

This section describes the general procedure that we follow in this chapter. A simpler procedure for compiling, linking, and running a single-language program is given in the section "Writing and Running a Simple Program" in Chapter 2.

> **Note:** For simplicity's sake, the words *compiler* and *compile* are used in this chapter to include *assembler* and *assemble*.

1. Set the system language to the language type of the source code you intend to write, open a file for editing, and write the source code for the first part of your program. Save the file to disk.
2. Execute the shell `COMPILE` (or `ASSEMBLE`) command.

   You now have several files on disk: the source-code file and one or more object-code files (the root file and files with alphabetic extensions such as `.A`).
4. Write the next part of the program. This part need not be in the same programming language as the first part. Give this part a different source filename than the first part and a different `KEEP` filename.
5. Execute the shell `COMPILE` command. Debug the program and recompile as necessary until successful.
6. Repeat steps 4 and 5 for each part of the program, until you are sure that each part compiles successfully.
7. Execute the `LINK` command, specifying the root filenames of all of the object files in the program.

8. If you wish, execute the COMPACT command to create a more compact version of the load file.

If you prefer, you can write the entire program, including parts in several languages, and compile and link them all at once. Use the CMPL command to compile and link the program. Each source file except the last can end in an APPEND directive (or the equivalent), or you can specify multiple source files in the CMPL command. Every time an APW compiler executes an APPEND directive, it checks the APW language type of the file being appended. If the language doesn't match that of the compiler, then the compiler returns control to the shell, which calls the appropriate compiler to continue processing the program. If all compiles are successful, the APW Linker is called automatically. The linker processes the file, writes out any errors, and (if the link was successful), writes the load file to disk.

> **Note:** The compiler may check the language type of a file when executing a COPY directive, but does not return control to the shell; instead, the compiler returns an error if any file being copied into the program does not match the language of the compiler.

Writing and editing the sample source code

The sample program shown in Figures 3.1 and 3.2 takes input from the keyboard, converts every letter to uppercase, and prints the result to the screen. It is written with a main segment in C and a subroutine in assembly language. The C routine handles the input and output. The assembly language routine does the lowercase to uppercase conversion.

Use the following steps to write the source code for the C routine shown in Figures 3.1 and 3.2:

1. Boot APW and type the following command to set the system default language (the **current language**) to C. (To execute an APW command, press the Return key.)

   ```
   CC
   ```

2. Call the editor to open a file called SAMPLEC with the following command:

   ```
   EDIT SAMPLEC
   ```

3. Type in the program in Figure 3.1. Use the cursor keys to move around in the file. The Delete key deletes the character to the left of the cursor. The Tab key moves the cursor for indenting subroutines. Other basic editor commands are given in Table 2.3.

4. Press ⌘-Q to quit the editor. Press S to save the file to disk, then press E to exit the editor and return to the shell.

```
/* Convert all characters taken from standard input to uppercase */
/*  and write the result to standard output.                     */
/* ...                                                           */
/* NOTE: Control-C terminates the input                          */

#include <stdio.h>
#define MAXLEN 1024
extern void UPSTR();
char *gets();


main(argc, argv)
int argc;
char *argv[];
{
       char str[MAXLEN];
       while  (gets(str) != NULL)
                     {
                          UPSTR(str);
                          printf("%s\n", str);
                     }
}
```

Figure 3.1. Sample C Source Code

5. Type the following command to set the current language to 65816 assembler.

   ```
   ASM65816
   ```

6. Call the editor to open file called SAMPLEA with the following command:

   ```
   EDIT SAMPLEA
   ```

7. Type in the program in Figure 3.2. Note that the default tab stops are different for assembly language than for C.

8. Press ⌘-Q to quit the editor. Press S to save the file to disk, then press E to exit the editor and return to the shell.

```
          LONGA ON       Set long accumulator
          LONGI ON       Set long index registers
UPSTR     START          Start first object segment
;                        Stack at beginning of routine:
;
;                          Bank of pointer to character string
;                          High byte of character-string address
;                          Low byte of character-string address
;                          Bank of return address
;                          High byte of return address
;                          Low byte of return address
;                          Stack pointer
;
          TSC            Transfer stack pointer to accumulator
          CLC            Clear carry flag (required before addition)
          ADC   #$0003   Add 3 to accumulator
          TCS            Transfer accumulator to stack pointer.  Stack pointer
;                          is now above return address.
```

```
          PLA             Pull word off stack into accumulator.  This is first
;                           two bytes of pointer to character string.
          STA   $AA       Store accumulator in direct page at $AA
          SEP   #$20      Set 8-bit accumulator
          LONGA OFF       Set long addresses off
          PLA             Pull next 8 bits into accumulator.  This is bank
;                           address of pointer to character string.
          STA   $AC       Store accumulator into $AC  Full 3-byte pointer
;                           to character string is now in direct page starting
;                           at $AA.
          REP   #$20      Set 16-bit accumulator
          LONGA ON        Set long addresses on
          TSC             Transfer stack pointer to accumulator
          SEC             Set carry flag (required before subtraction)
          SBC   #$0006    Subtract 6 from accumulator
          TCS             Transfer accumulator to stack.  Stack pointer is now
;                           back where it was when control was transferred to
;                           this routine.
          END             End of first object segment
;
LOOP      START           Start of second object segment
          LDA   [$AA]     Load accumulator with the value of the character
          AND   #$00FF    Extract the value of the character
          CMP   #$0000    Is it end of string?
          BEQ   FINISH    If it is then go to FINISH
          CMP   #$0061    Compare accumulator to 'a' ($61)
          BLT   ITERATE   Character is smaller than 'a' - go to the next one
          CMP   #$007A    Compare accumulator to 'z' ($7A)
          BEQ   UPPER     If equal then convert to upper case
          BGE   ITERATE   Character is greater than 'z' - go to the next one
UPPER     SEC             Set carry
          SBC   #$0020    Convert the character in accumulator to upper case
          SEP   #$20      Set 8-bit accumulator
          LONGA OFF       Switch to short addresses
          LONGI ON        Switch to long integers
          STA   [$AA]     Store accumulator (8-bit only!) to string element
          REP   #$20      Reset m status bit
ITERATE   LONGA ON        Set long addressing on
          LONGI ON        Set long integers on
          INC   $AA       Increment by one (go to the next char)
          BNE   LOOP      Branch non-zero to 52
          BRA   LOOP      Perform the next iteration of the LOOP
          END             End of second object segment
;
FINISH    START           Start of third object segment
          RTL             Return to C routine
          END             End of third object segment
```

Figure 3.2. Sample 65816 Source Code

Creating object code: compiling and assembling

To compile and assemble your programs, use the following commands:

```
COMPILE SAMPLEC  KEEP=SAMPLEC.O
```

```
ASSEMBLE SAMPLEA  KEEP=SAMPLEA.O
```

Note: If you have a two 3.5" drives and no hard disk, you will have to compile using the APW C disk and assemble using the APW assembler disk. You may have to restart APW from the APW assembler disk before you can assemble. Once you have compiled and assembled your source files, you can link the object files using either the APW assembler disk or the APW C disk.

If an APW compiler finds a fatal error (one that prevents the compile from continuing), it writes out an error message to standard output (normally the screen) and waits for you to press any key. When you press a key, the compiler passes control to the APW Editor, which loads the source file that the compiler was working on, placing the line that caused the error at the top of the screen.

If the compiler finds a nonfatal error, it finishes processing the program, writes out the error messages, and returns control to the shell.

If your first attempt was not successful, correct the source code and try again. Repeat this process until the module compiles successfully. Remember to save the source file each time you make changes: the disk file is updated only when you save it.

When the compiler processes the file, it takes the first segment that will be executed when the program is run and places it in an object file with the root filename you specified and the extension `.ROOT` (some compilers do not append any filename extension to the root file). All other segments (if any) are placed in a second object file with the same root filename and the extension `.A`.

The following files should be on your disk after using these commands:

- `SAMPLEC` C source code
- `SAMPLEA` 65816 source code
- `SAMPLEC.O.ROOT` object segment created by the C compiler
- `SAMPLEA.O.ROOT` first object segment created by the assembler
- `SAMPLEA.O.A` the rest of the object segments created by the assembler

Note that since the C routine contains only one function call, there is only one segment in the object file created by the C compiler and hence only the root file is created.

Alternatively, you can compile both files in one operation. To do this, you can add a line to the file `SAMPLEC` as follows:

1. Reopen the file in the editor with the following command:

   ```
   EDIT SAMPLEC
   ```

2. Press ⌘-9 to jump to the end of the file. Add the following line to the file:

   ```
   #append "SAMPLEA"
   ```

3. Press ⌘-Q to quit the editor, S to save the file, and E to exit the editor.
4. Now when you use the following command, the shell calls the C compiler to compile the C routine, then calls the APW Assembler to assemble the 65816 routine:

```
COMPILE SAMPLEC KEEP=SAMPLE.O
```

The following files should be on your disk after using this command:

- `SAMPLEC` C source code
- `SAMPLEA` 65816 source code
- `SAMPLE.O.ROOT` first object segment created by the C compiler
- `SAMPLE.O.A` object segments created by the assembler

Creating load files: linking

When you execute the `LINK` command, the APW Linker combines all object segments that have the same load segment name into the same load segment, and places the entire program into a single load file with the `KEEP` filename you specified. (For a discussion of object segments and load segments, see the section "APW C Concepts" in Chapter 1.)

> **Important:** Be sure to include the `KEEP` parameter in the `LINK` command. If you do not specify a `KEEP` filename in the `LINK` command, no load file is saved to disk.

Here are two ways to link the object files you have just created:

1. If you did *not* add the `#append` directive to the end of the C routine, use the following command to link the object files into a single executable load file:

   ```
   LINK 2/START SAMPLEC.O SAMPLEA.O KEEP=SAMPLE
   ```

   The first file listed links the file `START.ROOT` in the library prefix. This file must be linked to the beginning of every program when the main segment is in C.

   The load file is named `SAMPLE`.

   The following files should be on your disk after using this command:

   - `SAMPLEC` C source code
   - `SAMPLEA` 65816 source code
   - `SAMPLEC.O.ROOT` first object segment created by the C compiler
   - `SAMPLEA.O.ROOT` first object segment created by the assembler
   - `SAMPLEA.O.A` the rest of the object segments created by the assembler
   - `SAMPLE` load file

2. If you *did* add the `#append` command to the end of the C routine, use the following command to link the object files into a single executable load file:

   ```
   LINK 2/START SAMPLEC.O KEEP=SAMPLE
   ```

The following files should be on your disk after using this command:

- `SAMPLEC` C source code
- `SAMPLEA` 65816 source code
- `SAMPLE.O.ROOT` first object segment created by the C compiler
- `SAMPLE.O.A` object segments created by the assembler
- `SAMPLE` load file

Running your program

To run the program you just created, use the following command:

```
SAMPLE
```

Each character you type is printed on the screen as you type it. Press Return to have the program retype the line in all uppercase. Press Control-C to terminate the program. The following sequence illustrates the use of this routine. The characters in boldface are the ones you type (remember to press Return at the end of each line you type):

```
#SAMPLE
Now is the Time for aLL good PeoPle to Buy an Apple II gs
NOW IS THE TIME FOR ALL GOOD PEOPLE TO BUY AN APPLE II GS
Granny Smith is always getting her apples into a jam
GRANNY SMITH IS ALWAYS GETTING HER APPLES INTO A JAM
Control-C
#
```

You can use I/O redirection to use this routine to convert the characters in a file to uppercase. The following command converts all the characters in the file `TEXT.IN` to uppercase and writes them out to the file `TEXT.OUT`:

```
SAMPLE  <TEXT.IN  >TEXT.OUT
```

The file `TEXT.OUT` contains the output that would have appeared on the screen; that is, each line of text in the file `TEXT.IN` is printed, followed by the same line converted to uppercase.

Compiling, linking, and running in one step

*****Note: In the present (B3) incarnation of APW, this example does not work. This must be tested before the final draft to make sure it works as described.*****

If you are using a hard disk, you can use a single APW command to compile, link, and run your program in one step. Here are two ways to do so:

1. If you did *not* add the `#append` directive to the end of the C routine, use the following commands to compile, link, and run the program:

```
SET KEEPNAME %.O
```

```
RUN SAMPLEC SAMPLEA
```

The SET KEEPNAME command establishes a default filename for output files. The percent sign (%) is a wildcard character that APW replaces with the source filename; the default root filename for the file SAMPLEC is thus SAMPLEC.O. The load file is given the same name as the root filename of the first object file created. For this command, for example, the first object file created has the name SAMPLEC.O.ROOT and the load file has the name SAMPLEC.O

The first file listed links the file START.ROOT in the library prefix (be sure to use the correct prefix for your system). This file must be linked to the beginning of every program when the main segment is in C.

The following files should be on your disk after using this command:

- SAMPLEC C source code
- SAMPLEA 65816 source code
- SAMPLEC.O.ROOT first object segment created by the C compiler
- SAMPLEA.O.ROOT first object segment created by the assembler
- SAMPLEA.O.A the rest of the object segments created by the assembler
- SAMPLEC.O load file

2. If you *did* add the #append command to the end of the C routine, use the following command to compile, link, and run the program:

```
RUN SAMPLEC KEEP=SAMPLE
```

The following files should be on your disk after using this command:

- SAMPLEC C source code
- SAMPLEA 65816 source code
- SAMPLE.ROOT first object segment created by the C compiler
- SAMPLE.A object segments created by the assembler
- SAMPLE load file

When you use the RUN command, APW automatically executes the program after the compile and link processes are complete.

Creating a compact load file

As a final step in program development, you can run the Compact utility program. Compact converts a load file to the most compact form provided by the object module format. If your load file is named SAMPLE, type the following line and press Return:

```
COMPACT SAMPLE SAMPLE.CMPCT
```

Compacted load files take up less space on disk and load faster than noncompacted load files. The SAMPLE program you created here, for example, should be about 31 blocks in size (as shown in a catalog listing), while SAMPLE.CMPCT should be about 25 blocks.

The Compact utility writes to the screen an account of the records it has converted. If you are interested in understanding the format and use of these records, see the section "Segment Body" in chapter 8 of the *APW Reference*.

Not all load files are significantly improved by compacting, however, so you may want to test both a compacted and noncompacted version of your program before releasing it.

Important: In order to load a compacted load file, you must have version 1.2 or later of the System Loader on your boot disk.

Part II

Language Reference

Chapter 4
The APW C Language

The information provided in this chapter supplements *The C Programming Language* by Kernighan and Ritchie. Where their language definition leaves choices to the implementers, this chapter describes how these aspects of C have been implemented on the Apple IIGS. Where Apple has modified or extended their language definition, this chapter documents the changes.

Language definition

This section describes the APW C language, including language extensions such as type `void`, type `enum`, the SANE data types, and calling Pascal-compatible functions.

Variable names

The compiler limits the length of each local variable name to 1000 characters. Global variable names and function names are limited to 250 characters by the object-module format. Therefore, different function names whose first 250 characters are identical will be treated as different functions by the compiler but will be treated as the same function by the linker.

Data types

Table 4-1 lists the arithmetic and pointer types available in APW C and shows the number of bits allocated for variables of these types. Types `short` and `long` represent 16-bit and 32-bit integers, respectively. The machine type `int` is a 16-bit integer on the Apple IIGS: it is the type the 65C816 uses most efficiently. Pointers require 32 bits. Enumeration types require 16 bits. Types `char`, `short`, `int`, and `long` use two's-complement representation. Note that The Apple IIGS has no signed 8-bit type: `char` and `unsigned char` are identical. Naturally, a prudent programmer will make no assumption about features not guaranteed to be portable.

Table 4-1. Size and range of data types

Data type	Bits	Description
`char`	8	Range 0 to 255
`unsigned char`	8	Range 0 to 255
`short`	16	Range –32,768 to 32,767
`unsigned short`	16	Range 0 to 65,535
`int`	16	Range –32,768 to 32,767
`unsigned int`	16	Range 0 to 65,535
`long`	32	Range –2,147,483,648 to 2,147,483,647
`unsigned long`	32	Range 0 to 4,294,967,295
`enum`	16	Range 0 to 65,535
*	32	Pointer types
`float`	32	IEEE single-precision floating point
`double`	64	IEEE double-precision floating point
`comp`	64	SANE signed integral values
`extended`	80	IEEE extended-precision floating point

Note: Some programs assume that

```
sizeof(int) = sizeof(char *)
```

These programs may not work properly under APW C, because an `int` is 2 bytes long and a pointer is 4 bytes.

You can find more information on types in Table 4-2.

Numeric constants

Integer constants in the range of `long` are treated as type `long`. Integer constants in the range of `unsigned long` are treated as type `unsigned long`. Integer constants outside the union of the ranges of the `long` and `unsigned long` types are treated as type `extended`. For example, the initialization statement

```
long i = 4000000000;
```

is incorrect because 4,000,000,000, being too big for a `long`, is interpreted as an `extended` value. However, the initialization statement

```
unsigned long i = 4000000000;
```

is correct because 4,000,000,000 is within the range of `unsigned long` values.

Type void

The `void` keyword tells the compiler that the function being declared does not return a value. Calls to functions of type `void` may not be used in expressions, where a value is required. (See "Pascal-Style Functions" later in this chapter.)

Type enum

Type `enum` is a type analogous to the enumeration types of Pascal. Its syntax is similar to that of the `struct` and `union` declarations:

enum-specifier:
`enum { `*enum-list* `}`
`enum` *enumeration-tag* `{` *enum-list* `}`
`enum` *enumeration-tag*

enumeration tag:
identifier

enum-list:
enumeration-declaration
enumeration-declaration , *enum-list*

enumeration-declaration:
identifier
identifier = *constant-expression*

The optional*enumeration-tag* in *enum-specifier*, like the structure tag in a *struct-specifier*, names a particular enumeration type, and allows you to define other objects of that type. For example,

```
enum color {chartreuse, burgundy, claret, winedark};
. . .
enum color *cp, col;
```

This enumeration makes `color` the enumeration-tag of a type describing various colors and then declares `cp` as a pointer to an object of that type and `col` as an object of that type. The identifiers in *enum-list* are declared as constants and may appear wherever constants are required.

If no enumerators with a *constant-expression* appear, the values of the constants begin at 0 and increase by 1 as the declaration is read from left to right. Each enumerator with a *constant-expression* is given the value indicated. Each enumerator without a *constant-expression* is given a value one greater than the enumerator before it. This means that two or more enumerators with *constant-expressions* can be assigned the same constant value, and that an enumerator without a *constant-expression* may have the same value assigned by the compiler as one with a *constant-expression* in the same enumeration list. Let us consider some examples:

```
enum digit {zero,one,two,three,four,five,six,seven,eight,nine } num;
```

has the values 0, 1, 2, 3, 4, 5, 6, 7, 8, 9

```
enum mixedup {a,b,c,d = 1,e,f } mix;
```

has the values 0, 1, 2, 1, 2, 3

```
enum zapped {g = 1, h,i,j =2,k,1} zap;
```

has the values 1, 2, 3, 2, 3, 4

```
enum ok {m=45,n,o,p=100,q,r};
```

has the values 45, 46, 47, 100, 101, 102

It you declare values, it is safest to declare all of them.

Each *enumeration-tag* and *enumeration-constant* must be unique. They are drawn from the set of ordinary identifiers, unlike structure tags and members. Objects of a given enumeration type have a type distinct from objects of all other types.

Enumeration types are allocated the amount of space required by the smallest predefined type that allows representation of all of the literal values specified by the enumeration. The predefined types considered are `unsigned char` (8 bits) and `unsigned short` (16 bits).

Register variables

Most versions of C support register variables. Their function is undefined in The Apple IIGS as a result of the small number of registers available on the 65C816 microprocessor. Use of the `register` declaration causes the compiler to generate code at least as efficient as that generated by the same program without `register` declarations.

Structures

Structures may be assigned, passed as parameters, and returned as function results. The left and right sides of a structure assignment must have identical types. Similiarly, actual and formal parameters must have identical types. Equality comparison for structures has been implemented, provided the structures have the same type. (The equality test may give unpredictable results if the structure contains a `union`.)

Since the 65C816 is a byte-oriented machine, data structures can be aligned on byte boundaries. For this reason, APW C does not pad structures to ensure word alignment.

> **Important:** In functions that return structures, if an interrupt occurs during the return sequence and the same function is called reentrantly during the interrupt, the value returned from the first call may be corrupted. The problem can occur only in the presence of interrupts. Recursive calls are quite safe.

Reserved symbols

__LINE__ is a reserved preprocessor symbol whose value is the current line number within the current source file.

__FILE__ is a reserved preprocessor symbol whose value is a character string consisting of the current file name.

__LINE__ and __FILE__ begin and end with two underscore characters.

The symbol **AppleIIgs** is predefined for use in conditional compilation. It can be used to distinguish C code written for the APW C compiler from C code written for, say, the MPW C compiler. The symbol has the value 1, as if a statement of this form had appeared at the beginning of the source code:

```
#define AppleIIgs 1
```

The symbol **APW** is predefined for use in conditional compilation. It can be used to distinguish C code written for the APW C compiler from C code written for some other compiler.The symbol has the value 1, as if a statement of this form had appeared at the beginning of the source code:

```
#define APW 1
```

The symbol **WD65816** is predefined for use in conditional compilation. It can be used to distinguish C code written to run on the Western Design Center 65SC816 from C code written to run on some other microprocessor, even for some other flavor of 65816.The symbol has the value 1, as if a statement of this form had appeared at the beginning of the source code:

```
#define WD65816 1
```

Any of these can be tested by an `ifdef` statement.

Standard Apple Numeric Environment extensions

APW C has built-in support for the Standard Apple Numeric Environment (SANE). In combination with the C SANE library the language composes a scrupulously conforming extended-precision implementation of the IEEE Standard for Binary Floating-Point Arithmetic (754). SANE provides an extra data type for storing large integral values and basic functions for application development. APW C recognizes the SANE data types, uses SANE for all C floating-point operations and conversions, and correctly handles NaNs (Not-a-Number) and infinities in comparisons and in ASCII–binary conversions. Furthermore, source programs from other C implementations, if they are written using only `float` and `double` types and standard C operations, will compile and run under APW C without modification.

Much of SANE is provided through the run-time library `CLIB` and the include file `SANE.H`. However, to use extended-precision arithmetic efficiently and effectively, and to handle IEEE NaNs and infinities, some extensions to standard C are required, including use of the `extended` data type.

A change from `double` to `extended` as the basic floating-point type is the most important difference from standard C. Since C was originally developed on the DEC PDP-11, the PDP-11 architecture is reflected in standard C in the use of `float` and `double` as floating-point types, with `double` as the basic type: floating-point

expressions are evaluated to `double`, anonymous variables are `double`, and floating-point parameters and function results are passed as `double` values. However, the low-level SANE arithmetic (as well as the Intel 8087, Motorola 68881, and Zilog Z8070 floating-point chips) evaluates arithmetic operations to the range and precision of an 80-bit `extended` type. Thus, `extended` naturally replaces PDP-11 `double` as the basic arithmetic type for computing purposes. The types `float` (IEEE single), `double`, and `comp` serve as space-saving storage types, just as `float` does in standard C. The `comp` type, a 64-bit type for storing integral values, is a SANE extension. It has two properties that suit it to accounting applications: it is sufficiently large to represent the U.S. national debt in Argentine pesos, and it has a NaN value to record overflows and other exceptions.

The IEEE Standard specifies two kinds of special representations for its floating-point formats: NaNs and infinities. APW C expands the syntax for I/O to accommodate NaNs and infinities, and includes the treatment of NaNs in relationals as required by the IEEE Standard.

The SANE extensions to standard C are backward-compatible: programs written with only the `float` and `double` floating-point types and standard C operations compile and run without modification. All intermediate values are computed in the `extended` type, an 80-bit floating-point type, and the results are returned to the types specified in the program. SANE does not affect integer arithmetic.

The *Apple Numerics Manual* contains detailed documentation of SANE. The *Apple IIGS Toolbox Reference* contains detailed documentation of the Apple IIGS SANE Toolset, which makes SANE available on the Apple IIGS.

Constants

Numeric constants that include floating-point syntax—a point (`.`) or an exponent field—or that lie outside the union of the ranges of the `long` and `unsigned long` types are of `extended`. Decimal-to-binary conversion for numeric constants is done at compile time (and hence is governed by the default numeric environment: see the section "Numeric Environment" in this chapter).

Expressions

The SANE types—`float`, `double`, `comp`, and `extended`—can be mixed in expressions with each other and with integer types in the same manner that `float` and `double` can in standard C. An expression consisting solely of a SANE-type variable, constant, or function is of type `extended`. An expression formed by subexpressions and an arithmetic operation is of type `extended` if either of its subexpressions is. Expressions of type `extended` are evaluated using extended-precision SANE arithmetic, with conversions to type `extended` generated automatically as needed. Parentheses in `extended`-type expressions are honored: the compiler will not rearrange terms in violation of parentheses. Initialization of external and static variables, which may include expression evaluation, is done at compile time; all other evaluation of `extended`-type expressions is done at run time.

Comparison involving a NaN

The result of a comparison involving a NaN operand is **unordered.** The usual set of comparison results—less than (<), greater than (>), and equal to (==)—is expanded to include unordered. For example, the negation of "*a* less than *b*" is not "*a* greater than or equal to *b*" but "(*a* greater than or equal to *b*) OR (*a* and *b* unordered)." The `CLIB` function `relation` tests all four alternatives.

Parameters and function results

A numeric actual parameter passed by value is an expression and hence is of `extended` or `integer` type. All `extended`-type arguments are passed as `extended` values. Similarly, all results of functions declared `float`, `double`, `comp`, or `extended` are returned as `extended` values.

Numeric input/output

In addition to the usual syntax accepted for numeric input, the Standard C Library function `scanf` recognizes the string "`INF`" as infinity and the string "`NAN`" as a NaN. "`NAN`" may be followed by parentheses, which may contain an integer (a code indicating the NaN's origin). "`INF`" and "`NAN`" are optionally preceded by a sign and are case-insensitive. The `scanf` specifiers for SANE types extend standard C as follows: conversion characters `f`, `e`, and `g` indicate type `float`; `lf`, `le`, and `lg` indicate type `double`; `mf`, `me`, and `mg` indicate type `comp`; and `ne`, `nf`, and `ng` indicate type `extended`.

The Standard C Library function `printf` writes infinities as "`INF`" and NaNs as "`NAN` (*ddd*)", where *ddd* is the NaN code. "`INF`" and "`NAN` (*ddd*)" may be preceded by a minus sign.

Numeric environment

The **numeric environment** comprises the rounding direction, rounding precision, halt enables, and exception flags. IEEE Standard defaults—rounding to nearest, rounding to extended precision, and all halts disabled—are in effect for compile-time arithmetic (including decimal-to-binary conversion). Each program begins with these defaults and with all exception flags clear. Functions for managing the environment are included in the library `CLIB`. The compiler, in optimizing, will not change any part of the numeric environment, including the exception-flag setting, which is a side effect of arithmetic operations.

About the C SANE Library

The SANE library provides the basic tools for developing a wide range of applications. It includes the following:

- logarithmic, exponential, and trigonometric functions
- financial functions
- random-number generation

- conversions between binary and decimal
- numeric scanning and formatting
- environment control
- other functions required or recommended by the IEEE Standard

Additional information can be found in the SANE Tool Set chapter of the *Apple IIGS Toolbox Reference*.

Programming with IEEE arithmetic

APW C's automatic use of the `extended` type produces results that are generally better than those of other C systems. For example, extended precision yields more accuracy and extended range avoids unnecessary underflow and overflow of intermediate results. You can further exploit the `extended` type by declaring all floating-point temporary variables to be of type `extended`. This is both time-efficient and space-efficient, since it reduces the number of automatic conversions between types. External data should be stored in one of the three smaller SANE types (`float`, `double`, or `comp`), not only for economy but also because the `extended` format may vary between SANE implementations. As a general rule, use `float`, `double`, or `comp` data as program input; `extended` arithmetic for computations; and `float`, `double`, or `comp` data as program output.

In many instances, IEEE arithmetic allows simpler algorithms than were possible without IEEE arithmetic. The handling of infinities enlarges the domain of some formulas. For example, $1+1/x^2$ computes correctly even if x^2 overflows. Running with halts disabled (the default), a program will never crash because of a floating-point exception. Hence, by monitoring exception flags, a program can test for exceptional cases after the fact. The alternative of screening out bad input is often infeasible, and sometimes impossible.

Pascal-style functions

The function-calling conventions used by APW C and by conventional Pascal implementations differ in the order of parameters on the stack, the type coercions applied to parameters, and the location of the return result. Like the Macintosh Toolbox, the Apple IIGS Toolbox adheres to Pascal-style calling conventions. APW C has been extended to allow you to use both C-style and Pascal-style calling conventions. The specifier `pascal` in a function declaration or definition indicates a Pascal-style function. This extension is intended to allow for the addition of Pascal and other lanugages to APW.

Pascal-style function declarations

A function or procedure written using Pascal-style calling conventions can be called from APW C. Before it can be called, it must be declared as an external function. Here is the general form for a declaration:

`[extern] pascal` [*result-type*] *func-name* ();

This declaration says that the Pascal procedure named *func-name* can be called from your program, returning a result of type *result-type*.

For example, the `DrawText` procedure would be defined in Pascal as follows:

```
PROCEDURE DrawText (textBuf: Ptr;
      firstByte, byteCount: integer);
```

The syntax for declaring this procedure so that it can be called from APW C is

```
extern pascal void DrawText();
```

To make the code more informative, we can list the parameters in a comment:

```
extern pascal void DrawText();
      /* Ptr textBuf;
      short firstByte, byteCount; */
```

Inline declarations

An inline declaration is used for declaring Apple IIGS tool routines. Its syntax is

`[extern] pascal [`*result-type*`] `*func-name*` () inline (`*m*`,`*n*`);`

This says that the tool routine with tool call number *m* and tool locator entry point *m* can be called by the function name *func-name* and that it returns a result of type *result-type*. The `pascal` keyword is necessary because the tools use Pascal-style conventions.

Inline assembly-code declarations

Your C program can contain assembly code inline. Anywhere a statement is legal, you can insert a series of assembly-language statements with this format:

`asm{`*assembly-statements*`}`

Anywhere a function definition is legal, you can have a definition with this format:

`asm(`*external-name*`) {`*assembly-statements*`}`

This function can be called in the same way as a C function called *external-name*. Here *external-name* is the entry point of the segment containing the assembly-language code.

Pascal-style function definitions

A C function definition (the actual function), like a function declaration, can also be preceded by the `pascal` specifier. The C compiler then produces code that adheres to Pascal-style calling conventions and the function can be called using these conventions.

The APW syntax for defining this procedure as a C function is

`pascal [`*result-type*`] `*func-name*` (`*formal-parameter-list*`) {`*statement-list*`}`

For example, the following C function could be called from Pascal:

```
pascal void MyText(byteCount,textAddr,numer,denom)
      short byteCount;
      Ptr textAddr;
      Point numer,demon;
{
      ...
}
```

The corresponding Pascal function declaration would be

```
PROCEDURE MyText(bytecount: INTEGER; textAddr: Ptr;
      numer,denom: Point);
```

For compatibility with Pascal and assembly language, the compiler converts the names of Pascal-compatible functions to uppercase before writing them to the object file. When they are called in C programs, these routines should be capitalized exactly as they were declared in C. Pascal-compatible functions whose names differ only in their capitalization will become duplicate declarations when their names are converted to uppercase by the compiler; therefore such names should be avoided.

Pascal-style strings: \p

One of the complications of calling Pascal-style functions from C is that the two languages have different conventions for handling strings. A C-style string is a set of characters followed by a null byte; a Pascal-style string is a count byte *n* followed by a set of *n* characters. Conveniently, these two forms are the same length, so conversion from one to the other is not hard. The functions `c2pstr` and `p2cstr` perform runtime conversions between the two types of strings.

If you wish to call a Pascal-style function that expects a Pascal-style string, you can use an Apple extension to the standard C character escapes: `\p`. When the compiler encounters this escape sequence at the beginning of a string, it substitutes for the `\p` the character value equivalent to the number of non-null characters in the remainder of the string. Thus a string constant is created that is equivalent to a Pascal-style string. Since it is also a C-style string, it is also terminated by the null character: this character is not included in the character count.

You can use it like this:

```
WriteString("\pHello, world.\n");
```

Parameter and result data types

C and Pascal support different data types. When writing a Pascal-style function declaration in C, a translation of the parameter types and function-result type (from Pascal to C) is therefore required. Often this translation is obvious, but some cases are surprising.

Table 4-2 summarizes this translation. Find the Pascal parameter or result type in the first column. Use the equivalent C type found in the second column when declaring the

function in C. Comments in the table point out unusual cases which may require special attention.

Table 4-2. Parameter and result data types

Pascal Data Type	C Equivalent	Comments
`enumeration`	`enum`	Use identical ordering of the `enumeration` literals.
`var enumeration`	`enum *`	
`enumeration result`	`enum`	
`char`	`char`	Pascal passes `chars` as 16-bit values.
`var char`	`char *`	Pascal stores unpacked `chars` as 16-bit values.
`char result`	`char`	
`integer`	`int or short`	16-bit signed values
`var integer`	`int * or short *`	
`integer result`	`int or short`	
`longint`	`long`	32-bit signed values
`var longint`	`long *`	
`longint result`	`long`	
`real`	`float *`	Pascal passes `real` parameters as `extended`.
`var real`	`float *`	
`real result`	`float`	
`double`	`double *`	Pascal passes `double` parameters as `extended`.
`var double`	`double *`	
`double result`	`double`	
`comp`	`comp *`	Pascal passes `comp` parameters as `extended`.
`var comp`	`comp *`	
`comp result`	`comp`	
`extended`	`extended *`	
`var extended`	`extended *`	
`extended result`	`extended`	

`pointer`	`pointer`	32-bit addresses
`var pointer`	`pointer *`	
`pointer result`	`pointer`	
`array`	`array`	Pascal passes `arrays` by address.
`var array`	`array`	
`array result`	---	C does not allow `array` results.
`record`	`struct`	Pascal passes `records` by value.
`var record`	`struct *`	
`record result`	`struct`	
`set`	`struct`	Pascal passes `sets` by value.
`var set`	`struct *`	
`set result`	`struct`	

Note: The C `struct` type and the Pascal `record` type do not exactly correspond, as C lacks an equivalent to the Pascal `variant record` type.

Global and external data types

When a C program and a Pascal program use the same global or external variables, they must use types of like size. This requires care, as one can't be sure whether a given Pascal compiler puts 0..255 into a byte or a word. If possible, use a signed type for a signed type. If you have to pass values from a signed type into an unsigned type, or vice versa, you will have to test the sign bit and perform the appropriate conversions.

How parameters are passed

High-level languages on the Apple IIgs use the stack and the A and X registers to pass parameters. Assembly-language programs have other means of passing parameters, such as the direct page, but they must use the stack to communicate with C programs, because this is how C expects parameters to be passed. Here's how it works.

C-style functions

Let's declare a typical C-style function:

```
int foo();
```

This function takes three values and returns one result. We can call it like this:

```
zoo = foo(a,b,c);
```

When the call is executed, the values `c`, `b`, and `a` are pushed, in that order. Function `foo` returns its result in the A register. The calling program then pulls `a`, `b`, and `c` off the stack and stores the contents of the A register into the variable `zoo`.

If `foo` had been 4 bytes long, it would have been returned in the A and X registers: the high bytes in X and the low bytes in A. Structure and extended results are returned by passing a pointer to them in the A and X registers.

Pascal-style functions

Pascal-style functions use the stack for the return value and also reverse the order of reading parameters. Consider this function:

```
pascal int foobar();
```

This function also takes three values and returns one result. We can call it like this:

```
x = foobar(a,b,c);
```

When the call is executed, space for the result `foobar` is pushed on the stack, then the values `a`, `b`, and `c` are pushed, in left-to-right order. The routine pulls `c`, `b`, and `a` off the stack, computes `foobar`, and pushes `foobar` on the stack. The calling program then pulls `foobar` off and copies it into the variable `x`.

When you write a function, you can declare it as a C or a Pascal-style function, thus determining the way the parameters are passed. The C style of passing parameters is more efficient than the Pascal style, but it should be used only with functions that will be called from C and not from Pascal. Whatever language a function is written in, if declared as a Pascal-style function it can be called from either Pascal or C; if declared as a C-style function it can be called only from C.

Implementation notes

A number of details in any language definition are left to the discretion of its individual implementations. Most programs do not rely on these details and therefore yield the same results on the various implementations. Knowledge of the major differences between implementations can, however, help you avoid reliance on language semantics that vary from implementation to implementation. This section explains several areas of the language definition that are specific to APW C.

Size and byte-alignment of variables

Because the 65C816 is a byte-oriented processor, it levies no speed penalty for using odd addresses. Therefore, APW C does not align variables on word boundaries. In particular, enumerated types and structures are not padded to make fields fall on word boundaries.

When you recompile an MPW C program on the APW C compiler, for example, all padding added by the MPW C compiler disappears. Any padding you added remains. You can save space and possibly time by removing this padding from data structures and deleting code that does word alignment.

Byte ordering

On the 65C816, the microprocessor used in the Apple IIgs, the least significant byte of a short or long integer has the lowest memory address. This byte ordering is also used on the PDP-11, VAX, 8086, and NS16000 processors. The 68000 , IBM/370, and Z8000 processors store the least significant byte at the highest address. Programs that rely on the order of the bytes within words and long words will not be portable from machines of one of these classes of machines of the other.

Sign extension

In APW C, the >> operator always performs a logical right shift: that is, the left bit positions are filled with 0's.

Variable allocation

The APW C compiler allocates static and global variables in the order in which they appear in the source. This is also true for the order of fields within structures.

Array indexing

Array indexing is done using long arithmetic wherever the compiler cannot determine the actual size of the array (as in `extern int array[];`) or determines that the size requires long arithmetic for correct calculation of offsets for the whole array.

If the compiler determines that the entire array can be accessed using word arithmetic, it will do so, for example:

```
extern int array[N]; /* N <= 0x8000 */

char string[] = "It would be hard to create a string long
enough to require long indexing, wouldn't it?"

int notToMany[] = {0,1,2,3,4,5,6,7,8,9};
long larray[0x4000];

long larray[0x4000]; /* Though the array is too large, the
second index will be done with word arithmetic. This is of
dubious advantage. */
```

Because word arithmetic is more efficient than long arithmetic, you can use certain tricks to force word arithmetic when speed is important. These apply whenever you only need to access 64K (0x1000) bytes within an array.

1. The form

```
extern int array[1000];
```

   is better than the form

```
extern int array[];
```

   (as long as you know about how much of the array you need to access.)

2. To optimize access to a part of a larger array, place the code in a subroutine and pass a pointer to the first element of the part to the subroutine: e.g.,

```
long array[0x10000] /*This will normally cause long index
                      arithmetic.*/

main()
{ unsigned int i;
   for(i=0; i<4; i++) fill(array+i*0x4000);}
fill(smaller)
long smaller[0x4000]; /*This is just small enough to force
                        word index arithmetic.*/
{ unsigned i;
  for (i=0; i < 0x4000; i++) smaller[i] = 0xFFFFFFFF; }
```

   Calling `fill()` four times allows us to fill an array whose actual size in bytes is `0x40000`, using long-arithmetic index calculation only four times, once at each call from `main`. Note that the arithmetic is further optimized by the use of `unsigned` for `i`.

Types unsigned char, unsigned short, and unsigned long

Types `unsigned char`, `unsigned short`, and `unsigned long` are supported by the APW C compiler and by many implementations of PCC, although they are not required by the basic C language definition. The VAX implementation of PCC and the APW C compiler differ in the way they evaluate expressions involving these types. For example, the negation operator subtracts an `unsigned short` from 2^{16} under PCC, and from 2^{32} under APW C.

Bit fields

APW C does not support signed bit fields. In the following example, implementations using unsigned bit fields will set `i` to 3:

```
struct {unsigned int field:2;}  x;
   x.field  =  3;
   i  =  x.field;
```

Evaluation order

APW C does not define the evaluation order of certain expressions. Expressions with side effects, such as function calls and the ++ and – – operators, may yield different results on different machines or with different compilers. Specifically, when a variable is modified as a side effect of an expression's evaluation and the variable is also used at another point in the same expression, the value used may be either the value before modification or the value after modification.

Programs that rely on the order of evaluation in these situations are in error. The function call

```
f(i,i++)
```

is an example of an expression whose value is undefined.

Case statements

Some implementations of C, including PCC, allow cases of a `switch` statement to be nested within compound statements. APW C considers this an error. The following `switch` statement compiles using PCC but generates an error message using the APW C compiler. The error is that `case 2:` is within the `if` statement.

```
switch (i)  {
   case 1:
      if (j)  {
         case 2:
            i = 3;
      }
   }
```

Language anachronisms

Several constructs formerly considered part of the C language are now considered anachronisms. The compiler considers these invalid. The anachronisms are described below.

Assignment operators

The *=op* form of assignment operators is not supported. Alternative interpretations are accepted without warning. In particular,

`x =- 5;`	is interpreted as	`x = (-5);`
`x =* 5;`	is interpreted as	`x = (*5);`
`x =& p;`	is interpreted as	`x = (&p);`

Initialization

The equal sign that introduces an initializer must be present. The anachronism

```
int i 1;
```

is considered an error.

Compiler limitations

On the Apple IIGS, the total size of all declared global scalar variables, static scalar variables, and scalar constants cannot exceed 64K, because they are accessed using short addressing. Aggregate types (structures, arrays, and string constants) are stored in a separate large memory segment and accessed with long addressing. Their size is effectively limited only by available memory.

Automatic variables are limited by the available stack space, which can never exceed 32K.

Each code segment is limited to 64K.

Performance tips

The following practices improve performance:

- Use `unsigned` types whenever possible. (This improves performance markedly.)
- Declare `auto` aggregate variables after all `auto` scalars. (This improves performance markedly.)
- Declare `auto` pointers before other `auto` variables.

Creating load segments: the segment command

`segment "`*segname*`"{, dynamic}"`

When this command is used, it must appear between functions. It assigns the load segment name *segname* to a function: all code following the directive until the end of file or the next `segment` command will be assigned to the load segment *segname*. By default, this command creates a static load segment. The `dynamic` option creates a dynamic segment.

The `segment` command can be used to split up a code segment that would be larger than 64K.

The #append directive

The APW C preprocessor processes the usual directives, plus one that is peculiar to APW C:

`#append` *filename*

When this directive is used, it must appear between functions: *filename* is the name of the next file in the compilation sequence. This directive normally appears at the end of a file, as everything after it will be ignored. It should not appear in an include file.

Code Generation Memory Model

The memory model used by the code generation is a mixed model, intended to most effectively exploit the architecture of the 65816, which has addressing modes that deal with memory in a linear fashion, and others which treat memory as being divided into segments.

Essentially, long, or linear, addressing is used for all pointer values: pointers are 32-bit values, which contain 24-bit machine addresses. Global scalar variables, however, are referenced internally using the more efficient 16-bit addressing modes. For these operations, the high byte of the 24-bit address is derived from the processor's data bank register, which is initialized by the `START.ROOT` module to point to the bank in which the load segment containing the global data has been loaded. This feature is why total global scalar storage is limited to 64K. Global arrays and structures, on the other hand, are always addressed using long addressing, so it is possible to have more than 64K of array space. `Structs` and `unions` are accessed using indexed addressing, so they are limited in size to 64K. Array refences will use the faster 16-bit indexed addressing modes if the array is less than 64K in size. To access elements within large (greater than 64K) arrays, the index expression must evaluate to a long; if necessary, a cast must be used. In this context, static variables are treated as if they were global variables.

Local variables ("auto") variables are allocated on the 65816 machine stack. The machine stack is a 16-bit register; the bank address of the stack is always bank 0. Thus the maximum stack size is limited to a theoretical 64K: in practice, this is considerably smaller, due to competing use of bank-0 memory by the system and other potentially resident programs. The start code initializes a default stack size of 8K; by creating a global integer variable named `_stack_size_` in your program, and initializing its value, you can define your own stack size (recognizing that the initialization code will fail if you specify a stack size larger than the memory manager can allocate in bank zero). For example, the following global declaration will cause the initialization code to allocate a 16K stack:

```
int _stack_size_ = 1024 * 16;
```

Storage for local variables is created dynamically on the stack upon function entry. If less than 256 bytes are required for parameter storage, internal temporary variables, and local variables, then all of the local variables will be addressed via direct page addressing, and pointer derefrencing using local variables will generally use indirect long addressing. If more than 256 bytes are required, the compiler will have to use indexed addressing to access variables that extend beyond the first 256 bytes of stack storage allocated. The first declared variables are the first allocated, so declaring your frequently-used local variables first will guarantee that the most efficient addressing modes will be used in referencing them.

Function calls are all made via long subroutine calls.

Chapter 5

The Standard C Library

About the Standard C Library

This chapter describes the Standard C Library provided with APW C. The Standard C Library is a collection of basic routines that let you read and write files, examine and manipulate strings, perform data conversion, acquire and release memory, and perform mathematical operations.

The chapter begins with an introduction to the error-number conventions used in the Standard C Library, followed by the library functions and macros arranged alphabetically under the name of the header file containing them. Each header file contains a group of related functions or macros. For example, both the `fread` and `fwrite` macros are found under the `fread` header. All of the function names and other identifiers used in Standard C Library routines are listed in Appendix D, "The Library Index." To find out where in this chapter a particular identifier is described, consult Appendix D.

❖ *Note:* Remember that identifiers in C are case sensitive and should be spelled exactly as shown in the synopsis. Filenames (as in `#include` statements) are not case-sensitive. By convention, they are written in uppercase.

The library routines under each header are documented as follows:

- *Synopsis* shows the code you need to add to your program when using these library routines and the files you need to include at compile time.
- *Description* discusses the library routines and their input and output.
- *Diagnostics* describes error conditions.
- *Return value* describes the values returned by the routines.
- *Example* contains examples of commands.
- *Note* contains remarks.
- *Warning* gives cautions.
- *See also* provides the names of other library routines or sections in this chapter related to the ones described in the current section.

Some of these will not be found under each header.

Note: Specific support for desk accessories has not been a consideration in the design of this library.

Error numbers

Synopsis

```
#include <ERRNO.H>

extern int errno;
```

Description

Many of the Standard C Library functions have one or more possible error returns. An otherwise meaningless return value, usually –1, indicates an error condition: see the descriptions of individual functions for details. The external variable `errno` also provides an error number. Variable `errno` is only valid immediately after a call; it is not cleared on successful calls, so it should be tested only if the return value indicates an error.

The error name appears in brackets following the text in a library function description. For example:

"The next attempt to write a nonzero number of bytes will signal an error. [ENOSPC]"

Not all possible error numbers are listed for each library function because many errors are possible for most of the calls. Some UNIX operating system error numbers do not apply to the Apple IIGS and are not documented in this manual. Some calls go to the Apple IIGS ROM and as a side effect return a value in `_toolErr` as well as the value in `errno`. Some calls, such as `printf` and `scanf`, may change these global variables even when they succeed.

Here is a list of the error numbers that can be returned in `errno` and their names as defined in the `<ERRNO.H>` file.

2 ENOENT *No such file or directory*
A file whose filename is specified does not exist or one of the directories in a pathname does not exist.

5 EIO *I/O error*
Some physical I/O error has occurred. This error may in some cases be signaled on a call following the one to which it actually applies.

6 ENXIO *No such device or address*
I/O on a special file refers to a subdevice that does not exist, or the I/O is beyond the limits of the device. This error may also occur when, for example, no disk is present in a drive.

9 EBADF *Bad file number*
Either a file descriptor does not refer to an open file, or a read (or write) request is made to a file that is open only for writing (or reading).

12 ENOMEM *Not enough space*
The system ran out of memory while the library call was executing.

13 EACCES *Permission denied*
An attempt was made to access a file in a way forbidden by the protection system.

14 EFAULT *Bad pathname*
A supplied pathname has incorrect syntax.

16 EBUSY *Device or resource busy*
Two or more online volumes have identical volume names.

17 EEXIST *File exists*
An existing file was mentioned in an inappropriate context; for example, `open(file, O_CREAT|O_EXCL)`.

19 ENODEV *No such device*
An attempt was made to apply an inappropriate system call to a device; for example, an attempt to read from a write-only device.

20 ENOTDIR *Not a directory*
An object that is not a directory was specified where a directory is required (for example, in a pathname prefix).

22 EINVAL *Invalid parameter*
Some invalid parameter was provided to a library function.

24 ENFILE *Too many open files*
The system cannot allocate memory to record another open file.

28 ENOSPC *No space left on device*
During a write to an ordinary file, there is no free space left on the device.

29 ESPIPE *Illegal seek*
An `lseek` was issued incorrectly.

30 EROFS *Read-only file system*
An attempt to modify a file or directory was made on a device mounted for read-only access.

45 ETXTBUSY *Text file busy*
An attempt has been made to perform a disallowed operation on an open file.

Note

Calls that interface to the Apple IIGS I/O system (such as `open`, `close`, `read`, `write`, and `ioctl`) can set the external variable `_toolErr` as well as `errno` on errors. This is a side effect: it is not safe to assume any relationship between the error number returned in `errno` and the number that may be returned in `_toolErr`. To detect errors in Standard C Library calls, use `errno`; to detect errors in Toolbox calls use `_toolErr`.

This section documents the values returned in `errno`. The Toolbox errors returned in `_toolErr` are documented in the System Error Handler chapter of the *Apple IIGS Toolbox Reference*.

abs—return integer absolute value

Synopsis

```
int abs(i)
  int i;
```

Description Function `abs` returns the absolute value of `i`.

Note The absolute value of the negative integer with the largest magnitude is undefined.

See also `floor`

atof—convert ASCII string to floating-point number

Synopsis

```
#include <MATH.H>

extended atof(str)
  char *str;
```

Description

Function `atof` converts a character string pointed to by `str` to an extended-precision floating-point number. The first unrecognized character ends the conversion. Function `atof` recognizes an optional string of white-space characters (spaces or tabs), then an optional sign, then a string of digits optionally containing a decimal point, then an optional *e* or *E* followed by an optionally signed integer. If the string begins with an unrecognized character, `atof` returns a NaN.

Function `atof` recognizes "`INF`" as infinity and "`NAN`" (optionally followed by parentheses that may contain a string of digits) as a NaN, with NaN code given by the string of digits. Case is ignored in the infinity and NaN string.

Diagnostics

Function `atof` honors the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

`scanf`

`str2dec`, `dec2num` in the *Apple Numerics Manual*

atoi—convert string to integer

Synopsis

```
#include <STDLIB.H>

int atoi(str)
  char *str;
long atol(str)
  char *str;
```

Description

The character string `str` is scanned up to the first nondigit character other than an optional leading minus sign (-). Leading white-space characters (spaces and tabs) are ignored.

A plus sign (+) is considered a nondigit character.

Return value

Function `atoi` returns as an integer the decimal value represented by `str`. Function `atol` returns as a long integer the decimal value represented by `str`.

Note

Overflow conditions are ignored.

See also

`atof`, `scanf`, `strtol`

close—close a file descriptor

Synopsis

```
int close(fildes)
  int fildes;
```

Description

Parameter `fildes` is a file descriptor obtained from an `open`, `creat`, `dup`, or `fcntl` call. Function `close` closes the file descriptor indicated by `fildes`.

Function `close` fails if `fildes` is not a valid open file descriptor. [EBADF]

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

See also

`creat`, `dup`, `fcntl`, `open`

conv—translate characters

Synopsis

```
#include <CTYPE.H>

int toupper(c)
  int c;
int tolower(c)
  int c;
int _toupper(c)
  int c;
int _tolower(c)
  int c;
int toascii(c)
  int c;
```

Description

Functions `toupper` and `tolower` have as their domain the set of ASCII characters (0 through 127) and the constant `EOF` (–1). If parameter `c` to `toupper` represents a lowercase letter, the result is the corresponding uppercase letter. If parameter `c` to `tolower` represents an uppercase letter, the result is the corresponding lowercase letter. All other parameters in the domain are returned unchanged.

Macros `_toupper` and `_tolower` produce the same results as functions `toupper` and `tolower` but have restricted domains and are faster. Macro `_toupper` requires a lowercase letter as its parameter; its result is the corresponding uppercase letter. Macro `_tolower` requires an uppercase letter as its parameter; its result is the corresponding lowercase letter. Parameters outside the domain cause undefined results.

Function `toascii` converts `c` by clearing all bits that are not part of a standard ASCII character. It is used for compatibility with other systems.

Note

These routines do not support the Apple IIGS extended character set (with values greater than 0x7F). For values outside the stated domain, the result is undefined.

See also

`ctype`, `getc`

creat—create a new file or rewrite an existing file

Synopsis

```
int  creat(filename)
  char  *filename;
```

Description

Function `creat` creates a new file or prepares to rewrite an existing file, `filename`. If the file exists, the length of its data fork is set to 0.

Function `creat(filename)` is equivalent to

```
open(filename,O_WRONLY|O_TRUNC|O_CREAT)
```

Upon successful completion, a nonnegative integer (the file descriptor) is returned and the file is open for writing. The file pointer is set to the beginning of the file. A maximum of about 30 files may be open at a given time; the actual maximum depends upon the current system environment.

Return value

Upon successful completion, a nonnegative integer (the file descriptor) is returned. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

Note

Other implementations of `creat` specify a second parameter, `mode`. This version ignores any second parameter.

See also

`close`, `open`

ctype—classify characters

Synopsis

```
#include <CTYPE.H>

int isalpha(c)
  int c;
int isupper(c)
  int c;
int islower(c)
  int c;
int isdigit(c)
  int c;
int isxdigit(c)
  int c;
int isalnum(c)
  int c;
int isspace(c)
  int c;
int ispunct(c)
  int c;
int isprint(c)
  int c;
int isgraph(c)
  int c;
int iscntrl(c)
  int c;
int isascii(c)
  int c;
```

Description

These macros classify character-coded integer values by table lookup, returning nonzero for `true`, zero for `false`. Macro `isascii` is defined on all integer values; the rest are defined only where `isascii` is true and on the single non-ASCII value EOF (–1).

Macro	Returns true if
`isascii`	c is an ASCII character code lower than octal 0200.
`isalpha`	c is a letter [A–Z] or [a–z].
`isupper`	c is an uppercase letter [A–Z].
`islower`	c is a lowercase letter [a–z]
`isdigit`	c is a digit [0–9].
`isxdigit`	c is a hexadecimal digit [0–9], [A–F], or [a–f].
`isalnum`	c is alphanumeric (letter or digit).
`isspace`	c is a space, tab, return, new line, vertical tab, or form feed.
`ispunct`	c is a punctuation character (neither control nor alphanumeric).
`isprint`	c is a printing character, space (octal 040) through tilde (octal 0176).

`isgraph`	`c` is a printing character, similar to `isprint` except false for space.
`iscntrl`	`c` is a delete character (octal 0177) or an ordinary control character (less than octal 040).

Warning If `c` is not in the domain of the function, the result is undefined.

Note These macros do not support the Apple IIGS extended character set. For values outside the domain, the result is undefined.

dup—duplicate an open file descriptor

Synopsis

```
int dup(fildes)
  int fildes;
```

Description

Parameter `fildes` is a file descriptor obtained from an `open`, `creat`, `dup`, or `fcntl` call. The new file descriptor returned by dup is the lowest one available.

The function call `dup(fildes)` is equivalent to

```
fcntl(fildes, F_DUPFD, 0)
```

Function dup fails if parameter `fildes` is not a valid open file descriptor. [EBADF]

Return value

Upon successful completion, a nonnegative integer (the file descriptor) is returned. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

See also

`close`, `fcntl`, `open`

ecvt—convert a floating-point number to a string

Synopsis

```
#include <MATH.H>

char *ecvt(value, ndigit, decpt, sign)
  extended value;
  int ndigit, *decpt, *sign;
char *fcvt(value, ndigit, decpt, sign)
  extended value;
  int ndigit, *decpt, *sign;
```

Description

Function `ecvt` converts `value` to a null-terminated string of `ndigit` digits and returns a pointer to this string as the function result. The low-order digit is rounded.

The decimal point is not included in the returned string. The position of the decimal point is indicated by `decpt`, which indirectly stores the position of the decimal point relative to the returned string. If the `int` pointed to by `decpt` is negative, the decimal point lies to the left of the returned string. For example, if the string is `"12345"` and `decpt` points to an `int` of 3, the value of the string is 123.45; if `decpt` points to –3, the value of the string is .00012345.

If the sign of the converted value is negative, the `int` pointed to by `sign` is nonzero; otherwise it is zero.

Function `fcvt` provides fixed-point output in the style of Fortran F-format output. Function `fcvt` differs from `ecvt` in its interpretation of `ndigit`:

- In `fcvt`, `ndigit` specifies the number of digits to the right of the decimal point.
- In `ecvt`, `ndigit` specifies the number of digits in the string.

Note

The string pointed to by the function result is static data whose contents are overwritten by each call. To preserve the value, copy it before calling the function again.

See also

`printf`

`str2dec`, `dec2num` in the *Apple Numerics Manual*

exit—terminate the current application

Synopsis

```
#include <STDLIB.H>

void exit(status)
  int status;
void _exit(status)
  int status;
```

Description

Functions `exit` and `_exit` close open file descriptors and terminate the application or tool. Here is the order in which `exit` performs its duties:

1. It executes all exit procedures in reverse order of their installation by `onexit`, including the exit procedures for the Standard I/O Package if routines from that package were used. All buffered files are flushed and closed.
2. It closes all open files that were opened with `open` or `fopen`.
3. If the program is a tool running under the APW Shell, the `exit` function returns status and control to the APW Shell by placing a return value in the lower three bytes of `status` and terminating the application.

Function `_exit` circumvents the exit procedures described in step 1 above. Use `_exit` instead of `exit` to abort your program when you are uncertain about the integrity of the data space.

Return value

The `main` program is a function that returns an integer. The return value of `main` is interpreted by the APW Shell as the program status. When you call `exit` or `_exit`, the `status` parameter is returned to the APW Shell as the return value for the application's main function: 0 for normal execution or a small positive value for errors (typically 1..3). `Main` programs that return to the shell without setting `status` to an integer value appear to be returning a random status.

There is no return from `exit` or `_exit`.

Note

Functions `exit` and `_exit` do not close files you opened with calls to the I/O routines documented in the *Apple IIGS Toolbox Reference*.

See also

`onexit`, `stdio`

exp—exponential, logarithm, power, square-root functions

Synopsis

```
#include <MATH.H>

extended exp(x)
  extended x;
extended log(x)
  extended x;
extended log10(x)
  extended x;
extended pow(x, y)
  extended x, y;
extended sqrt(x)
  extended x;
```

Description

Function `exp(x)` returns e^x, where e is the natural logarithm base.

Function `log(x)` returns the natural logarithm of x, $\log_e x$.

Function `log10(x)` returns the base-10 logarithm of x, $\log_{10} x$.

Function `pow(x, y)` returns x^y.

Function `sqrt(x)` returns the square root of x.

For special cases, these functions return a NaN or signed infinity as appropriate.

Diagnostics

These functions honor the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

`hypot`, `sinh`

Apple Numerics Manual

faccess—named-file access and control

Synopsis

```
#include <FCNTL.H>

int faccess(filename, cmd, arg)
  char *filename;
  unsigned int cmd;
  char *arg;
```

Description

Function `faccess` provides access to control and status information for named files. (Compare function `ioctl`, which provides different control and status information for open files.)

Parameter cmd must be set to one of the constants in the following list to indicate what operation is to be performed on the file. As noted in the list, some calls to `faccess` also require the `arg` parameter, usually as a pointer to a `char`.

The following commands are available to all programs:

Value of cmd	Description
`F_DELETE`	Deletes the named file, or returns an error if the file is open. Parameter `arg` is ignored.
`F_RENAME`	Renames the named file. Parameter `arg` is a pointer to a string containing the new name.
`F_TYPE`	Sets the type of the file to the value of the parameter `arg`.
`F_AUX`	Sets the auxiliary type of the file to the value of the parameter `arg`.
`F_STAT`	Gets the directory entry information for the file `filename`, and puts the information in the `struct DirectoryEntry` pointed to by `arg`.

For example, `faccess(thing, F_TYPE, 0x04)` sets the type of file "thing" to $04–ASCII text file.

Return value

Upon successful completion, `faccess` returns a nonnegative value, usually 0. If the device for the named file cannot perform the requested command, `faccess` returns –1 and `errno` is set to indicate the error.

Note

The cmd value `F_OPEN` is reserved for operating system use.

See also

`ioctl`, `unlink`

fclose—close or flush a stream

Synopsis

```
#include <STDIO.H>

int fclose(stream)
  FILE *stream;
int fflush (stream)
  FILE *stream;
```

Description

Function `fclose` closes a file that was opened by `fopen`, `freopen`, or `fdopen`. Function `fclose` causes any buffered data for `stream` to be written out, and the buffer (if one was allocated by the system) is released; `fclose` then calls `close` to close the file descriptor associated with `stream`. The value of the parameter `stream` cannot be used unless reassigned with `fopen`, `fdopen`, or `freopen`.

Function `fclose` fails if the file descriptor associated with `stream` is already closed. [`ENOENT`]

Function `fclose` is performed automatically for all open `FILE` streams upon calling `exit`.

Function `fflush` causes any buffered data for `stream` to be written out; `stream` remains open.

Return value

These functions return 0 if the operation succeeded or `EOF` if an error was detected (such as trying to write to a file that has not been opened for writing).

See also

`close`, `exit`, `fopen`, `setbuf`

fcntl—file control

Synopsis

```
#include <FCNTL.H>

int fcntl(fildes, cmd, arg)
  int fildes;
  unsigned int cmd;
  int arg;
```

Description

Function `fcntl` is used for duplicating file descriptors. A file remains open until all of its file descriptors are closed.

Parameter `fildes` is an open file descriptor obtained from an `open`, `creat`, `dup`, or `fcntl` call. Parameter `cmd` takes the value `F_DUPFD`, which tells `fcntl` to return the lowest numbered available file descriptor greater than or equal to `arg`. Normally `arg` is greater than or equal to 3, to avoid obtaining the standard file descriptors 0, 1, and 2. Function `fcntl` returns a new file descriptor that points to the same open file as `fildes`. The new file descriptor has the same access mode (read, write, or read/write) and file pointer as `fildes`. Any I/O operation changes the file pointer for all file descriptors that share it.

Function `fcntl` fails if one or more of the following are true:

- Parameter `fildes` is not a valid open file descriptor. [`EBADF`]
- Parameter `arg` is negative or greater than the highest allowable file descriptor. [`EINVAL`]

Return value

Upon successful completion, the value returned is a new file descriptor. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

Note

The `F_GETFD`, `F_SETFD`, `F_GETFL`, and `F_SETFL` commands of `fcntl` are not supported on the Apple IIGS.

See also

`close`, `dup`, `open`

ferror—stream status inquiries

Synopsis

```
#include <STDIO.H>

int feof(stream)
  FILE *stream;
int ferror(stream)
  FILE *stream;
void clearerr(stream)
  FILE *stream;
int fileno(stream)
  FILE *stream;
```

Description

Macro `feof` returns nonzero when an end-of-file condition has previously been detected reading the named input stream; otherwise, it returns zero.

Macro `ferror` returns nonzero when an I/O error has previously occurred reading from or writing to the named stream; otherwise, it returns zero.

Macro `clearerr` resets the error indicator and end-of-file indicator to zero on the named stream.

Macro `fileno` returns the integer file descriptor associated with the named stream; see open.

See also

open, fopen

floor—floor, ceiling, mod, absolute value functions

Synopsis

```
#include <MATH.H>

extended floor(x)
  extended x;
extended ceil(x)
  extended x;
extended fmod(x, y)
  extended x, y;
extended fabs(x)
  extended x;
```

Description

Function `floor(x)` returns the largest integer (as an extended-precision number) not greater than `x`.

Function `ceil(x)` returns the smallest integer not less than `x`.

Whenever possible, `fmod(x, y)` returns the number f with the same sign as `x`, such that `x` = i `y` + f for some integer i, and $|f| < |$`y`$|$. If `y` is 0, `fmod` returns a NaN.

Function `fabs(x)` returns $|$`x`$|$, the absolute value of `x`.

See also

`abs`

`rint`, `setround` in the *Apple Numerics Manual*

fopen—open a buffered file stream

Synopsis

```
#include <STDIO.H>

FILE *fopen(filename, type)
  char *filename, *type;
FILE *freopen(filename, type, stream)
  char *filename, *type;
  FILE *stream;
FILE *fdopen(fildes, type)
  int fildes;
  char *type;
```

Description

Function `fopen` opens the file named by `filename` and associates a stream with it. Function `fopen` returns a pointer to the `FILE` structure associated with the stream.

Parameter `filename` points to a character string that contains the name of the file to be opened.

The value of `type` should be one of the string values in the first column in the following table, including the quotes. The headings Open Mode Used and Description explain how `type` is used. For more information, see `open`.

Value	Open mode used	Description
`"r"`	`O_RDONLY`	Open for reading only.
`"w"`	`O_WRONLY|O_CREAT|O_TRUNC`	Truncate or create for writing.
`"a"`	`O_WRONLY|O_CREAT|O_APPEND`	Append: open for writing at end of file, or create for writing.
`"r+"`	`O_RDWR`	Open for update (reading and writing).
`"w+"`	`O_RDWR|O_CREAT|O_TRUNC`	Truncate or create for update.
`"a+"`	`O_RDWR|O_CREAT|O_APPEND`	Append: open or create for update at end of file.

When a file is written to a device, normally certain characters are translated to match the needs of the device or the expectations of ProDOS for a normal text file (such as tr anslating `\n` to `CR` rather than `LF`). The following values, with `b` added to the string, suppress such translations:

Value	Open mode used	Description
`"rb"`	`O_RDONLY|O_BINARY`	Open for reading only.
`"wb"`	`O_WRONLY|O_CREAT|O_TRUNC|O_BINARY`	Truncate or create for writing.
`"ab"`	`O_WRONLY|O_CREAT|O_APPEND|O_BINARY`	Append: open for writing at end of file, or create for writing.

`"rb+"`	`O_RDWR|O_BINARY`	Open for update (reading and writing).
`"wb+"`	`O_RDWR|O_CREAT|O_TRUNC|O_BINARY`	Truncate or create for update.
`"ab+"`	`O_RDWR|O_CREAT|O_APPEND|O_BINARY`	Append: open or create for update at end of file.

Note: The `b` and the `+` can be reversed.

Function `freopen` substitutes the named file for the open stream. The original stream is closed, regardless of whether the open operation ultimately succeeds. Function `freopen` returns a pointer to the `FILE` structure associated with `stream`. Function `freopen` is typically used to attach the previously opened streams associated with `stdin`, `stdout`, and `stderr` to other files.

Function `fdopen` associates a stream with a file descriptor by formatting a file structure from the file descriptor. Thus, `fdopen` can be used to access the file descriptors returned by `open`, `creat`, `dup`, or `fcntl`. (These calls return file descriptors, not pointers to a `FILE` structure.) The type of stream must agree with the mode of the open file.

When a file is opened for update, both input and output may be done on the resulting stream. However, output may not be directly followed by input without an intervening `fseek` or `rewind`, and input may not be directly followed by output without an intervening `fseek`, `rewind`, or an input operation that encounters an end-of-file condition.

When a file is opened for append (that is, when `type` is `a` or `a+`), it is impossible to overwrite information already in the file. Function `fseek` may be used to reposition the file pointer to any position in the file, but when output is written to the file the current file pointer is disregarded. All output is written at the end of the file and causes the file pointer to be repositioned at the end of the output.

Return values If they succeed, the functions `fopen`, `freopen`, and `fdopen` return a valid file pointer. If they fail, they return `NULL`.

The maximum number of open `FILE` streams is `NFILE` (defined in `STDIO.H`, currently 20). The maximum number of open disk files may be less than `NFILE`, as determined by the current release of ProDOS.

Note The parameter type must have one of the values in the first column in the table; do not use values intended for `open`, such as `O_RDONLY`.

See also `open`, `fclose`, `fseek`

fread—binary input/output

Synopsis

```
#include <STDIO.H>

int fread(ptr, size, nitems, stream)
  char *ptr;
  int size, nitems;
  FILE *stream;
int fwrite(ptr, size, nitems, stream)
  char *ptr;
  int size, nitems;
  FILE *stream;
```

Description

Function `fread` copies `nitems` items of data from the named input stream into an array beginning at `ptr`. An item of data is a sequence of `size` bytes (not necessarily terminated by a null byte). Function `fread` stops appending bytes if an end-of-file or error condition is encountered while reading `stream` or if `nitems` items have been read. Function `fread` leaves the file pointer in `stream` pointing to the byte following the last byte read.

Function `fwrite` writes at most `nitems` items of data to the named output stream from the array pointed to by `ptr`. An item is a sequence of size bytes. Function `fwrite` stops writing when it has written `nitems` items of data or if an error condition is encountered on `stream`. Function `fwrite` does not change the contents of the array pointed to by `ptr`.

The parameter `size` is typically

```
sizeof(*ptr)
```

where `sizeof` specifies the length of an item pointed to by `ptr`. If `ptr` points to a data type other than `char`, it should be cast into a pointer to `char`.

Return values

Functions `fread` and `fwrite` return the number of items read or written. If `nitems` is 0 or negative, no characters are read or written and 0 is returned by both `fread` and `fwrite`.

See also

`fopen`, `getc`, `gets`, `printf`, `putc`, `puts`, `read`, `scanf`, `stdio`, `write`

frexp—manipulate parts of floating-point numbers

Synopsis

```
#include <MATH.H>

extended frexp(value, eptr)
  extended value;
  int *eptr;
extended ldexp(value, exp)
  extended value;
  int exp;
extended modf(value, iptr)
  extended value, *iptr;
```

Description

Every nonzero number can be written uniquely as $x * 2^n$, where the mantissa (fraction) x is in the range $0.5 \leq |x| < 1.0$ and the exponent n is an integer. Function `frexp` returns the mantissa of an `extended` value and stores the exponent indirectly in the location pointed to by `eptr`. Note that the mantissa here differs from the significand described in the *Apple Numerics Manual*, whose normal values are in the range $1.0 \leq |x| < 2.0$.

Function `ldexp` returns the quantity $value * 2^{exp}$.

Function `modf` returns the signed fractional part of `value` and stores the integral part indirectly in the location pointed to by `iptr`.

Diagnostics

Function `ldexp` honors the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

`logb`, `scalb` in the *Apple Numerics Manual*

fseek—reposition a file pointer in a stream

Synopsis

```
#include <STDIO.H>

int fseek(stream, offset, ptrname)
  FILE *stream;
  long offset;
  int ptrname;
void rewind(stream)
  FILE *stream;
long ftell(stream)
  FILE *stream;
```

Description

Function `fseek` sets the position of the next input or output operation on the stream. The new position is `offset` bytes from the beginning, the current position, or the end of the file, when the value of `ptrname` is 0, 1, or 2, respectively. If `ptrname` is 1 or 2, `offset` may be negative.

The call

```
rewind(stream)
```

is equivalent to

```
fseek(stream, 0L, 0)
```

except that no value is returned.

Functions `fseek` and `rewind` undo any effects of `ungetc`.

After `fseek` or `rewind`, the next operation on a file opened for update may be either input or output.

Function `ftell` returns the offset of the current byte relative to the beginning of the file associated with the named stream.

Diagnostics

Function `fseek` returns nonzero for improper seeks; otherwise it returns zero. An example of an improper seek is an `fseek` before the beginning of the file.

See also

`lseek`, `fopen`, `ungetc`

getc—get a character or a word from a stream

Synopsis

```
#include <STDIO.H>

int getc(stream)
  FILE *stream;
int getchar()
int fgetc(stream)
  FILE *stream;
int getw(stream)
  FILE *stream;
```

Description

Macro `getc` returns the next character from the named input stream. It also moves the file pointer, if defined, ahead one character in `stream`. Macro `getc` cannot be used if a function is necessary; for example, you cannot have a function pointer point to it. Macro `getc` returns the integer `EOF` on end of file or error.

Macro `getchar` returns the next character from the standard input stream, `stdin`.

Function `fgetc` produces the same result as macro `getc`; function `fgetc` runs more slowly than macro `getc` but takes less space per invocation. You can also have a pointer to `fgetc` but not to `getc`.

Function `getw` returns the next `int` (that is, two bytes) from the named input stream so that the order of bytes in the stream corresponds to the order of bytes in memory. Function `getw` returns the constant `EOF` upon encountering an end-of-file or error condition. Because `EOF` is a valid integer value, `feof` and `ferror` should be used to check the success of `getw`. Function `getw` increments the associated file pointer, if defined, to point to the next `int`. Function `getw` assumes no special alignment in the file.

Return values

These calls return either data from the stream or the integer constant `EOF` (–1) on end of file or error condition.

Note

Because it is implemented as a macro, `getc` treats a stream parameter with side effects incorrectly. In particular,

```
getc(*f++)
```

doesn't work as you would expect. Instead use

```
fgetc(*f++)
```

See also

`ferror`, `fopen`, `fread`, `gets`, `iocntl`, `scanf`, `stdio`

getenv—access exported APW Shell variables

Synopsis

```
#include <STDLIB.H>

char *getenv(varname)
  char *varname;
```

Description

The **environment** is the set of exported variables provided by the APW Shell. Function `getenv` provides access to variables in this set. (See the Variables section in Chapter 4 of the *Apple IIGS Programmer's Workshop Reference* for the list of standard exported shell variables.)

Function `getenv` searches the environment for a shell variable with the name specified by `varname` and returns a pointer to the character string containing its value. The null pointer is returned if the shell variable is not defined or has not been exported. The shell-variable name search is case insensitive.

Return value

Upon successful completion, a pointer to the value of `varname` is returned. If the shell variable is not defined or not exported, the function returns the null pointer.

For standalone applications, which do not run under the APW shell, `getenv` always returns the null pointer.

Note

The environment can also be accessed by means of a parameter to the C main-entry-point function `main` if the main procedure is declared as

```
main(argc, arv, envp)
```

The `envp` array represents the set of APW shell variables that have been made available to tools by means of the APW `EXPORT` command. The `i`th `envp` entry has the form

```
envp[i] = "varname\0varvalue\0";
```

The last `envp` entry is the null pointer.

If you use `envp` to search the environment, be sure to use case-insensitive string comparisons.

Warning

Function `getenv` returns a pointer to the place in memory where a copy of the APW shell variable resides. Do not modify the value of a shell variable in such a way as to increase its length.

gets—get a string from a stream

Synopsis

```
#include <STDIO.H>

char *gets(str)
  char *str;
char *fgets(str, maxlen, stream)
  char *str;
  int maxlen;
  FILE *stream;
```

Description

Function `gets` reads characters from the standard input stream `stdin` into the array pointed to by `str` until a newline character is read or an end-of-file condition is encountered. The newline character is discarded, and the string is terminated with a null (`\0`) character.

Function `fgets` reads characters from `stream` into the array pointed to by `str` until `maxlen`–1 characters are read, a newline character is read and transferred to `str`, or until an end-of-file condition is encountered. The string is then terminated with a null character.

Return values

If the end-of-file is encountered and no characters have been read, no characters are transferred to `str` and NULL is returned. If a read error occurs, NULL is returned. Otherwise `str` is returned. (A read error will occur, for example, if you attempt to use these functions on a file that has not been opened for reading.)

Note

The array pointed to by `str` is assumed to be large enough; overflow is not checked.

The function `gets` omits the newline character in the string; `fgets` leaves it in.

See also

`ferror`, `fopen`, `fread`, `getc`, `scanf`, `stdio`

hypot—Euclidean distance function

Synopsis

```
#include <MATH.H>

extended hypot(x, y)
  extended x, y;
```

Description

Function hypot returns

```
sqrt (x * x + y * y)
```

taking precautions against unwarranted overflows.

Diagnostics

Function hypot honors the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

exp

Apple Numerics Manual

ioctl—control a device

Synopsis

```
#include <IOCTL.H>

int ioctl(fildes, cmd, arg)
  int fildes;
  unsigned int cmd;
  long *arg;
```

Description

Function `ioctl` communicates with a file's device handler by sending control information, requesting status information, or both. Parameter `cmd` indicates which device-specific operations `ioctl` must perform. Here are the control values:

Value of cmd	Description
`FIOINTERACTIVE`	Function `ioctl` returns 0 if the device is interactive; if not, it returns –1 and `errno` is set to `EINVAL`. Parameter `arg` is ignored.
`FIOBUFSIZE`	Function `ioctl` returns, in bytes, the optimal buffer size for this device; the buffer size is returned in a `long` pointed to by `arg`. If the device has no default buffer size, `ioctl` returns –1 and `errno` is set to `EINVAL`.
`FIOREFNUM`	Function `ioctl` returns the Apple IIGS file reference number associated with `fildes`; the reference number is returned in the `short` pointed to by `arg`. If the `fildes` is not open on a Apple IIGS file (such as the console device), `ioctl` returns –1.
`FIOSETEOF`	Function `ioctl` sets the logical end-of-file specified in the `long` parameter `arg`. The value of `arg` is the new size of the file, in bytes. This command can be used to reduce or increase the size of the open file. The current file pointer is not affected unless the file size is set below it.
`FIO_STAT`	Function `ioctl` stores the directory information associated with `fildes` into the `struct DirectoryEntry` pointed to by `arg`.

Function `ioctl` fails if one or both of the following conditions exist:

- File descriptor `fildes` is not valid or is not open. [`EBADF`]
- Parameters `cmd` or `arg` are not valid for the device handler associated with `fildes`. [`EINVAL`]

Diagnostics If an error has occurred, a value of –1 is returned and `errno` is set to indicate the error.

Note For cmd values `FIOINTERACTIVE` and `FIOBUFSIZE`, a function return of –1 is a meaningful response, not an error. For `FIOINTERACTIVE`, `errno` is set to `EINVAL` for devices that are not interactive. For `FIOBUFSIZE`, `errno` is set to `EINVAL` for devices that have no default buffering.

The cmd values `FIOLSEEK` and `FIODUPFD` are reserved for operating system use.

Warning `FIOREFNUM` lets you do Apple IIGS I/O operations such as `Allocate` that are not available through `ioctl`. Do not close or modify the file pointer using the reference number.

See also `fcntl`

lseek—move read/write file pointer

Synopsis

```
#include <FCNTL.H>

long lseek(fildes, offset, whence)
  int fildes;
  long offset;
  int whence;
```

Description

A file descriptor, `fildes`, is returned from a call to `creat`, `dup`, `fcntl`, or `open`. Function `lseek` sets the file pointer associated with `fildes` as follows:

- If `whence` is 0, the pointer is set to `offset` bytes.
- If `whence` is 1, the pointer is set to its current location plus `offset`. (The value of `offset` may be negative, zero, or positive.
- If `whence` is 2, the pointer is set to the size of the file plus `offset`. (The value of `offset` may be negative or zero.

Upon successful completion, the file pointer value as measured in bytes from the beginning of the file is returned.

The file pointer remains unchanged and `lseek` fails if one or more of the following are true:

- File descriptor `fildes` is not open. [EBADF]
- Parameter `whence` is not 0, 1, or 2. [EINVAL]
- The resulting file pointer would point before the beginning of the file. [EINVAL]

Some devices are incapable of seeking. The value of the file pointer associated with such a device is undefined.

Return value

Upon successful completion, a nonnegative long integer indicating the file-pointer value is returned. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

Note

In previous versions of the Standard C Library, `tell(fildes)` was a function that returned the current file position. It is equivalent to the call

```
lseek(fildes, 0L, 1)
```

Warning

Function `lseek` has no effect on a file opened with the `O_APPEND` flag because the next write to the file always repositions the file pointer to the end before writing.

See also `fseek`, `open`

malloc—memory allocator

Synopsis

```
#include <MALLOC.H>

char *malloc(size)
  unsigned int      size;
void free(ptr)
  char *ptr;
char *realloc(ptr, size)
  char *ptr;
  unsigned int size;
char *calloc(nelem, elsize)
  unsigned int nelem, elsize;
void cfree(ptr, nelem, elsize)
  char *ptr;
  unsigned int nelem, elsize;
```

Description

Functions `malloc` and `free` provide a simple general-purpose memory-allocation package. The storage area expands as necessary when `malloc` is called.

Function `malloc` allocates the first sufficiently large contiguous free space it finds and returns a pointer to a block of at least `size` bytes suitably aligned for any use. It calls `NewHandle` (see the *Apple IIGS Toolbox Reference*) to get more memory from the system when there is no suitable space already free.

Function `free` takes a parameter that is a pointer to a block previously allocated by `malloc`. If its size is greater than 2K bytes, it is returned to the system using `DisposeHandle`. Blocks smaller than that are cached by `malloc` for further allocation by `malloc` only. Undefined results occur if the space assigned by `malloc` is overrun or if a random value is passed to `free`.

Function `realloc` changes the size of the block pointed to by `ptr` to `size` bytes and returns a pointer to the (possibly moved) block. The contents are unchanged up to the lesser of the new and old sizes. If no free block of `size` bytes is available in the storage area, `realloc` asks `malloc` to enlarge the storage area by `size` bytes and then moves the data to the new space. If `ptr` is `NULL`, `realloc` is equivalent to `malloc`.

Function `calloc` allocates space for an array of `nelem` elements of size `elsize`. The space is initialized to zeros.

Function `cfree`, like `free`, frees memory allocated by `calloc`; `cfree` is included for compatibility with other systems. Parameters `nelems` and `elsize` are ignored.

Diagnostics Functions `malloc`, `realloc`, and `calloc` return NULL if there is no available memory or if the storage area has been detectably corrupted by a program's storing data outside the bounds of a block. When this happens, the block pointed to by `ptr` may have been destroyed.

memory—memory operations

Synopsis

```
#include <MEMORY.H>

char *memccpy(dest, source, c, n)
  char *dest, *source;
  int c, n;
char *memchr(source, c, n)
  char *source;
  int c, n;
int memcmp(a, b, n)
  char *a, *b;
  int n;
char *memcpy(dest, source, n)
  char *dest, *source;
  int n;
char *memset(dest, c, n)
  char *dest;
  char c;
  int n;
```

Description

These functions operate efficiently on memory areas (arrays of characters bounded by a count, not terminated by a null character). They do not check for the overflow of any receiving memory area.

Function memccpy copies characters from memory area source into dest, stopping after the first occurrence of character c has been copied or after n characters have been copied, whichever comes first. It returns either a pointer to the character after the copy of c in dest or NULL if c was not found in the first n characters of source.

Function memchr returns either a pointer to the first occurrence of character c in the first n characters of memory area source or NULL if c does not occur.

Function memcmp compares its parameters, a and b, looking at the first n characters only. It returns an integer less than, equal to, or greater than 0, depending on whether a is less than, equal to, or greater than b.

Function memcpy copies n characters from memory area source to dest. It returns dest.

Function memset sets the first n characters in memory area dest to the value of character c. It returns dest.

Warning

Overlapping moves yield unexpected results.

See also `string`

`BlockMove` in the Toolbox Reference Manual

onexit—install a function to be executed at program termination

Synopsis

```
int onexit(func);
  void (*func)();
```

Description

Function `onexit` installs the `exit` function pointed to by `func` by adding it to a list. The list is initially empty. A list entry is added whenever `onexit` is called. Function `exit` calls the functions in the list in the reverse order in which they were added. To ensure that buffers are flushed at program termination, the Standard I/O Package adds its cleanup function to the list the first time it allocates a buffer. Each function in the list is called without parameters either at program termination or when `exit` is called.

The number of user-supplied exit functions is limited to five.

Diagnostics

The function returns a nonzero value if the installation fails.

Note

A call to `_exit` circumvents user exit procedures installed by `onexit`.

Warning

If a function is installed more than once, its behavior is undefined.

See also

`exit`, `stdio`

open—open for reading or writing

Synopsis

```
#include <FCNTL.H>
int open(filename, oflag)
  char *filename;
  int oflag;
```

Description

Parameter `filename` is a filename or pseudo-filename (such as `.NULL`). Function open opens a file descriptor for the named file and sets the file-status flags according to the value of `oflag`. The value of `oflag` is constructed by OR-ing flag settings; for example,

```
fildes = open("MyFile", O_WRONLY|O_CREAT|O_TRUNC);
```

To construct `oflag`, first select one of the following access modes:

- `O_RDONLY` Open for reading only.
- `O_WRONLY` Open for writing only.
- `O_RDWR` Open for reading and writing.

Then optionally add one or more of these modifiers:

- `O_APPEND` The file pointer is set to the end-of-file before each write.
- `O_CREAT` If the file does not exist, it is created.
- `O_TRUNC` If the file exists, its length is truncated to 0; the mode and owner are unchanged.

The following setting is valid only if `O_CREAT` is also specified:

- `O_EXCL` Function open fails if the file exists.

When a file is written to a device, normally certain characters are translated to match the needs of the device or the expectations of ProDOS for a normal text file (such as translating \n to CR rather than LF). The following flag suppresses such translation.

- `O_BINARY` The file is read or written verbatim, suppressing the device driver's conversions.

Upon successful completion, a nonnegative integer (the file descriptor) is returned. The file pointer used to mark the current position within the file is set to the beginning of the file.

The named file is opened unless one or more of the following are true:

- `O_CREAT` is not set and the named file does not exist. [`ENOENT`]
- More than about 30 file descriptors are currently open. The actual limit varies according to run-time conditions. [`ENFILE`]
- `O_CREAT` and `O_EXCL` are set and the named file exists. [`EEXIST`]

Return value Upon successful completion, a nonnegative integer (the file descriptor) is returned. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

See also `close`, `creat`, `lseek`, `read`, `write`

printf—print formatted output

Synopsis

```
#include <STDIO.H>

int printf(format [ , arg ] ... )
  char *format;
int fprintf(stream, format [ , arg ] ... )
  FILE *stream;
  char *format;
int sprintf(str, format [ , arg ] ... )
  char *str, *format;
```

Description

Function `printf` places formatted output on the standard output stream `stdout`. Function `fprintf` places formatted output on the named output stream `stream`. Function `sprintf` places formatted output, followed by the null character (`\0`), into the character array pointed to by `str`; it's your responsibility to ensure that enough room is available. Each function returns the number of characters transmitted (not including the `\0` in the case of `sprintf`), or a negative value if an output error was encountered.

Each of these functions converts, formats, and prints its `arg` parameters under control of the `format` parameter. The `format` parameter is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in fetching zero or more `arg` parameters. The behavior of the function is undefined if there are insufficient `arg` parameters for the format. If the format is exhausted while `arg` parameters remain, the extra `arg` parameters are ignored.

Each conversion specification is introduced by the character `%`. After `%`, the following appear in sequence:

1. Zero or more flag characters, which modify the meaning of the conversion specification.
2. An optional decimal digit string specifying a minimum field width. If the converted value has fewer characters than the field width, it will be padded to the field width on the left (default) or right (if the left-adjustment flag has been given): see the discussion of the flag specification, below.
3. A precision that gives the minimum number of digits to appear for the `d`, `o`, `u`, `x`, or `X` conversions; the number of digits to appear after the decimal point for the `e`, `E`, and `f` conversions; the maximum number of significant digits for the `g` and `G` conversions; or the maximum number of characters to be printed from a string in the `s` conversion. The format of the precision is a period (`.`) followed by a decimal digit string; a null digit string is treated as zero.

4. An optional `l` specifying that a following `d`, `o`, `u`, `x`, or `X` conversion character applies to an `arg` parameter of type `long`.

5. A character that indicates the type of conversion to be applied.

A field width or precision may be indicated by an asterisk (*) instead of a digit string. In this case, an integer `arg` parameter supplies the field width or precision. The `arg` parameter that is actually converted is not fetched until the conversion letter is seen; therefore, the `arg` parameters specifying field width or precision must appear immediately before the `arg` parameter (if any) to be converted.

These are the flag characters and their meanings:

-	The result of the conversion will be left justified within the field.
+	The result of a signed conversion always begins with a sign (+ or -).
`blank`	If the first character of a signed conversion is not a sign, a space will be prefixed to the result. This implies that if the `blank` and + flags both appear, the `blank` flag will be ignored.
#	The value is to be converted to an alternate form. For `c`, `d`, `s`, and `u` conversions, the flag has no effect. For `o` conversion, it increases the precision to force the first digit of the result to be a zero. For `x` (`X`) conversion, a nonzero result will have `0x` (`0X`) prefixed to it. For `e`, `E`, `f`, `g`, and `G` conversions, the result will always contain a decimal point, even if no digits follow the point. (Normally, a decimal point appears in the result of these conversions only if a digit follows it.) For `g` and `G` conversions, trailing zeros in the fractional part will not be removed from the result (as they normally are).

The conversion characters and their meanings are these:

`d`, `o`, `u`, `x`, `X`	The integer `arg` parameter is converted to signed decimal (`d`), unsigned octal (`o`), unsigned decimal (`u`), or unsigned hexadecimal notation (`x` and `X`), respectively; the letters `abcdef` are used for `x` conversion and the letters `ABCDEF` for `X` conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is a null string.

`f`	The `float`, `double`, `comp`, or `extended` `arg` parameter is converted to decimal notation in the form "[-]*ddd.ddd*", where the number of digits after the decimal point is equal to the precision specification. If the precision is missing, it is assumed to be 6; if the precision is explicitly 0, no decimal point appears. Infinities are printed in the form "[-]`INF`", and NaNs are printed in the form "[-]`NAN`(*ddd*)", where *ddd* is a code indicating why the result is not a number.
`e, E`	The `float`, `double`, `comp`, or `extended` `arg` parameter is converted in the form "[-]d.*ddd*e±*dd*", where there is one digit before the decimal point and the number of digits after it is equal to the precision. When the precision is missing, it is assumed to be 6; if the precision is 0, no decimal point appears. The `E` format code produces a number with `E` instead of `e` introducing the exponent. The exponent always contains at least two digits. Infinities are printed as `INF` and NaNs are printed in the form "[-]`NAN`(*ddd*)", where *ddd* is a code indicating why the result is not a number.
`g, G`	The `float`, `double`, `comp`, or `extended` `arg` parameter is printed in style `f` or `e` (or in style `f` or `E` in the case of a `G` format code), with the precision specifying the number of significant digits. The style used depends on the value converted: style `e` is used only if the exponent resulting from the conversion is less than –4 or greater than the precision. Trailing zeros are removed from the result. A decimal point appears only if it is followed by a digit.
`c`	The character `arg` parameter is printed.
`s`	The `arg` parameter is taken to be a string (character pointer) and characters from the string are printed until a null character (`\0`) is encountered or the number of characters indicated by the precision specification is reached. If the precision is missing, it is taken to be infinite, with the result that all characters up to the first null character are printed. If the string pointer `arg` parameter has the value zero, the result is undefined; a zero `arg` parameter yields undefined results.
`%`	Print a `%`; no parameter is converted.

In no case does a nonexistent or small field width cause truncation of a field. If the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result. Characters generated by `printf` and `fprintf` are printed as if `putc` had been called.

Examples

To print a date and time in the form "Sunday, July 3, 10:02", where weekday and month are pointers to null-terminated strings, use this:

```
printf("%s, %s %d, %.2d:%.2d", weekday, month, day, hour, min);
```

To print pi to five decimal places, use this:

```
printf("pi = %.5f", pi());
```

Note

Calling `sprintf` causes other Standard I/O functions to be loaded, even though `sprintf` doesn't perform any I/O.

See also

`dec2str`, `ecvt`, `num2dec`, `putc`, `scanf`, `stdio`

putc—put character or word on a stream

Synopsis

```
#include <STDIO.H>

int putc(c, stream)
  char c;
  FILE *stream;
int putchar(c)
  char c;
int fputc(c, stream)
  char c;
  FILE *stream;
int putw(w, stream)
  int w;
  FILE *stream;
```

Description

Macro `putc` writes the character `c` to the output stream at the current position of the file pointer. Macro `putchar(c)` is equivalent to

```
putc(c, stdout)
```

Function `fputc` behaves like macro `putc`. Function `fputc` runs more slowly than macro `putc` but takes less space per invocation.

Function `putw` writes an `int` (that is, two bytes) to the output stream at the current position of the file pointer. This function neither assumes nor causes special alignment in the file.

For information about buffering of output files, see the `stdio` page.

Return values

When `putc`, `putchar`, `fputc`, or `putw` succeeds, it returns the value they have written. When one of these fails, it returns the constant `EOF`. This occurs if the file stream is not open for writing or if the output file cannot be grown. When `putw` succeeds, it returns 0; when it fails, it returns a nonzero value.

Note

Because it is implemented as a macro, `putc` treats a stream parameter with side effects incorrectly. In particular,

```
putc(c, *f++)
```

produces unexpected results. Instead, use

```
fputc(c, *f++)
```

See also

`fclose`, `ferror`, `fopen`, `fread`, `getc`, `printf`, `puts`, `setbuf`, `stdio`

puts—write a string to a stream

Synopsis

```
#include <STDIO.H>

int puts(str)
  char *str;
int fputs(str, stream)
  char *str;
  FILE *stream;
```

Description

Function `puts` writes the null-terminated string pointed to by `str`, followed by a newline character, to the standard output stream `stdout`.

Function `fputs` writes the null-terminated string pointed to by `str` to the named output stream `stream`.

Neither function writes the terminating null character.

Return value

Both routines return the number of characters written, or return `EOF` if there is a write error.

Note

Function `puts` appends a newline character, while `fputs` does not.

See also

`ferror`, `fopen`, `fread`, `printf`, `putc`, `stdio`

qsort—quicker sort

Synopsis

```
void qsort(base, nelem, elsize, compar)
  char *base;
  unsigned int nelem, elsize;
  int (*compar)();
```

Description

Function `qsort` is an implementation of the quicker-sort algorithm. It sorts a table of data in place.

Parameter `base` points to the element at the base of the table. Parameter `nelem` is the number of elements in the table. Parameter `elsize` is the size of an element in the table; it can be specified as `sizeof(*base)`.

Parameter `compar` is a pointer to a comparison function that you supply. Function `qsort` calls your comparison function with pointers to two elements being compared. Here is a sample declaration for your comparison function:

```
int myCompare(elem1, elem2)
      char *elem1, *elem2;
```

Your comparison function supplies the result of the comparison to `qsort` by returning one of the following integer values:

Result	Meaning
<0	The first parameter is less than the second parameter.
0	The first parameter is equal to the second parameter.
>0	The first parameter is greater than the second parameter.

Note

Parameter `base`, the pointer to the base of the table, should be of the pointer-to-element type and cast to `(char *)`.

rand—a simple random-number generator

Synopsis

```
int rand()
void srand(seed)
  unsigned seed;
```

Description

Function `rand` uses a multiplicative congruential random-number generator with period 2^{32} that returns successive pseudorandom numbers in the range from 0 to $2^{15}-1$.

Function `srand` can be called at any time to reset the random-number generator to a specific seed. The generator is initially seeded with a value of 1. Identical seeds produce identical sequences of pseudorandom numbers.

See also

`Random`, `randomx` in the *Apple Numerics Manual*

read—read from file

Synopsis

```
int read(fildes, buf, nbyte)
  int fildes;
  char *buf;
  unsigned nbyte;
```

Description

File descriptor `fildes` is obtained from a call to `open`, `creat`, `dup`, or `fcntl`.

Function `read` transfers up to `nbyte` bytes from the file associated with `fildes` into the buffer pointed to by `buf`.

On devices capable of seeking, `read` starts reading at the current position of the file pointer associated with `fildes`. Upon return from `read`, the file pointer is incremented by the number of bytes actually read.

Nonseeking devices always read from the current position. The value of a file pointer associated with such a file is undefined.

Upon successful completion, `read` returns the number of bytes actually read and placed in the buffer; this number may be less than `nbyte` if the number of bytes left in the file is less than `nbyte` bytes. A value of 0 is returned when the end-of-file has been reached, or –1 if a read error occurred.

Function `read` fails if `fildes` is not a valid file descriptor open for reading. [EBADF]

File descriptor 0 is opened by the APW Shell as the standard input.

Return value

Upon successful completion, a nonnegative integer is returned indicating the number of bytes actually read. Otherwise, –1 is returned and `errno` is set to indicate the error.

See also

`creat`, `open`

scanf—convert formatted input

Synopsis

```
#include <STDIO.H>

int scanf(format [ , pointer ] ... )
  char *format;
int fscanf(stream, format [ , pointer ] ... )
  FILE *stream;
  char *format;
int sscanf(str, format [ , pointer ] ... )
  char *str, *format;
```

Description

Function `scanf` reads characters from the standard input stream `stdin`. Function `fscanf` reads characters from the named input stream `stream`. Function `sscanf` reads characters from the character string `str`. Each function converts the input according to a control string (`format`) and stores the results according to a set of `pointer` parameters that indicate where the converted output should be stored.

Parameter `format`, the control string, contains specifications that control the interpretation of input sequences. The format consists of characters to be matched in the input stream and/or conversion specifications that start with the character `%`. The control string may contain

- White-space characters (spaces and tabs) that cause input to be read up to the next non-white-space character, except as described below.
- A character (any except `%`) that must match the next character of the input stream. To match a `%` character in the input stream, use `%%`.
- Conversion specifications beginning with the character `%` and followed by an optional assignment suppression character `*`, an optional numeric maximum field width, an optional `l`, `m`, `n`, or `h` indicating the size of the receiving parameter, and a conversion code.

An input field is defined relative to its conversion specification. The input field ends when the first character inappropriate for conversion is encountered or when the specified field width is exhausted. After conversion, the input pointer points to the inappropriate character.

A conversion specification directs the conversion of the next input field; the result is placed in the variable pointed to by the corresponding parameter, which is a pointer to a basic C type such as `int` or `float`.

Assignment can be suppressed by preceding a format character with the character `*`. *Assignment suppression* means an input field is skipped; the field is read and converted but not assigned. Therefore, `pointer` should be omitted when assignment of the corresponding input field is suppressed.

The **format character** dictates the interpretation of the input field. The following format characters are legal in a conversion specification, after `%`:

`%`	A single `%` is expected in the input at this point. No assignment is done.
`d`	A decimal integer is expected. The corresponding parameter should be an integer pointer.
`u`	An unsigned decimal integer is expected. The corresponding parameter should be an unsigned integer pointer.
`o`	An octal integer is expected. The corresponding parameter should be an integer pointer.
`x`	A hexadecimal integer is expected. The corresponding parameter should be an integer pointer. The conversion characters `d`, `u`, `o`, and `x` may be preceded by `l` or `h` to indicate that a pointer to `long` or `short`, rather than `int`, is in the parameter list. The `h` is ignored in this implementation because `int` and `short` are both 16 bits.
`e, f, g`	A floating-point number is expected. The next field is converted accordingly and stored through the corresponding parameter, which should be a pointer to a `float`, `double`, `comp`, or `extended`, depending on the size specification. The input format for floating-point numbers is an optionally signed string of digits, possibly containing a decimal point, followed by an optional exponent field consisting of `E` or `e` followed by an optionally signed integer. In addition, infinity is represented by the string "`INF`", and NaNs are represented by the string "`NAN`", optionally followed by parentheses that may contain a string of digits (the NaN code). Case is ignored in the infinity and NaN strings. The conversion characters `e`, `f`, and `g` may be preceded by `l`, `m`, or `n` to indicate that a pointer to `double`, `comp`, or `extended`, rather than `float`, is in the parameter list.
`s`	A character string is expected. The corresponding parameter should be a character pointer to an array of characters large enough to accept the string; a terminating null character (`\0`) is added automatically. The input field is terminated by a white-space character (space or tab), or when the number of characters specified by the maximum field width has been read.
`c`	A character is expected; the corresponding parameter should be a character pointer. The normal skip over white space is suppressed in this case; use `%1s` to read the next non-white-space character. If a field width is given, the corresponding parameter should refer to a character array; the indicated number of characters is read.

[The left bracket introduces a **scanset** format. The input field is the maximal sequence of input characters consisting entirely of characters in the scanset. When reading the input field, string data and the normal skip over leading white space are suppressed. The corresponding pointer parameter must point to a character array large enough to hold the input field and the terminating null character (\0), which will be added automatically. The left bracket is followed by a set of characters (the scanset) and a terminating right bracket.

^ When it appears as the first character in the scanset, the circumflex serves as a complement operator and redefines the scanset as the set of all characters not contained in the remainder of the scanset string.

] The right bracket ends the scanset. To be included as an element of the scanset, the right bracket must appear as the first character (possibly preceded by a circumflex) of the scanset. Otherwise, it will be interpreted syntactically as the closing bracket.

A range of characters may be represented by the construct *first-last*; thus the scanset `[0123456789]` may be expressed `[0-9]`. To use this convention, `first` must be less than or equal to `last` in the ASCII collating sequence. Otherwise, the minus (-) will stand for itself in the scanset. The minus will also stand for itself whenever it is the first or the last character in the scanset.

Conversion terminates at the end of file, at the end of the control string, or when an input character doesn't match the control string. In the last case, the unmatched character is left unread in the input stream.

Examples

Here are some ways the `scanf` function can be used:

Example 1

The call

```
int i;
float x;
char name[50];
scanf("%d%f%s", &i, &x, name);
```

with input

```
25 54.32E-1 reed
```

will assign the value 25 to `i` and the value 5.432 to `x`; name will contain "`reed\0`".

Example 2

The call

```
int i;
extended x;
char name[50];
scanf("%2d%nf%*d %[0-9]", &i, &x, name);
```

with input

```
56789 0123 56a72
```

will assign 56 to `i` and 789.0 to `x`, skip 0123, and place the string "`56\0`" in `name`. The next call to `getchar` will return "a".

Example 3

The call

```
int i;
scanf("answer1=%d", &i);
```

with input

```
answer1=51 answer2=45
```

will assign the value 51 to `i` because "`answer1`" is matched explicitly in the input stream. The input pointer will be left at the space before "`answer2`".

Return value

Functions `scanf`, `fscanf`, and `sscanf` return the number of successfully matched and assigned input items. This number can be zero when an early mismatch between an input character and the control string occurs. If the input ends before the first mismatch or conversion, `EOF` is returned.

These functions return `EOF` on end of input and a short count for missing or illegal data items.

Note

Trailing white space is left unread unless matched in the control string.

The success of literal matches and suppressed assignments is not directly determinable.

Warning

The `pointer` parameters in these functions must be addresses: for example, `&i`. Be sure not to pass `i` rather than its address.

See also

`atof`, `getc`, `printf`, `stdio`, `strtol`

`dec2num`, `str2dec` in the *Apple Numerics Manual*

setbuf—assign buffering to a stream

Synopsis

```
#include <STDIO.H>

void setbuf(stream, buf)
  FILE *stream;
  char *buf;
int setvbuf(stream, buf, type, size)
  FILE *stream;
  char *buf;
  int type;
  int size;
```

Description

A buffer is normally allocated by the Standard C Library at the time of the first `getc` or `putc` on a file. If you prefer to provide your own buffer, you can call `setbuf` or `setvbuf` after a stream has been associated with an open file but before it is read or written. Functions `setbuf` and `setvbuf` let you provide your own buffering for a file stream. Function `setvbuf` is a more flexible extension of `setbuf`.

Function `setbuf` causes the character array pointed to by `buf` to be used instead of an automatically allocated buffer. `BUFSIZ`, a constant defined in the <StdIO.h> header file, lets you specify the size of the `buf` array as

```
char buf[BUFSIZ];
```

If `buf` is `NULL`, input/output is unbuffered.

Function `setvbuf` lets you specify two parameters in addition to those required by `setbuf`: `size` and `type`. Parameter `size` specifies the size in bytes of the array to be used; the standard I/O functions work most efficiently when `size` is a multiple of `BUFSIZ`. If buffer pointer `buf` is `NULL`, a buffer of `size` bytes is allocated from the system. If `size` is not zero, `size` is assigned to the `FILE` variable's size parameter; if `buf` is not `NULL`, `buf` is assigned to the `FILE` variable's buffer-pointer parameter. The value of `type` determines how `stream` is buffered by `setvbuf`, as follows:

Value of type	Description
`_IOFBF`	Causes input/output to be file buffered.
`_IOLBF`	Causes output to be line buffered. The buffer is flushed when a newline character is written or when the buffer is full.
`_IONBF`	Causes input/output to be unbuffered. Parameters `buf` and `size` are ignored.

The following function calls are equivalent when `buf` is not `NULL`:

```
setbuf(stream, buf);
setvbuf(stream, buf, _IOFBF, BUFSIZ);
```

The following function calls are equivalent when `buf` is NULL:

```
setbuf(stream, NULL;
setvbuf(stream, NULL, _IONBF, BUFSIZ);
```

Diagnostics Function `setvbuf` returns nonzero if an invalid value is given for `type`.

Note The buffer must have a lifetime at least as great as the open stream. Be sure to close the stream before the buffer is deallocated. If you allocate buffer space as an automatic variable in a code block, be sure to close the stream in the same block.

If `buf` is NULL and the system cannot allocate `size` bytes, a smaller buffer will be allocated.

See also `fopen`, `getc`, `malloc`, `putc`, `stdio`

setjmp—nonlocal transfer of control

Synopsis

```
#include <SETJMP.H>

int setjmp(env)
  jmp_buf env;
void longjmp(env, val)
  jmp_buf env;
  int val;
```

Description

These functions let you escape from an error or interrupt encountered in a low-level subroutine of your program.

Function `setjmp` saves its stack environment in `env` for later use by `longjmp`. It returns the value 0.

Function `longjmp` restores the environment saved by the last call of `setjmp` with the corresponding `env` environment. After a call to `longjmp`, the program continues as if the preceding call to `setjmp` had returned the value `val`.

Function `longjmp` cannot cause `setjmp` to return the value 0. If `longjmp` is invoked with a second parameter of 0, `setjmp` returns 1. Data values will be those in effect at the time `longjmp` was called, except for register variables (see "Warning").

Warning

If `longjmp` is called without a previous call to `setjmp` or if the function that contained the `setjmp` has already returned, results are unpredictable.

See also

`signal`

sinh—hyperbolic functions

Synopsis

```
#include <MATH.H>

extended sinh(x)
  extended x;
extended cosh(x)
  extended x;
extended tanh(x)
  extended x;
```

Description

Functions `sinh`, `cosh`, and `tanh` return, respectively, the hyberbolic sine, cosine, and tangent of their parameter.

Diagnostics

Functions `sinh`, `cosh`, and `tanh` honor the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

See also

Apple Numerics Manual

stdio—standard buffered input/output package

Synopsis

```
#include <STDIO.H>
#include <STRING.H>

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

Description

The Standard I/O Package constitutes an efficient user-level I/O buffering scheme. The inline macros `getc` and `putc` handle characters quickly. Macros `getchar` and `putchar`, and the higher-level routines `fgetc`, `fgets`, `fprintf`, `fputc`, `fputs`, `fread`, `fscanf`, `fwrite`, `gets`, `getw`, `printf`, `puts`, `putw`, and `scanf` all use `getc` and `putc`. Calls to these macros and functions can be freely intermixed.

The constants and the following functions are implemented as macros: `getc`, `getchar`, `putc`, `putchar`, `feof`, `ferror`, `clearerr`, and `fileno`. Redeclaration of these names should be avoided.

Any program that uses the Standard I/O Package must include the `<StdIO.h>` header file of macro definitions. The functions, macros, and constants used in the Standard I/O package are declared in the header file and need no further declaration.

A *stream* is a file with associated buffering and is declared to be a pointer to a `FILE` variable. Functions `fopen`, `freopen`, and `fdopen` return this pointer. The information in the `FILE` variable includes

- the file access—read or write
- the file descriptor as returned by `open`, `creat`, `dup`, or `fcntl`
- the buffer size and location
- the buffer style (unbuffered, line-buffered, or file-buffered)

Standard I/O buffering

Output streams, with the exception of the standard error stream `stderr`, are by default file buffered if the output refers to a file. File `stderr` is by default line buffered. When an output stream is *unbuffered,* it is queued for writing on the destination file or window as soon as written; when it is *file buffered,* many characters are saved up and written as a block; when it is *line buffered,* each line of output is queued for writing as soon as the line is completed (that is, as soon as a newline character is written). Function `setvbuf` may be used to change the stream's buffering strategy.

Normally, there are three open streams with constant pointers declared in the `<STDIO.H>` header file and associated with the standard open files:

FILE variable	Fildes	Description	Buffer style

stdin	0	standard input file	line buffered
stdout	1	standard output file	file buffered
stderr	2	standard error file	line buffered

Buffer initialization

The FILE variable returned by fopen, freopen, or fdopen has an initial buffer size of 0 and a NULL buffer pointer. The buffer size is set and the buffer allocated by a call to setbuf, setvbuf, or the first I/O operation on the stream, whichever comes first. Buffer initialization is done using the following algorithm:

1. If _IONBF (no buffering) was set by a call to setvbuf, initialization steps 2 and 3 are skipped. The buffer size remains 0 and the buffer pointer remains NULL.
2. Checks the access-mode word for _IOLBF (line buffering). This bit is usually set only in the predefined file stderr, but a call to setvbuf can set it for any file. If line buffering is set, the buffer size is set to LBUFSIZ (100). If line buffering is not set, ioctl is called with an FIOBUFSIZE request and the buffer size is set to the returned value or to BUFSIZ (1024) if no value is returned.
3. If the buffer pointer is NULL, a request is made for a buffer whose size was determined in step 2; the buffer pointer is set to point to the newly allocated buffer. If the requested size cannot be allocated, attempts are made to allocate BUFSIZ or LBUFSIZ if these are smaller than the requested size. If all requests fail, the buffer pointer remains NULL and the _IONBF (no buffering) bit is set.
4. Function ioctl is called with an FIOINTERACTIVE request; if it returns true, the _IOSYNC bit is set in the access-mode word. This is done for all FILE variables, regardless of their buffering style and size. (The _IOSYNC bit is described in the following section.)

The setvbuf function lets you specify values for buffer size, buffer pointer, and access mode word other than the default values of 0, NULL, and 0, respectively. The setvbuf function must be called before the first I/O operation occurs, so that the buffer initialization procedure described above receives the values you specify instead of the default values.

Buffered I/O

On each write request, the bytes are transferred to the buffer and an internal counter is set to account for the number of bytes in the buffer. If _IOLBF is set and a newline character is encountered while transferring bytes to the buffer, the buffer is flushed (written immediately) and the transfer continues at the beginning of the buffer. This continues until the write-request count is satisfied or a write error occurs.

On each read request, the _IOSYNC bit in the access-mode word is checked. If _IOSYNC is on, all current FILE variables that have _IOSYNC on and are open for writing are flushed. In other words, a read from an interactive FILE variable flushes all interactive output files before reading. This ensures that any prompts, I/O in a window, or other visual feedback is displayed before the read is initiated. Then if the internal counter is 0, an entire buffer is read into memory if possible. (For the console device, less than a buffer's worth is likely to be read.) The bytes required to satisfy the read request are transferred, going back to the device for more if necessary, and an internal pointer is advanced if any bytes remain unread.

When the Standard I/O Package is used, Standard I/O cleanup is performed just before termination of the application. Any normal return including a call to exit causes Standard I/O cleanup, which consists of a call to fclose for every open FILE stream.

Note

Do not use a file descriptor (0, 1, or 2) where a FILE variable (stdin, stdout, or stderr) is required.

File <StdIO.h> includes definitions other than those described above, but their use is not recommended.

Invalid stream pointers cause serious errors, possibly including program termination. Individual function descriptions describe the possible error conditions.

Diagnostics

An integer constant EOF (–1) is returned upon end of file or error by most integer functions that deal with streams. See the descriptions of the individual functions for details.

See Also

open, close, lseek, read, write, fclose, ferror, fopen, fread, fseek, getc, gets, printf, putc, puts, scanf, setbuf, ungetc

string—string operations

Synopsis

```
#include <STRING.H>

char *strcat(destStr, srcStr)
  char *destStr, *srcStr;
char *strncat(destStr, srcStr, n)
  char *destStr, *srcStr;
  int n;
int strcmp(str1, str2)
  char *str1, *str2;
int strncmp(str1, str2, n)
  char *str1, *str2;
  int n;
char *strcpy(destStr, srcStr)
  char *destStr, *srcStr;
char *strncpy(destStr, srcStr, n)
  char *destStr, *srcStr;
  int n;
int strlen(str)
  char *str;
char *strchr(str, c)
  char *str, c;
char *strrchr(str, c)
  char *str, c;
char *strpbrk(srcStr, findChars)
  char *srcStr, *findChars;
int strspn(srcStr, spanChars)
  char *srcStr, *spanChars;
int strcspn(srcStr, skipChars)
  char *srcStr, *skipChars;
char *strtok(destStr, tokenStr)
  char *destStr, *tokenStr;
```

Description

The string parameters (`srcStr`, `destStr`, and so forth) and `s` point to arrays of characters terminated by a null character. The functions `strcat`, `strncat`, `strcpy`, and `strncpy` all alter `destStr`. These functions do not check for overflow of the array pointed to by `destStr`.

Function `strcat` appends a copy of string `srcStr` to the end of string `destStr`. Function `strncat` appends at most `n` characters. Each function returns a pointer to the null-terminated result.

Function `strcmp` performs a comparison of its parameters according to the ASCII collating sequence and returns an integer less than, equal to, or greater than 0 when `str1` is less than, equal to, or greater than `str2`, respectively. Function `strncmp` makes the same comparison but looks at a maximum of `n` characters.

Function `strcpy` copies string `srcStr` to string `destStr`, stopping after the null character has been copied. Function `strncpy` copies exactly `n` characters, truncating `srcStr` or adding null characters to `destStr` if necessary. The result is not terminated with a null if the length of `srcStr` is `n` or more. Each function returns `destStr`.

Function `strlen` returns the number of characters in `str`, not including the terminating null character.

Functions `strchr` and `strrchr` both return a pointer to the first and last occurrence, respectively, of character `c` in string `str`; they return `NULL` if `c` does not occur in the string. The null character terminating a string is considered to be part of the string. In previous versions of the Standard C Library, `strchr` was known as `index` and `strrchr` was known as `rindex`.

Function `strpbrk` returns a pointer to the first occurrence in string `srcStr` of any character from string `findChars`, or `NULL` if no character from `findChars` exists in `srcStr`.

Function `strspn` returns the length of the initial segment of string `srcStr` that consists entirely of characters from string `spanChars`.

Function `strcspn` returns the length of the initial segment of string `srcStr` that consists entirely of characters not from string `skipChars`.

Function `strtok` considers the string `destStr` as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string `tokenStr`. The first call (with pointer `destStr` specified) returns a pointer to the first character of the first token and writes a null character into `destStr` immediately following the returned token. The function keeps track of its position in the string between calls. Subsequent calls for the same string must be made with `NULL` as the first parameter. The separator string `tokenStr` may be different from call to call. When no token remains in `destStr`, `NULL` is returned.

Warning

Overlapping moves yield unexpected results.

Functions `strcmp` and `strncmp` use signed arithmetic when comparing their parameters. The sign of the result will be incorrect for characters with values greater than `0x7F` in the Apple IIGS extended character set.

See also

`BlockMove`, `EqualString`, `memory`

strtol—convert a string to a long

Synopsis

```
#include <STDLIB.H >

long strtol(str, ptr, base)
  char *str;
  char **ptr;
  int base;
```

Description

Function `strtol` returns a `long` containing the value represented by the character string `str`. The string is scanned up to the first character inconsistent with the base (decimal, hexadecimal, or octal). Leading white-space characters are ignored.

If the value of `ptr` is not `NULL`, a pointer to the character terminating the scan is returned in `*ptr`. If no integer can be formed, `*ptr` is set to `str` and 0 is returned.

If `base` is 0, the base is determined from the string. If the first character after an optional leading sign is not 0, decimal conversion is done; if the 0 is followed by x or X, hexadecimal conversion is done; otherwise octal conversion is done.

The function call `atol(str)` is equivalent to

```
strtol(str, (char **)NULL, 10)
```

The function call `atoi(str)` is equivalent to

```
(int) strtol(str, (char **)NULL, 10)
```

Note

Overflow conditions are ignored.

Apple base conventions (`$` for hexadecimal, `%` for binary) are not supported.

See Also

`atof`, `atoi`, `scanf`

trig—trigonometric functions

Synopsis

```
#include <MATH.H>

extended sin(x)
  extended x;
extended cos(x)
  extended x;
extended tan(x)
  extended x;
extended asin(x)
  extended x;
extended acos(x)
  extended x;
extended atan(x)
  extended x;
extended atan2(y, x)
  extended y, x;
```

Description

Functions `sin`, `cos`, and `tan` return, respectively, the sine, cosine, and tangent of their argument, which is in radians.

Function `asin` returns the arcsine of `x`, in the range $-\pi/2$ to $\pi/2$.

Function `acos` returns the arccosine of `x`, in the range 0 to π.

Function `atan` returns the arctangent of `x`, in the range $-\pi/2$ to $\pi/2$.

Function `atan2` returns the arctangent of `y/x`, in the range $-\pi$ to π, using the signs of both arguments to determine the quadrant of the return value.

For special cases, these functions return a NaN or infinity as appropriate.

Diagnostics

These functions honor the floating-point exception flags—invalid operation, underflow, overflow, divide by zero, and inexact—as prescribed by SANE.

Note

Functions `sin`, `cos`, and `tan` have periods based on the nearest extended-precision representation of mathematical π. Hence these functions diverge from their mathematical counterparts as their argument gets farther from zero.

See also

Apple Numerics Manual

ungetc—push a character back into the input stream

Synopsis

```
#include <STDIO.H>
int ungetc(c, stream)
  char c;
  FILE *stream;
```

Description

Function `ungetc` inserts the character `c` (which was returned by the last read call) into the buffer associated with an input stream. The stream must be file buffered or line buffered; it cannot be unbuffered. The inserted character, `c`, will be returned by the next `getc` call on that stream. Function `ungetc` returns `c` and leaves the file stream unchanged.

Only one character of pushback is allowed, provided something has been read from the stream and the stream is not unbuffered.

If `c` equals `EOF`, `ungetc` does nothing to the buffer and returns `EOF`.

Function `fseek` undoes the effect of `ungetc`.

Diagnostics

For `ungetc` to perform correctly, a read must have been performed before the call to the `ungetc` function. Function `ungetc` returns `EOF` if it can't insert the character.

Note

Function `ungetc` does not work on unbuffered streams.

See also

`fseek`, `getc`, `setbuf`, `stdio`

unlink—delete a named file

Synopsis

```
int unlink(fileName)
  char *fileName;
```

Description

Function `unlink` deletes the named file. The function fails if the named file is open.

A call to `unlink` is equivalent to

```
faccess(fileName, F_DELETE)
```

Diagnostics

Upon successful completion, a value of 0 is returned. Otherwise, a value of –1 is returned and `errno` is set to indicate the error.

See also

`faccess`

write—write on a file

Synopsis

```
int write(fildes, buf, nbyte)
  int fildes;
  char *buf;
  unsigned nbyte;
```

Description

File descriptor `fildes` is obtained from an `open`, `creat`, `dup`, or `fcntl` call.

Function `write` attempts to write `nbyte` bytes from the buffer pointed to by `buf` to the file associated with the `fildes`. Internal limitations may cause `write` to write fewer bytes than requested; the number of bytes actually written is indicated by the return value. Several calls to `write` may therefore be necessary to write out the contents of `buf`.

On devices capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file pointer. Upon return from `write`, the file pointer is incremented by the number of bytes actually written.

On nonseeking devices, writing starts at the current position. The value of a file pointer associated with such a device is undefined.

If the `O_APPEND` file status flag set in `open` is on, the file pointer is set to the end of file before each write.

The file pointer remains unchanged and `write` fails if `fildes` is not a valid file descriptor open for writing. [`EBADF`]

If you try to write more bytes than there is room for on the device, `write` writes as many bytes as possible. For example, if `nbyte` is 512 and there is room for 20 bytes more on the device, `write` writes 20 bytes and returns a value of 20. The next attempt to write a nonzero number of bytes will return an error. [`ENOSPC`]

File descriptor 1 is standard output; file descriptor 2 is standard error.

Return value

Upon successful completion, the number of bytes actually written is returned. Otherwise, –1 is returned and `errno` is set to indicate the error.

See also

`creat`, `lseek`, `open`

Chapter 6
Shell Calls

The Apple IIGS Progammer's Workshop Shell acts as an interface and extension to ProDOS 16. The shell provides several functions not provided by ProDOS 16; these functions are called exactly like ProDOS 16 functions. Every time a program running under the APW Shell issues a ProDOS 16-like call, the shell intercepts the call. If the call is a shell call, the shell interprets it and acts on it; if it is a ProDOS 16 call, the shell passes it on to ProDOS 16. This chapter describes all of the shell's ProDOS 16-like calls, here referred to as **shell calls**.

The shell calls that are provided are listed in Table 6-1.

Table 6-1. Shell Calls

Call Name	Call Use Number	
GET_LINE_INFO	($0101)	Passes parameters from the shell to a program
SET_LINE_INFO	($0102)	Passes parameters from a program to the shell
GET_LANG	($0103)	Reads the current language number
SET_LANG	($0104)	Sets the current language number
ERROR	($0105)	Prints error message for a Apple IIGS tool call
SET_VAR	($0106)	Sets the value of a shell variable
VERSION	($0107)	Returns the version number of the APW Shell
READ_INDEXED	($0108)	Reads variable table
INIT_WILDCARD	($0109)	Provides a filename that includes a wildcard character to the shell
NEXT_WILDCARD	($010A)	Causes the shell to find the next filename that matches the wildcard filename
GET_VAR	($010B)	Reads the value of a shell variable
EXECUTE	($010D)	Sends a command or list of commands to the shell command interpreter
DIRECTION	($010F)	Tells whether I/O redirection has occurred
REDIRECT	($0110)	Sets device and file for I/O redirection
STOP	($0113)	Detects a request for an early termination of the program

How to make a shell call

To make a shell call, do the following:

- Include the statements

```
#include <TYPES.H>
#include <SHELL.H>
```

  in your source text. Your object file will be automatically linked with the library file CLIB.

- Set values in the shell data structures and call the shell routines from your program, following the information given below.

How a program makes a shell call

A C program makes a shell call by calling a function in the file ORCA.C. Most of these calls are simple C function calls: parameters are passed in the normal way.

Two of these, GET_LINE_INFO and SET_LINE_INFO, are called differently. Values and results are passed via a parameter block. To get information from the shell, your program declares and initializes this parameter block, then calls GET_LINE_INFO, then reads results from the block. To send information to the shell, your program writes values into the block, then calls SET_LINE_INFO to send the information. These calls are explained in detail in the section "GET_LINE_INFO and SET_LINE_INFO," below.

With the exception of EXECUTE, all calls expect Pascal-style strings.

Call descriptions

This section lists each of the shell calls, describes its use, and describes the contents of its parameter block. The possible errors returned by a call are listed at the end of each call description. The calls are listed in alphabetical order. Table 6-1 lists all of the calls in order of their call numbers.

DIRECTION (0x010F)

```
DIRECTION(device, direct)
      int device, *direct;
```

A program can use this function to find out whether command-line I/O redirection has occurred. This function can be used by a program to determine whether to send form feeds to standard output, for example.

The device parameter indicates which type of input or output you are inquiring about, as follows:

0x0000	Standard input
0x0001	Standard output
0x0002	Error output

The `direct` parameter indicates the type of redirection that has occurred, as follows:

0x0000	Console
0x0001	Printer
0x0002	Disk file

Possible Errors

0x53	Parameter out of range

ERROR (0x0105)

```
int error()
```

When a Apple IIGS tool call returns an error, your program can use this function to print out the name of the tool and the appropriate error message. This function makes it unnecessary for your program to store a complete table of error messages for tool calls. The error number is returned in `_toolErr`.

Possible Errors

None

EXECUTE (0x010D)

```
EXECUTE(flag, comm)
      int flag;
      char *comm;
```

This function sends a command or list of commands to the APW shell.

The `flag` parameter is used to execute an Exec file with an `EXECUTE` command; if no new variable table is defined, then variables defined by the list of commands modify the current variable table. If you set the most significant bit of this flag to 1 (binary), then a new variable table is not defined when the commands are executed. If this flag is set to 0x0000, a new variable table is defined for the list of commands being executed; the current variable table is not modified. Exec files, variables, and the `EXECUTE` command are described in the section "Exec Files" in Chapter 4 of the *APW Reference*.

The `comm` parameter is the address of the buffer in which you place the commands. If you include more than one command, separate the commands with semicolons (;) or carriage return characters (0x0D). The command string is a C string: it has no length byte and is terminated with a null character (0x00). Any output is sent to standard output.

If the shell variable `{Exit}` is not null and any command returns a non-zero error code, then any remaining commands are ignored. Error codes and shell variables are described in the section "Exec Files" in Chapter 3 of the *APW Reference*.

Possible Errors

Any error returned from the last command or program executed by the list of commands executed.

GET_LANG (0x0103)

```
int GET_LANG()
```

This function reads the current language number. The current language number is set by the APW Editor when it opens an existing file, or by the user with an APW shell command. Language numbers are described in the section "Command Types and the Command Table" in Chapter 4 of the *APW Reference*, and are listed in Appendix A of the *APW Reference*.

Possible Errors

None

GET_LINE_INFO (0x0101) and SET_LINE_INFO (0x0102)

```
GET_LINE_INFO(pb)
      GetLinfoPB *pb;

SET_LINE_INFO(pb)
      GetLinfoPB *pb;
```

The GET_LINE_INFO function is used by an assembler, compiler, linker, or editor to read the parameters that are passed to it. When you make this call, you declare the parameter block GetLinfoPB; when the APW Shell returns control to your program, you can then read the parameter block to obtain the information you need.

Use the GET_LINE_INFO call to read parameters passed to your assembler, compiler, linker, or editor.

The SET_LINE_INFO function is used by an assembler, compiler, linker, or editor to pass parameters to the APW Shell before returning control to the shell. It can also be used by a shell program under which you are running APW to pass parameters to the APW Shell.

Use the SET_LINE_INFO call when your program is finished before returning control to the shell.

Both of these calls use the following parameter block:

```
GetLInfoPB                  /* get/set Line Info parameter block */
typedef struct {
/* unsigned char id;        */
   char *sfile;             /* address of source file name */
   char *dfile;             /* address of output file name */
   char *parms;             /* address of parameter list */
   char *istring;           /* address of language-specific input string */
```

```
    char merr;              /* maximum error level allowed */
    char merrf;             /* maximum error level found */
    char lops;              /* operations flag */
    char kflag;             /* KEEP flag */
    unsigned long mflags; /* set of letters selected with '-' */
    unsigned long pflags; /* set of letters selected with '+' */
} GetLInfoPB;
```

To call GET_LINE_INFO, first declare the parameter block GetLInfoPB. The GET_LINE_INFO call passes to the shell the pointer, pb, to your parameter block. The shell then writes its results into your parameter block: you can read them from there.

To call SET_LINE_INFO, first declare the parameter block GetLInfoPB, then write your values into that block. The GET_LINE_INFO call passes to the shell the pointer, pb, to your parameter block. The shell then reads your values from the parameter block.

The sfile (source file) field is the address of a buffer containing the filename of the source file; that is, the next file that a compiler or assembler is to process. The filename can be any valid ProDOS 16 filename, and can be a partial or full pathname.

The dfile (destination file) field is the address of a buffer containing the filename of the output file (if any); that is, the file that the compiler or assembler writes to. The filename can be any valid ProDOS 16 filename, and can be a partial or full pathname.

The parms field is the address of a buffer containing the list of names from the NAMES= parameter list in the APW Shell command that called the assembler or compiler. The compiler can remove or modify these names as it processes them, so this list can be different from the one received through the GetLInfo call..

The istring field is a placeholder for the address of a buffer containing the string of commands passed to the compiler. This command string is not reused by the shell, so it is not necessary to pass it back to the shell with the SET_LINE_INFO call.

The merr field is the maximum error level allowed. If the maximum error level found by the assembler, compiler, or linker is greater than merr, then the shell does not call the next program in the processing sequence. For example, if you use the ASML command to assemble and link a program, but the assembler finds an error level of 8 when merr equals 2, then the linker is not called when the assembly is complete.

The merrf field is the maximum error level found. If merrf is greater than merr, then no further processing is done by the shell. If the high bit of merrf is set, then merrf is considered to be negative; a negative value of merrf indicates a fatal error (normally, all fatal errors are flagged as merrf=0xFF). In this case, processing terminates immediately and control is passed by the shell to the APW Editor. See also the discussion of the org field.

The lops field comprises the operation flags. This field is used to keep track of the operations that have been performed, and remain to be performed, by the system,. The format of this byte is as follows:

Bit:	7	6	5	4	3	2	1	0
Value:	0	0	0	0	0	E	L	C

where C = Compile
L = Link
E = Execute

When a bit is set (1), the indicated operation is to be done. When a compiler finishes its operation and returns control to the shell, it clears bit 0 unless a file with another language is appended to the source. When a linker returns control to the shell, it clears bit 1. When you execute the APW Linker by compiling a LinkEd file, the linker clears bits 0 and 1.

The `kflag` field is the keep flag. This flag indicates what should be done with the output of a compiler, assembler, or linker, as follows:

Kflag Value	**Meaning**
0x00	Do not save output.
0x01	Save to an object file with the root filename pointed to by `dfile`. For example, if the output filename pointed to by `dfile` is `PROG`, then the first segment to be executed should be put in `PROG` or `PROG.ROOT` and the remaining segments should be put in `PROG.A`. For linkers, save to a load file with the name pointed to by `dfile` (for example, `PROG`). A compiler or assembler will never set `kflag` to 0x01, but a shell program calling APW might use this value.
0x02	The `.ROOT` file has already been created. In this case, the first file created by the next compiler or assembler should end in the `.A` extension.
0x03	At least one alphabetic suffix has been used. In this case, the compiler or assembler must search the directory for the highest alphabetic suffix that has been used, and then use the next one. For example, if `PROG.ROOT`, `PROG.A`, and `PROG.B` already exist, the compiler should put its output in `PROG.C`.

When the compiler or assembler passes control back to the shell, it should reset `kflag` to indicate which object files it has written; for example, if it found only one segment and created a `.ROOT` file but no `.A` file, then `kflag` should be 0x02 in the `SET_LINE_INFO` call. See the section "Compilers and Assemblers" in Chapter 8 of the *APW Reference* for more information on object-file naming conventions.

The `mflags` (minus flags) field passes the flags with a minus sign. This field passes command-line-option flags such as `-L` or `-C`. The first 26 bits of these four bytes represent the letters A–Z, arranged with A as the most significant bit of the most significant byte; the bytes are ordered least significant byte first. The bit map is as follows:

```
11000000 11111111 11111111 11111111
YZ       QRSTUVWX IJKLMNOP ABCDEFGH
```

For each flag set with a minus sign in the command, the corresponding bit in this field is set to 1. See the discussions of the `ALINK` and `ASML` commands in Chapter 4 of the *APW Reference* for descriptions of these option flags.

The pflags (plus flags) field passes the flags with a plus sign. This field passes command-line-option flags such as +L or +C. The first 26 bits of these four bytes represent the letters A–Z; the bit map for this field is the same as for the mflags field. See the discussions of the ALINK and ASML commands in Chapter 4 of the *APW Reference* for descriptions of these option flags.

Possible Errors

None

INIT_WILDCARD (0x0109)

```
INIT_WILDCARD(file, flags)
      char *file;
      int flags
```

This function provides to the APW Shell a filename that can include a wildcard character. The shell can then search for filenames matching the filename you specified when it receives a NEXT_WILDCARD command. This function accepts any filename, whether it includes a wildcard or not, and expands device names (such as .D1/), prefix numbers, and the double-period (..) before the filename is passed on to ProDOS 16. Therefore, you should call this function every time you want to search for a filename. Doing so will assure that your routine supports all of the conventions for partial pathnames that the user expects from APW.

The file parameter is the address of a buffer containing a pathname or partial pathname that can include a wildcard character. Examples of such pathnames are:

```
A=
/APW/MYPROGS/?.ROOT
.D2/HELLO
```

Important: The file parameter must be all uppercase, or the file will not be found.

When you execute a NEXT_WILDCARD call, the shell finds the next filename that matches the filename pointed to by file. If the wildcard character you specified was a question mark (?), then the filename is written to standard output and you are prompted for confirmation before the file is acted on or the next filename is found. The use of wildcard characters is described in the section "Using Wildcard Characters" in Chapter 2 of the *APW Reference*.

The flags parameter contains the prompting flags. If the most significant bit is set, prompting is not allowed; that is a question mark (?) is treated as if it were an equal sign (=). If the next-most significant bit is set and prompting is being used, only the first choice accepted by the user (that is, the first choice for which the user types a Y in response to the prompt) is acted on. The second flag is for use with commands that can act on only one file, such as RENAME or EDIT.

Possible Errors

Errors for the following ProDOS 16 and Memory Manager calls. Use the ERROR function to get the error message. See the *Apple IIGS ProDOS 16 Reference* manual and the *Apple IIGS Toolbox Reference* manual for descriptions of these errors.

NEXT_WILDCARD (0x010A)

```
char *NEXT_WILDCARD(nextfile)
      char *nextfile;
```

Once a filename that includes a wildcard has been suppled to the shell with an `INIT_WILDCARD` call, the `NEXT_WILDCARD` call causes the shell to find the next filename that matches the wildcard filename. For example, if the wildcard filename specified in `INIT_WILDCARD` were `/APW/UTILITIES/XREF.?`, then the first filename returned by the shell in response to a `NEXT_WILDCARD` call might be `/APW/UTILITIES/XREF.ASM65816`.

The `nextfile` parameter is the address of the buffer to which the shell has returned the next filename that matches a wildcard filename. The wildcard filename is the last one specified with an `INIT_WILDCARD` call. If there are no more matching filenames, or if `INIT_WILDCARD` has not been called, then the shell returns a null string (that is, a string with length zero). See also the description of `INIT_WILDCARD`.

Possible Errors

None

GET_VAR (0x010B)

```
GET_VAR(varname, value)
      char *varname, *value;
```

This function reads the string associated with a variable (that is, the value of the variable). The value returned is the one valid for the currently-executing Exec file, or for the interactive command interpreter. Variables and Exec files are described in the section "Exec Files" in Chapter 4 of the *APW Reference*. Use the `SET_VAR` call to set the value of a variable.

The `varname` parameter is a pointer to a buffer that contains the name of the variable whose value you wish to read. The variable name consists of a length byte and a string of up to 255 ASCII characters.

The `value` parameter is a pointer to a 256-byte buffer into which the shell places the value of the variable. The value consists of a length byte and a string of ASCII characters. The value consists of a null string (that is, the length byte is 0x00) for an undefined variable.

Possible Errors

None

READ_INDEXED (0x0108)

```
READ_INDEXED(varname, value, index)
      char *varname, *value;
      int index;
```

You can use this function to read the contents of the variable table for the command level at which the call is made. To read the entire contents of the variable table, you must repeat this call, incrementing the index number by 1 each time, until the entire contents have been returned.

The `varname` parameter is a pointer to a 256-byte buffer in which the shell places the name of the next variable in the variable table. The variable name consists of a length byte and a string of ASCII characters. A null string is returned when the index number exceeds the number of variables in the variable table.

The `value` parameter is a pointer to a 256-byte buffer into which the shell places the value of the variable. The value consists of a length byte and a string of ASCII characters. The value consists of a null string (that is, the length byte is 0x00) for an undefined variable.

The `index` parameter is an index number that you provide. Start with 0x01 and increment the number by 1 with each successive `READ_INDEXED` call until there are no more values in the variable table.

Possible Errors

Errors for the following Memory Manager calls. See the *Apple IIGS Toolbox Reference* manual for descriptions of these errors.

LOCK
UNLOCK

REDIRECT (0x0110)

```
REDIRECT(device, app, file)
      int device, app;
      char *file;
```

This function instructs the shell to redirect input or output to the printer, console, or a disk file.

The `device` parameter indicates which type of input or output you wish to redirect, as follows:

0x0000	Standard input
0x0001	Standard output
0x0002	Error output

The `app` flag indicates whether redirected output should be appended to an existing file with the same filename, or the existing file should be deleted first. If `append` is 0, the file is deleted, if it is any other value, the output is appended to the file.

The `file` parameter is the address of a 65-byte-long buffer containing the filename of the file to or from which output is to be redirected. The filename can be any valid ProDOS 16 filename, a partial or full pathname, or the device names .PRINTER or .CONSOLE.

Possible Errors

0x53 Parameter out of range

Errors for the following ProDOS 16 calls. See the *Apple IIGS ProDOS 16 Reference* manual and the *Apple IIGS Toolbox Reference* manual for descriptions of these errors.

OPEN
CLOSE
GET_VAR
WRITE
GET_EOF

SET_VAR (0x0106)

```
SET_VAR(varname, value)
     char *varname, *value;
```

This function sets the value of a variable. If the variable has not been previously defined, this function defines it. Variables are described in the section "Exec Files" in Chapter 4 of the *APW Reference*. Use the GET_VAR call to read the current value of a variable and the READ_INDEXED call to read a variable table.

The `varname` parameter is a pointer to a buffer in which you place the name of the variable whose value you wish to change. The name is an ASCII string.

The `value` parameter is a pointer to a buffer in which you place the value to which the variable is to be set. The value is an ASCII string.

Possible Errors

Errors for the following Memory Manager calls. See the *Apple IIGS Toolbox Reference* manual for descriptions of these errors.

Lock
Unlock
Grow
New

SET_LANG (0x0104)

```
SET_LANG(lang)
     int lang;
```

This function sets the current language number. Language numbers are described in the section "Command Types and the Command Table" in Chapter 4 of the *APW Reference*, and are listed in Appendix A of the *APW Reference*.

The `lang` parameter is the APW language number to which the current APW language should be set. If the language specified is not installed (that is, not listed in the command table), then the "language not available" error is returned..

Possible Errors

0x80	Language not available

STOP (0x0113)

```
int STOP();
```

This function lets your application detect a request for an early termination of the program. The `STOP` flag is set when the keyboard buffer is read after the user presses ⌘-. (Apple-period).

The `STOP` flag is set (0x0001) by the shell when it finds an ⌘-. in the keyboard buffer. When a APW utility reads from the keyboard as standard input, the shell reads the keyboard buffer and passes the keys on to the utility. When standard input is not from the keyboard, the shell still checks the keyboard buffer for ⌘-. whenever a `STOP` call is executed. The flag is cleared (0x0000) when the `STOP` call is executed, when the utility program is terminated, or when windows are switched so that the utility program is no longer active.

Possible Errors

None

VERSION (0x0107)

```
int VERSION();
```

This function returns the version of the APW Shell that you are using

The `VERSION` parameter is a four-byte ASCII string specifying the version number of the APW Shell that you are using. The initial release returns 10 followed by two space characters (0x3130 0x2020), to indicate version number 1.0.

Possible Errors

None

Appendix A

Calling Conventions

APW C uses two different function-calling conventions: C calling conventions and Pascal-compatible calling conventions.

C calling conventions

This section describes the normal C calling conventions. It explains how function parameters are passed, how function results are returned, and how registers are saved across function calls. This information is useful when writing calls between C and assembly language.

Parameters

Parameters to C functions are evaluated from right to left and are pushed onto the stack in the order they are evaluated: that is, they are pushed in reverse order. Characters, integers, and enumeration types are passed as sign-extended 16-bit values. Pointers and arrays are passed as 32-bit addresses. Types `float`, `double`, `comp`, and `extended` are passed as extended 80-bit values. Structures are also passed by value on the stack. Their size is rounded up to a multiple of 16 bits (2 bytes). If rounding occurs, the unused storage has the highest memory address. The caller removes the parameters from the stack.

Function results

On the Apple IIGS, function results are returned in registers: the low 16 bits are in the A register, and the high 16 bits are in the X register. Results of types `float`, `double`, `comp`, and `extended` are passed as type `extended`, with the address in the A and X registers, as before. Structure results are returned in a static location, the address returned in the A and X registers.

Register conventions

No registers are preserved across function calls. Tool calls have their own conventions for returning error codes in the A register.

Pascal-style calling conventions

This section describes the conventions used for calling functions that use Pascal-style calling conventions: these functions are declared with the keyword `pascal` and may

have been written in any language. These conventions differ from the usual C calling conventions defined in Chapter 4.

Parameters

Parameters to Pascal-compatible functions are evaluated left to right: that is, in the order of the formal parameter list. The function first pushes space for the result (as shown in Table 3-2), then pushes the parameters onto the stack in the order in which they are evaluated. Characters and enumeration types whose literal values fall in the range of types `char` or `unsigned char` are pushed as bytes. (This requires a 16-bit word on the stack. The value is in the high-order 8 bits; the low-order 8 bits are unused.) Short `int`s and enumeration types whose literal values fall in the range of types `short` or `unsigned short` are passed as 16-bit values. `Int`s, `long int`s, and the remaining enumeration types are passed as 32-bit values. Pointers and `array`s are passed as 32-bit addresses. SANE types `float`, `double`, `comp`, and `extended` are passed as extended 80-bit values. Since this doesn't correspond to the Pascal compiler's calling conventions, however, a compiler warning is given. Table 3-2 shows the recommended way to pass SANE-type values to Pascal. Structures are also passed by value on the stack, and they also yield a compiler warning. Their size is rounded up to a multiple of 16 bits (2 bytes). If rounding occurs, the unused storage has the highest memory address. The function being called removes the parameters from the stack.

Function results

On the Apple IIGS, as on the Macintosh, results of Pascal-compatible functions are returned on the stack.

Register conventions

No registers are preserved across function calls. Tool calls have their own conventions for returning error codes in the A register.

Appendix B

Files Supplied with APW C

APW C is intended for use with the Apple Programmer's Workshop. The files listed below are on the APW C release disk, which contains the C compiler, the Standard C Library, and the Apple IIGS Interface Library. These files may be used directly from the release disk or copied to a hard disk.

*** These lists are subject to change as files are added or deleted. ***

The files are listed indented under their respective directories, with comments.

/APWC/	
LANGUAGES/	
CC	APW C compiler
LIBRARIES/	
CINCLUDE/	Standard C Library and Apple IIGS Toolbox include files
CTYPE.H	
ERRNO.H	
ERRORS.H	
FCNTL.H	
FILES.H	
IOCTL.H	
MATH.H	
MEMORY.H	Memory Manager
PRODOS.H	ProDOS interface
SANE.H	SANE interface
SIGNAL.H	
STDIO.H	Standard I/O Package
STRING.H	String conversion routines
TEXTTOOL.H	Text Tools
TYPES.H	common defines and types
VALUES.H	
VARARGS.H	
SETJMP.H	
CONTROL.H	Control Manager
DESK.H	Desk Accessory Manager
DIALOG.H	Dialog Manager
EVENT.H	Event Manager
FONT.H	Font Managen
INTMATH.H	Fixed-Point Math
LINEEDIT.H	Line Editor
LIST.H	List Manager
LOADER.H	System Loader
MENU.H	Menu Manager

MISCTOOL.H	Miscellaneous Tools
NOTESYN.H	
PRINT.H	
QDAUX.H	
QUICKDRAW.H	QuickDraw II
SCHEDULER.H	Scheduler
SCRAP.H	Scrap Handler
SHELL.H	APW shell interface
SOUND.H	Sound Driver
TEXTTOOL.H	Text Tools
WINDOW.H	Window Manager
ERRORS	Errors file
CLIB	Standard C Library
SYS.INTERFACE	
START.ROOT	

Appendix C

Comparison with Macintosh Programmer's Workshop C

Apple IIGS Programmer's Workshop C is as closely related to Macintosh Programmer's Workshop C as differences between the two machines allow. The differences between the two languages are explained here.

Data types

The following data types are implemented differently in APW and MPW C.

Data Type	Size in bits	
	APW	**MPW**
`int`	16	32
`unsigned int`	16	32
`enum`	8 or 16	8, 16 or 32

Register variables

Register variables are not supported in APW C due to the small number of registers available on the 65816. Use of the `register` declaration will cause the compiler to generate code at least as efficient as that generated by the same program without `register` declarations.

Structured variables

Structures may be assigned, passed as parameters, and returned as function results in both versions of C. Byte-sized elements in structures are not padded to word or long-word boundaries. APW C allows equality comparison for structures; MPW C does not.

Pascal-compatible function declarations

A function or procedure written in Pascal (or written in assembly language following Pascal calling conventions) can be called from either MPW C or APW C. For example, the `DrawText` procedure is defined in Pascal as

```
PROCEDURE DrawText (textBuf: Ptr;
      firstByte, byteCount: INTEGER);
```

The MPW C syntax for such a declaration is

```
pascal void DrawText(textBuf, firstByte, byteCount)
      Ptr textBuf;
      short firstByte, byteCount;
      extern;
```

The APW C syntax for this declaration is

```
extern pascal void DrawText();
```

To make the APW C form more readable, we can list the parameters in a comment:

```
extern pascal void DrawText();
      /* Ptr textBuf;
      short firstByte, byteCount;
      extern; */
```

In addition, in MPW C the word `extern` may be followed by a constant, which is interpreted as a 16-bit 68000 instruction that replaces the usual subroutine call (`JSR`) instruction in the calling sequence. This allows direct traps to the Macintosh ROM. For example:

```
pascal void OpenPort(port)
      GrafPtr port;
      extern 0xA86F;
```

On the Apple IIGS, an `inline` declaration is used for declaring tool routines. Its syntax is

`[extern] pascal [`*result-type*`]` *func-name* `() inline(`*m*`,`*n*`);`

This says that the tool routine with trap number *n* and entry point address *m* can be called by the function name *func-name*, and returns a result of type *result-type*.

Inline assembly-code declarations

An APW C program can contain assembly code inline. Anywhere a statement is legal, you can insert a series of assembly-language statements with this format:

`asm{`*assembly-statements*`}`

Anywhere a function definition is legal, you can have a definition with this format

`asm(`*external-name*`) {`*assembly-statements*`}`

This function can be called in the same way as a C function called *external-name*. Here *external-name* is the entry point of the segment containing the assembly-language code.

Appendix D

Library Index

The Library Index contains an index entry for all the defines, types, enumeration literals, global variables, and functions defined in the Standard C Library.

- Column 1 contains an alphabetical list of the index entries.
- Column 2 specifies the type of declaration (for example, "literal") for the index entry.
- Column 3 contains the library header under which documentation for the index entry can be found. If column 3 contains "(C)" following the library header—for example, "abs(C)"—look in Chapter 5, The Standard C Library. If column 3 contains "(S)" following the library header—for example, "bbb(S)"—look in Chapter 6, The Shell Interface. These chapters are organized alphabetically by library header except for the first entry in each, which contains introductory material.

Identifier	Type	Manual page
abs	function	abs
acos	function	trig
asin	function	trig
atan	function	trig
atan2	function	trig
atof	function	atof
atoi	function	atoi
atol	function	atoi
BUFSIZ	define	setbuf
calloc	function	malloc
ceil	function	floor
cfree	function	malloc
clearerr	macro	ferror
close	function	close
cos	function	trig
cosh	function	sinh
creat	function	creat
dup	function	dup
EACCES	define	Error
EBADF	define	Error
ecvt	function	ecvt
EEXIST	define	Error
EINVAL	define	Error
EIO	define	Error
EISDIR	define	Error
EMFILE	define	Error
ENFILE	define	Error
ENODEV	define	Error
ENOENT	define	Error
ENOMEM	define	Error
ENOSPC	define	Error
ENOTDIR	define	Error
ENXIO	define	Error
EOF	define	stdio
EPERM	define	Error
EROFS	define	Error
ESPIPE	define	Error
_exit	function	exit
exit	function	exit
exp	function	exp
fabs	function	floor
faccess	function	faccess
fclose	function	fclose
fcntl	function	fcntl
fcvt	function	ecvt
F_DELETE	define	faccess
fdopen	function	fopen
F_DUPFD	define	fcntl
feof	macro	ferror
ferror	macro	ferror
fflush	function	fclose
fgetc	function	getc
F_GETFD	define	fcntl
F_GETFL	define	fcntl
fgets	function	gets
F_GFONTINFO	define	faccess
F_GPRINTREC	define	faccess
F_GTABINFO	define	faccess
FILE	define	stdio
fileno	macro	ferror
FIOBUFSIZE	define	ioctl
FIODUPFD	define	ioctl
FIOFNAME	define	ioctl
FIOINTERACTIVE	define	ioctl
FIOLSEEK	define	ioctl
FIOREFNUM	define	ioctl
FIOSETEOF	define	ioctl
floor	function	floor
fmod	function	floor
F_OPEN	define	faccess
fopen	function	fopen
fprintf	function	printf
fputc	function	putc
fputs	function	puts
fread	function	fread
free	function	malloc
F_RENAME	define	faccess
freopen	function	fopen
frexp	function	frexp
fscanf	function	scanf
fseek	function	fseek
F_SETFD	define	fcntl
F_SETFL	define	fcntl
F_SFONTINFO	define	faccess
F_SPRINTREC	define	faccess
F_STABINFO	define	faccess
ftell	function	fseek
fwrite	function	fread
getc	macro	getc
getchar	macro	getc
getenv	function	getenv
gets	function	gets
getw	function	getc
hypot	function	hypot
ioctl	function	ioctl
_IOFBF	define	setbuf
_IOLBF	define	setbuf
_IONBF	define	setbuf
_IOSYNC	define	stdio
isalnum	macro	ctype
isalpha	macro	ctype
isascii	macro	ctype
iscntrl	macro	ctype
isdigit	macro	ctype
isgraph	macro	ctype
islower	macro	ctype
isprint	macro	ctype
ispunct	macro	ctype
isspace	macro	ctype
isupper	macro	ctype
isxdigit	macro	ctype
ldexp	function	frexp
log	function	exp
log10	function	exp
longjmp	function	setjmp
lseek	function	lseek
malloc	function	malloc
memccpy	function	memory
memchr	function	memory
memcmp	function	memory
memcpy	function	memory
memset	function	memory
modf	function	frexp
NULL	define	stdio
O_APPEND	define	open
O_CREAT	define	open
O_EXCL	define	open
onexit	function	onexit

Identifier	Type	Manual page
open	function	open
O_RDONLY	define	open
O_RDWR	define	open
O_RSRC	define	open
O_TRUNC	define	open
O_WRONLY	define	open
pow	function	exp
printf	function	printf
putc	macro	putc
putchar	macro	putc
puts	function	puts
putw	function	putc
qsort	function	qsort
rand	function	rand
read	function	read
realloc	function	malloc
rewind	function	fseek
scanf	function	scanf
setbuf	function	setbuf
setjmp	function	setjmp
setvbuf	function	setbuf
short	type	signal
SIGALLSIGS	define	signal
SIG_DFL	define	signal
_sig_dfl	function	signal
sighold	function	signal
SIG_IGN	define	signal
SIGINT	define	signal
SignalHandler	type	signal
SignalMap	type	signal
sigpause	function	signal
sigrelease	function	signal
sigset	function	signal
sin	function	trig
sinh	function	sinh
sprintf	function	printf
sqrt	function	exp
srand	function	rand
sscanf	function	scanf
strcat	function	string
strchr	function	string
strcmp	function	string
strcpy	function	string
strcspn	function	string
strlen	function	string
strncat	function	string
strncmp	function	string
strncpy	function	string
strpbrk	function	string
strrchr	function	string
strspn	function	string
strtok	function	string
strtol	function	strtol
tan	function	trig
tanh	function	sinh
TIOFLUSH	define	ioctl
TIOGPORT	define	ioctl
TIOSPORT	define	ioctl
toascii	macro	conv
tolower	function	conv
_tolower	macro	conv
toupper	function	conv
_toupper	macro	conv
ungetc	function	ungetc
unlink	function	unlink
write	function	write

Identifier	Type	Manual page	Identifier	Type	Manual page

Glossary

***:** A 32-bit pointer data type.

absolute code: Program code that must be loaded at a specific address in memory and never moved.

absolute segment: A segment that can be loaded only at one specific location in memory. Compare with **relocatable segment.**

accumulator: The register in the 65C816 microprocessor of the Apple IIGS used for most computations.

address: A number that specifies the location of a single **byte** of memory. Addresses can be given as decimal or hexadecimal integers. The Apple IIGS has addresses ranging from 0 to 16,777,215 (in decimal) or from $00 00 00 to $FF FF FF (in hexadecimal). A complete address consists of a 4-bit **bank** number ($00 to $FF) followed by a 16-bit address within that bank ($00 00 to $FF FF).

Apple key: A modifier key on the Apple IIGS keyboard, marked with an Apple icon. It performs the same functions as the*Open Apple* key on standard Apple II machines.

Apple II: A family of computers, including the original Apple II, the Apple II Plus, the Apple IIe, the Apple IIc, and the Apple IIGS.

`AppleIIgs`: A predefined constant identifying C code written for the AppleIIGS, in particular, for APW C.

Apple IIGS Interface Libraries: A set of **interfaces** that enable you to access toolbox routines from C.

`APW`: A predefined constant identifying C code written for the APW C compiler as opposed to another C compiler.

APW Shell: The programming environment of the Apple IIGS Programmer's Workshop. It lets you edit programs, manipulate files, and execute programs.

APW Linker: The linker supplied with APW.

application: A program (such as the APW Shell itself) that talks to ProDOS and the Toolbox directly, and can be exited via the `Quit` call.

assembler: A program that produces object files from source files written in assembly language.

automatic variable: A dynamic local variable that comes into existence when a function is called and disappears when it is exited.

bank: A 64K (65,536-byte) portion of the Apple IIGS internal memory. An individual bank is specified by the value of one of the 65C816 microprocessor's bank registers.

buffer: An area of memory allocated for reading from or writing to a file.

catalog: See **directory**.

carriage return character (`\r`): A control code (ASCII 13) generated by the Return key; in APW C, equal to **newline (`\n`).**

`char`: An 8-bit character data type whose range is 0 to 255; the same as **`unsigned char`** in APW C.

character: Any symbol that has a widely understood meaning and thus can convey information. Some characters—such as letters, numbers, and punctuation—can be displayed on the monitor screen and printed on a printer. Most characters are represented in the computer as one-byte values.

code segment: An object segment that consists mainly of code. Code segments are provided for programs that differientiate between code and data segments.

command: In the Standard C Library, a parameter that tells a function which of several actions to perform; in the APW Shell, a word that tells APW which utility to execute.

command interpreter: A program that interprets and executes commands. Specifically, the APW shell.

`comp`: A 64-bit SANE data type with signed integral values and one NaN.

compiler: A program that produces object files from source files written in a high-level language such as C.

conditional compilation: Use of preprocessor commands (`#if`, `#ifdef`, `#ifndef`, `#else`, `#endif`) to vary the output depending on compile-time conditions.

C SANE Library: A set of routines that provide extended-precision mathematical functions.

current language: The APW language type that is assigned to a file opened by the APW Editor. If an existing file is opened, the current language changes to match that of the file.

current prefix: The prefix that is used by the APW Shell if a partial pathname is used.

data segment: An object segment that consists primarily of data. Data segments are provided for programs that differentiate between code and data segments.

debugger: A shell utility that lets you step through a program and examine memory as you go.

denormalized number: A nonzero number that is too small for normalized representation.

desk accessory: A program that is accessed from the Apple menu and shares its runtime environment with an application, a utility, or another desk accessory.

diagnostic output: A file used to report errors and diagnostic information. Generally merged with **standard output,** but can be redirected. In APW C, synonymous with **standard error**.

directory: A file that contains a list of the names and locations of other files stored on a disk. Directories are either **volume directories** or **subdirectories**. A directory is sometimes called a *catalog*..

direct page: A page (256 bytes) of bank $00 of Apple IIGS memory, any part of which can be addressed with a short (one byte) address because its high address byte is always $00 and its middle address byte is the value of the 65C816 processor's direct register. Co-resident programs or routines can have their own direct pages at different locations. The direct page corresponds to the 6502 processor's **zero page**. The term *direct page* is often used informally to refer to the lower portion of the **direct-page/stack space**.

direct-page/stack space: A portion of bank $00 of Apple IIGS memory reserved for a program's **direct page** and **stack**. Initially, the 65C816 processor's **direct register** contains the base address of the space, and its **stack register** contains the highest address. In use, the stack grows downward from the top of the direct-page/stack space, and the lower part of the space contains direct-page data.

direct register: A hardware register in the 65C816 processor that specifies the start of the direct page.

dispose: To permanently deallocate a memory block. The Memory Manager disposes of a memory block by removing its master pointer. Any handle to that pointer will then be invalid. Compare **purge**

`double`: A 64-bit floating-point data type with IEEE double precision.

dynamic segment: A segment that can be loaded and unloaded during execution as needed. Compare with **static segment.**

editor: A shell utility for editing source files.

`enum`: An enumerated data type of 8, 16, or 32 bits depending on the range of the enumerated literals.

environment: In SANE, consists of rounding direction, rounding precision, exception flags, and halt settings; in APW, consists of exported variables and other features of the Integrated Environment.

exception: A condition in the SANE environment that can cause a program halt.

Exec file: A file containing APW commands that are executed as if typed on the keyboard.

`exit` function: A function that is registered with `onexit` for execution when the program terminates.

`extended`: An 80-bit floating-point data type with IEEE extended precision; used in C for all intermediate results.

external reference: A reference to a symbol that is defined in another segment. External references must be to global symbols.

fatal error: an error serious enough that the computer must halt execution.

field: A string of ASCII characters or a value that has a specific meaning to some program. Fields may be of fixed length, or may be separated from other fields by field delimiters. For example, each parameter in a segment header constitutes a field.

file-buffered: A buffer style in which characters sent to an output I/O function are queued and written as a block.

file descriptor: A file reference number returned by a `creat` or `open` call.

filename: The string of characters that identifies a particular file within a disk **directory**. ProDOS 16 filenames can be up to 15 characters long, and can specify directory files, subdirectory files, text files, source files, object files, load files, or any other ProODS 16 file type. Compare with **pathname**.

file pointer: A pointer to the next byte to be read or written in a **stream**.

file type: An attribute in a ProDOS 16 file's directory entry that characterizes the contents of the file and indicates how the file may be used. On disk, filetypes are stored as numbers; in a directory listing, they are often displayed as three-character mnemonic codes.

FILE variable: A variable containing information about a stream, including the file descriptor and buffer size, location, and style.

`float`: A 32-bit floating-point data type with IEEE single precision.

flush: Write out the contents of a buffer.

format character: A character that defines the interpretation of the input field in the `scanf` call.

full pathname: The complete name by which a file is specified. A full pathname always begins with a slash (/), because a volume directory name always begins with a slash. See **pathname**.

global label: A symbolic identifier in an object segment, which the linker enters into the relocation dictionaly and the loader replaces with an absolute address.

global symbol: A label in a code segment that is either the name of the segment or an entry point to it. Global symbols may be referenced by other segments. Compare with **local symbol**.

handle: See **memory handle**.

hexadecimal: The base–16 system of numbers, using the ten digits 0 through 9 and the six letters A through F. Hexadecimal numbers can be converted easily and directly to binary form, because each hexadecimal digit corresponds to a sequence of four bits. In C manuals hexadecimal numbers are usually preceded by a `0x`.

high–level language: A programming language that is relatively easy for people to understand. A single statement in a high–level language typically corresponds to several instructions of machine language. Compare **low–level language.**

image: A representation of the contents of memory. A code image consists of machine-language instructions or data that may be loaded unchanged into memory.

include file: A file whose contents will be included with the source file at compile time—it contains function declarations, macros, types, and `#define` directives used by the compiler.

infinity: A SANE representation of mathematical ∞.

`int`: A 16-bit integer data type whose range is –32,768 to 32,767.

interface: The compile-time and runtime linkage between your C program and toolbox routines.

Jump Table: A table contructed in memory by the System Loader from all Jump Table segments encountered during a load. The Jump Table contains all references to dynamic segments that may be called during execution of the program.

K: 1024 bytes

language.command: A command that changes the APW current language.

library file: A file produced by MAKELIB program from object files, generally ones containing functions useful to a number of programs. It can be searched by the Linker for necessary functions, but more quickly than an object file.

LinkEd: A command language that can be used to control the APW Linker.

linker: A program that combines files generated by compilers and assemblers, resolves all symbolic references, and generates a file that can be loaded into memory and executed.

line-buffered: A **buffer** style in which each line of output is queued for writing as soon as a newline character is written.

load file: A file that can be loaded into memory, one load segment at a time, by the System Loader.

load segment: A part of a load file corresponding (in C) to one or more functions. **Object segments** are assigned to load segments at compile time by means of the `overlay` command or at link time by LinkEd commands.

local symbol: A label defined only within an individual segment. Other segments cannot access the label. Compare with **global symbol.**

`long`: A 32-bit integer data type whose range is –2,147,483,648 to 2,147,483,647.

loop: A section of a program that is executed repeatedly until a limit or condition is met, such as an index variable's reaching a specified ending value.

low-level language: A programming language, such as assembly language, that is relatively close to the form the computer's processor can execute directly. One statement in a low-level language corresponds to a single machine-language instruction. Compare **high-level language.**

`main`: The name of the function that is the entry point for every C program.

main segment: The first segment in the initial load file of a program. It as loaded first and never removed from memory until the program terminates.

MakeLib utility: A program that creates library files from object files.

Mark: The current position in an open file. It is the point in the file at which the next read or write operation will occur.

memory block: see **block**

memory handle: The identifying number of a particular block of memory. A memory handle is a pointer to a master pointer to the memory block.

memory image: A portion of a disk file or segment that can be read directly into memory.

Memory Manager: A program in the Apple IIGS Toolbox that manages memory use. The Memory Manager keeps track of how much memory is available, and allocates memory **blocks** to hold program segments or data.

memory-resident: (adj) (1) Stored permanently in memory as firmware (ROM). (2) Held continually in memory even while not in use. For example, ProDOS is a memory-resident program.

movable: A memory block attribute, indicating that the Memory Manager is free to move the block . Opposite of *fixed.* Only **position-independent** program segments may be in movable memory blocks. A block is made movable or fixed through Memory Manager calls.

NaN: Not a Number; a SANE representation produced when an operation cannot yield a meaningful result.

native mode: The 16-bit operating state of the 65C816 processor.

newline character (`\n`): A control code that advances print position or cursor to the left margin of next output line; in APW C, same as **carriage return (`\r`).**

normalized number: A floating-point number that can be represented with a leading significand bit of 1.

number class: In SANE, a floating-point number can be characterized as either zero, normalized, denormalized, infinity, or NaN.

numeric environment: In SANE, the rounding direction, rounding precision, halt enables, and exception flags.

object segment: A part of an object file corresponding (in C) to a single function.

object file: The output from an assembler or compiler and the input to the linker. In APW an oxject file contains both machine-language instructions and instructions for the linker. Compare with **load file.**

object module format (OMF)**:** The general format used in object files, library files, and load files.

object segment: A segment in an object file.

OMF: Object module format.

OMF file: Any file in object module format.

page: (1)A portion of Apple IIGS memory that is 256 bytes long and that begins at an address that is an even multiple of 256. A memory block whose starting address is an even multiple of 256 is said to be *page aligned.* (2) An area of main memory containing text or graphical information being displayed on the screen.

parameter: A value passed to or from a command, function, or other routine.

Pascal-style function: A function using Pascal-style calling conventions that can be declared in C using the `pascal` specifier.

partial assembly: A procedure whereby only specific segments of a program are assembled. If you have performed one full assembly followed by one or more partial assemblies on a program, the linker extracts only the latest version of each object segment to be included in the load file.

partial compile: A procedure whereby only specific segments of a program are compiled. If you have performed one full assembly followed by one or more partial compiles on a program, the linker extracts only the latest version of each object segment to be included in the load file.

partial pathname: A **pathname** that includes the **filename** of the desired file but excludes the volume directory name (and possibly one or more of the subdirectories in the path). It is the part of a pathname following a **prefix**—a prefix and a partial pathname together constitute a **full pathname.** A partial pathname does not begin with a slash because it has no volume directory name.

patch: To replace one or more bytes in memory or in a file with other values. The address to which the program must jump to execute a subroutine is *patched* into memory at load time when a file is **relocated.**

pathname: The full name of a file, including its volume name and directory names.

pointer: A memory address at which a particular item of information is located. For example, the 65C816 Stack register contains a pointer to the next available location on the stack.

position-independent: Code that is written specifically so that its execution is unaffected by its position in memory. It can be moved without needing to be relocated.

position-independent segment: A load segment that is movable when loaded in memory.

prefix: A portion of a **pathname** starting with a volume name and ending with a subdirectory name. It is the part of a full pathname that precedes a **partial pathname**—a prefix and a partial pathname together constitute a full pathname. A prefix always starts with a slash (/) because a volume directory name always starts with a slash.

preprocessor: Part of the C compiler that provides file inclusion, macro substitution, and conditional compilation.

preprocessor symbol: One of a set of constants defined to be 1, equivalent to writing "`#define` *symbol* `1`" at the beginning of the source file.

ProDOS: A family of disk operating systems developed for the Apple II family of computers. *ProDOS* stands for *Professional Disk Operating System*, and includes both ProDOS 8 and ProDOS 16.

ProDOS 8: A **disk operating system** developed for standard Apple II computers. It runs on 6502-series microprocessors. It also runs on the Apple IIGS when the 65C816 processor is in 6502 **emulation mode**.

ProDOS 16: A **disk operating system** developed for 65C816 **native mode** operation on the Apple IIGS. It is functionally similar to **ProDOS 8** but more powerful.

purge: To temporarily deallocate a memory block. The Memory Manager purges a block by setting its master pointer to 0. All handles to the pointer are still valid, so the block can be reconstructed quickly. Compare **dispose**.

purgeable: A memory block attribute, indicating that the Memory Manager may purge the block if it needs additional memory space. Purgeable blocks have different **purge levels**, or priorities for purging; these levels are set by Memory Manager calls.

RAM Disk: A portion of memory (RAM) that appears to the operating system to be a disk volume. Files in a RAM disk can be accessed much faster than the same files on a floppy disk or hard disk.

`register` variable: An **automatic variable** that is allocated to a register. Not used by APW C compiler, because the 65C816 has only a few registers.

scanset: A set of characters allowed in a file scanned by the `scanf` call.

relocate: To modify a file or segment at load time so that it will execute correctly at the location in memory at which it is loaded. Relocation consists of **patching** the proper values into address operands. The loader relocates load segments when it loads them into memory. See also **relocatable code.**

relocatable code: Program code that includes no absolute addresses, and so can be relocated at load time.

relocatable segment: A segment that can be loaded at any location in memory. A relocatable segment can be static, dynamic, or position independent. A load segment contains a **relocation dictionary** that is used to recalculate the values of location-dependent addresses and operands when the segment is loaded into memory. Compare with **absolute segment.**

relocation dictionary: A portion of a load segment that contains relocation information necessary to modify the memory image immediately preceding it. When the memory image part of the segment is loaded into memory, the relocation dictionary is processed by the loader to calculate the values of location-dependent addresses and operands. Relocation dictionaries also contain the information necessary to transfer control to external references.

reference: The name of a segment or entry point to a segmentl; same as *symbolic reference*. To refer to a symbolic reference or to use one in an expression or as an address.

resolve: To find the segment and offset in a segment at which a symbolic reference is defined. When the linker resolves a reference it creates an entry in a **relocation dictionary** that allows the loader to **relocate** the reference at load time.

root filename: The filename of an object file minus any filename extensions added by the assembler or compiler. For example, a program that consists of the object files `MYPROG.ROOT`, `MYPROG.A`, and `MYPROG.B` has the root filename `MYPROG`.

run-time library file: A load file containing program segments--each of which can be used in any number of programs--that the system loader loads dynamically when they are needed.

segment: A component of an OMF file, consisting of a header and a body. In object files, each segment incorporates one or more subroutines. In load files, each segment incorporates one or more object segments.

segment body: That part of a segment that follows the **segment header**, and that contains the program code, data, and relocation information for the segment.

segment header: The first part of a program segment, containing such information as the segment name and the length of the segment.

segment kind: See **segment type.**

segment number: A number corresponding to the relative position of the segment in a file, starting with 1.

segment type: A classification of a segment based on its purpose, contents, and internal structure, as defined in the object module format. The segment type is specified by the KIND field in the segment header.

shell: A program that provides an operating environment for other programs, and that is not removed from memory when the those programs are running. For example, the APW Shell provides a command processor interface between the user and the other components of APW, and remains in memory when APW utility programs are running.

shell call: A request from a program to the APW Shell to perform a specific function.

shell application: A type of program, such as a compiler or shell command, that runs under the APW Shell. Called a **tool** in MPW.

`short`: A 16-bit integer data type whose range is –32,768 to 32,767.

signal: A software interrupt that causes a program to be temporarily diverted from its normal execution sequence.

shell load file: A load file designed to be run under a shell program; shell load files are ProDOS 16 file type $B5.

65C816: The microprocessor used in the Apple IIGS.

source file: An ASCII file consisting of instructions written in a particular language, such as C or assembly language. An assembler or compiler converts source files into **object files**.

stack:. A list in which entries are added (pushed) and removed (pulled) at one end only (the top of the stack), causing them to be removed in last-in, first-out (LIFO) order. The term *the stack* usually refers to the top portion of the **direct-page/stack** space; the top of this stack is pointed to by the 65C816's *Stack register*.

Standard C: A de facto standard based on the most widely used implementation, the Berkleley VAX Portable C Compiler.

Standard C Library: A collection of routines for I/O, string manipulation, data conversion, memory management, and Integrated Environment support.

standard error: A file used to report errors and diagnostic information. Generally merged with **standard output,** but can be redirected. In APW C, synonymous with **diagnostic output.**

standard input: The standard input stream. Generally the keyboard but can be redirected so that input is taken from a file or device.

standard output: The standard output stream. Generally the screen but can be redirected so that input is sent to a file or device.

static segment: A segment that is loaded at program boot time, and is not unloaded or moved during execution. Compare with **dynamic segment.**

stream: A file with associated buffering.

string: An item of information consisting of a sequence of text characters (a *character string*) or a sequence of bits or bytes.

`struct`: A record data type.

subdirectory: A directory within a directory; a file (other than the volume directory) that contains the names and locations of other files. Every ProDOS 16 directory file is either a volume directory or a subdirectory.

symbol: A character or string of characters that represents an address or numeric value; a symbolic reference or a variable.

symbolic reference: A name or label that is used to refer to a location in a program, such as the name of a subroutine. When a program is linked, all symbolic references are **resolved**; when the program is loaded, actual memory addresses are **patched** into the program to replace the symbolic references.

symbol table: A table of symbolic references created by the linker when it links a program. The linker uses the symbol table to keep track of which symbols have been resolved. At the conclusion of a link, you can have the linker print out the symbol table.

tool: An Apple IIGS Toolbox routine.

System Loader: The program that relocates load segments and loads them into Apple IIGS memory. The System Loader works closely with ProDOS 16 and the Memory Manager.

system program: (1) A software component of a computer system that supports **application programs** by managing system resources such as memory and I/O devices. Also called *system software*. (2) Under ProDOS 8, a stand-alone and potentially self-booting application. A ProDOS 8 system program is of file type $FF; if it is self-booting, its filename has the extension .SYSTEM.

token: The smallest unit of information processed by a compiler or assembler. In C, for example, a function name and a left bracket ({) are tokens.

toolbox: A collection of built-in routines on the Apple IIGS that programs can call to perform many commonly–needed functions. Functions within the toolbox are grouped into **tool sets**.

tool set: a related group of (usually firmware) routines, available to applications and system software, that perform necessary functions or provide programming convenience. The Memory Manager, the System Loader, and Quickdraw II are tool sets.

utility: In general, an application program that performs a relatively simple function or set of functions such as copying or deleting files. A APW utility is a program that runs under the APW Shell, and that performs a function not handled by the shell itself. MAKELIB is an example of a APW utility.

unbuffered: A buffer style that does not use a buffer for I/O; reading and writing is done one character at a time.

unload: To remove a load segment from memory. To unload a segment, the System Loader does not actually "unload" anything; it calls the Memory Manager to either **purge** or **dispose** of the memory block in which the code segment resides. The loader then modifies the Memory Segment Table to reflect the fact that the segment is no longer in memory.

unordered: The result of a comparison with a NaN; even identical NaNs compare unordered.

`unsigned char`: An 8-bit character data type whose range is 0 to 255. The same as **`char`** in APW C.

`unsigned int`: A 16-bit integer data type whose range is 0 to 65,535.

`unsigned long`: A 32-bit integer data type whose range is 0 to 4,294,967,295.

`unsigned short`: A 16-bit integer data type whose range is 0 to 65,535.

`void`: A data type used to declare a function that does not return a value.

volume: An item that stores data; the source or destination of information. A volume has a name and a volume directory with the same name. Volumes typically reside in **devices**; a *device* such as a floppy disk drive may contain one of any number of *volumes* (disks).

volume directory: The main directory file of a volume. It contains the names and locations of other files on the volume, any of which may themselves be directory files (called **subdirectories**). The name of the volume directory is the name of the volume. The pathname of every file on the volume starts with the volume directory name.

wildcard character: A character that may be used as shorthand to represent a sequence of characters in a pathname. In APW, the equal sign (=) and the question mark (?) can be used as wildcard characters.

word: A group of bits that is treated as a unit. For the Apple IIGS, a word is 16 bits (2 bytes) long.

WD65816: A predefined symbol identifying C code written to run on the Western Design Center 65SC816 as opposed to another microprocessor.

zero page: The first page (256 bytes) of memory in a standard Apple II computer (or in the Apple IIGS computer when running a standard Apple II program). Because the high-order byte of any address in this part of memory is zero, only a single byte is needed to specify a zero-page address. Compare **direct page.**