Apple® IIGS™ Programmer's Workshop Assembler Reference

APDA Draft
August 6 1987

Apple Technical Publications

This document does not include

- ***final art work***
- ***an index.***

Apple Technical Publications

8/6/87

Engineering Part Number: 030-3131
Marketing Part Number: A2L6001

20525 Mariani Avenue
Cupertino, California 95014
(408) 996-1010

Simultaneously published in the United States and Canada.

Contents

Preface-1 Preface
Preface-1 Roadmap to the Apple IIGS Technical Manuals
Preface-3 Introductory Manuals
Preface-3 The Technical Introduction
Preface-3 The Programmer's Introduction
Preface-3 Machine Reference Manuals
Preface-3 The Hardware Reference Manual
Preface-4 The Firmware Reference Manual
Preface-4 The Toolbox Manuals
Preface-4 The Programmer's Workshop Manual
Preface-5 Programming Language Manuals
Preface-5 Operating System Manuals
Preface-5 All-Apple Manuals
Preface-6 How to Use This Book
Preface-6 What This Manual Contains
Preface-7 What To Read When
Preface-8 Visual Cues
Preface-8 Language Notation
Preface-9 Conventions

PART I: PROGRAMMING GUIDE

1-1 Chapter 1: Overview
1-1 The Apple IIGS Programmer's Workshop
1-1 About The APW Shell
1-2 About the APW Editor
1-2 About the APW Assembler
1-2 Instruction Sets
1-3 Directives
1-3 Macros
1-3 Standard Apple Numeric Environment
1-4 APW Assembler Concepts
1-4 Source, Object, and Load Files
1-4 Symbolic References
1-4 Absolute and Relocatable Load Files
1-5 Converting Source Code into Executable Code
1-6 Program Segments
1-7 File Segments and Partial Assemblies
1-7 OMF File Format
1-7 Static and Dynamic Segments
1-7 Library Files
1-8 Backing Up Your APW Disk

2-1 Chapter 2: Using The APW Assembler
2-1 Writing and Running A Sample Program
2-2 The Assembly Process
2-3 The Assembly Listing
2-4 Pausing The Assembly
2-4 Stopping The Assembly

2-4 Assembler Error Messages
2-4 Printer Listings
2-5 APW Shell Commands
2-5 Editing Files
2-5 Assembling, Linking, and Running Programs
2-6 ASSEMBLE
2-6 ASML
2-7 ASMLG
2-7 LINK
2-7 RUN
2-7 Command Parameters
2-10 Appending Files
2-10 Partial Assemblies
2-11 The APW Linker
2-11 Making Library Files
2-13 The Apple IIGS Debugger and DumpOBJ

PART II: LANGUAGE REFERENCE

3-1 Chapter 3: APW Assembly Statements
3-1 Assembly Statement Formats
3-1 Labels
3-2 Label Scope
3-2 Case Sensitivity in Labels
3-3 Attributes and Labels
3-3 Count Attribute
3-3 Length Attribute
3-3 Type Attribute
3-4 Operation Code
3-4 65816 Instruction Set
3-11 APW Assembler Directives
3-11 APW Assembler Macro Calls
3-12 Operands
3-12 Instruction Operand Formats
3-16 Expressions
3-22 Comments
3-22 The Comment Field
3-22 Comment Lines

4-1 Chapter 4: APW Assembler Directives
4-1 Directive Formats
4-1 Directive Functions
4-1 Program Control Directives
4-2 Data Definition Directives
4-2 Symbol Definition Directives
4-2 Code Location Directives
4-2 File Control Directives
4-3 APW Assembler Option Directives
4-3 Listing Option Directives
4-3 The Comment Field and APW Assembler Directives
4-4 Settings Attribute and APW Assembler Directives
4-4 APW Assembler Directives
4-4 ABSADDR
4-6 ALIGN

4-6	ANOP
4-6	APPEND
4-7	CASE
4-8	CODECHK
4-8	COPY
4-9	DATA
4-9	DATACHK
4-10	DC
4-11	Address (A*x*)
4-11	Binary (B)
4-11	Character (C)
4-12	Hexadecimal (H)
4-12	Integers (I*x*)
4-13	Floating Point (F, D, E)
4-14	Reference an Address (R)
4-14	Soft Reference (S*x*)
4-14	DIRECT
4-14	DS
4-15	DYNCHK
4-15	EJECT
4-16	END
4-16	ENTRY
4-17	EQU
4-18	ERR
4-18	EXPAND
4-19	GEN
4-19	GEQU
4-20	IEEE
4-20	INSTIME
4-21	KEEP
4-21	KIND
4-22	LIST
4-23	LONGA
4-23	LONGI
4-24	MERR
4-24	MSB
4-25	NUMSEX
4-25	OBJ
4-26	OBJCASE
4-26	OBJEND
4-26	ORG
4-27	PRINTER
4-28	PRIVATE
4-28	PRIVDATA
4-29	RENAME
4-29	SETCOM
4-30	65C02
4-30	65816
4-30	START
4-31	SYMBOL
4-32	TITLE
4-32	TRACE
4-34	USING

5-1 Chapter 5: Using Macros in Assembly-Language Programs
5-1 Macro Definition
5-1 Macro Formats
5-2 Macro Expansions
5-3 Assembler Directives Used With Macros
5-3 MCOPY
5-3 MDROP
5-3 MLOAD
5-4 Macro and Equate Directory
5-4 Predefined Macro Files
5-5 Predefined Equates
5-5 M16.UTIL Macros
5-7 ADD
5-7 ADD4
5-8 ASL4
5-9 DEC4
5-9 DP
5-9 EMULATION
5-10 INC4
5-10 LONG
5-10 LONGM
5-11 LONGX
5-11 LSR4
5-12 NATIVE
5-12 PULLLONG
5-13 PULL1
5-15 PULL3
5-15 PULLWORD
5-16 PUSHLONG
5-17 PUSH1
5-18 PUSH3
5-19 PUSHWORD
5-20 READCH
5-21 SHORT
5-21 SHORTM
5-21 SHORTX
5-21 STR
5-22 SUB
5-23 SUB4
5-23 WRITECH
5-24 WRITELN
5-25 WRITESTR
5-25 Building Your Own Macro Library
5-26 Assembling a Program That Contains Library Macros
5-27 Listing Options

6-1 Chapter 6: Writing Macros
6-1 Macro Definition
6-2 Conditional Assembly Directives
6-3 Symbolic Parameters
6-3 Set Symbols
6-4 Using Symbolic Parameters
6-5 Symbolic Parameter Substitution
6-6 Symbolic Parameter Scope

6-6 Assigning Values to Symbolic Parameters
6-6 Positional Parameters
6-6 Keyword Parameters
6-7 Character Type Symbolic Parameters
6-7 Subscripted Symbolic Parameters
6-8 Conditional Assembly and Macro Directives
6-8 ACTR
6-9 AGO
6-10 AIF
6-10 AINPUT
6-11 AMID
6-12 ASEARCH
6-12 GBLA
6-13 GBLB
6-14 GBLC
6-14 LCLA
6-15 LCLB
6-15 LCLC
6-16 MACRO
6-16 MEND
6-16 MEXIT
6-17 MNOTE
6-18 SETA
6-18 SETB
6-19 SETC
6-20 &SYSCNT
6-21 &SYSDATE
6-22 &SYSNAME
6-22 &SYSTIME
6-22 Attributes and Symbolic Parameters
6-23 Count Attribute
6-24 Length Attribute
6-24 Type Attribute

PART III: Appendixes

Appendix A: 65816 Instruction Mnemonics and Addressing Modes

Appendix B: The ASCII Character Set

Appendix C: Error Messages

Glossary

List of Figures

P-1. Roadmap to the Technical Manuals
1-1. Creating and Assembling an Executable Program on the Apple IIGS
3-1. Syntax of a Compound Expression
3-2. Syntax of a Simple Expression
3-3. Syntax of a Term
3-4. Syntax of a Factor
3-5. Syntax of a Constant
3-6. Syntax of a Binary Number
3-7. Syntax of an Octal Number
3-8. Syntax of a Decimal Number
3-9. Syntax of a Hexadecimal Number
3-10. Syntax of a Character Constant

List of Tables

P-1. The Apple IIGS Technical Manuals
3-1. Operand Symbol Types
3-2. The 65816 Addressing Mode Operand Syntax
5-1. Predefined Macro Files in the AINCLUDE Directory
5-2. Equate Files in the AINCLUDE Directory

Preface

The *Apple® IIGS™ Programmer's Workshop Assembler Reference* manual is intended for all programmers writing Apple IIGS assembly-language programs. You should be familiar with assembly-language programming and the *Apple IIGS Programmer's Workshop Reference* manual before reading this manual.

This manual assumes that you are familiar with the 6502 and 65816 assembly language instructions and addressing modes. Three commercially available reference sources for this information are the following:

- David Eyes and Ron Lichty, *Programming the 65816*, Simon and Schuster, 1986
- Michael Fisher, *65816/65802 Assembly Language Programming*, Osborne McGraw-Hill, 1986
- William Labiak, *Programming the 65816*, Sybex, 1986

Roadmap to the Apple IIGS Technical Manuals

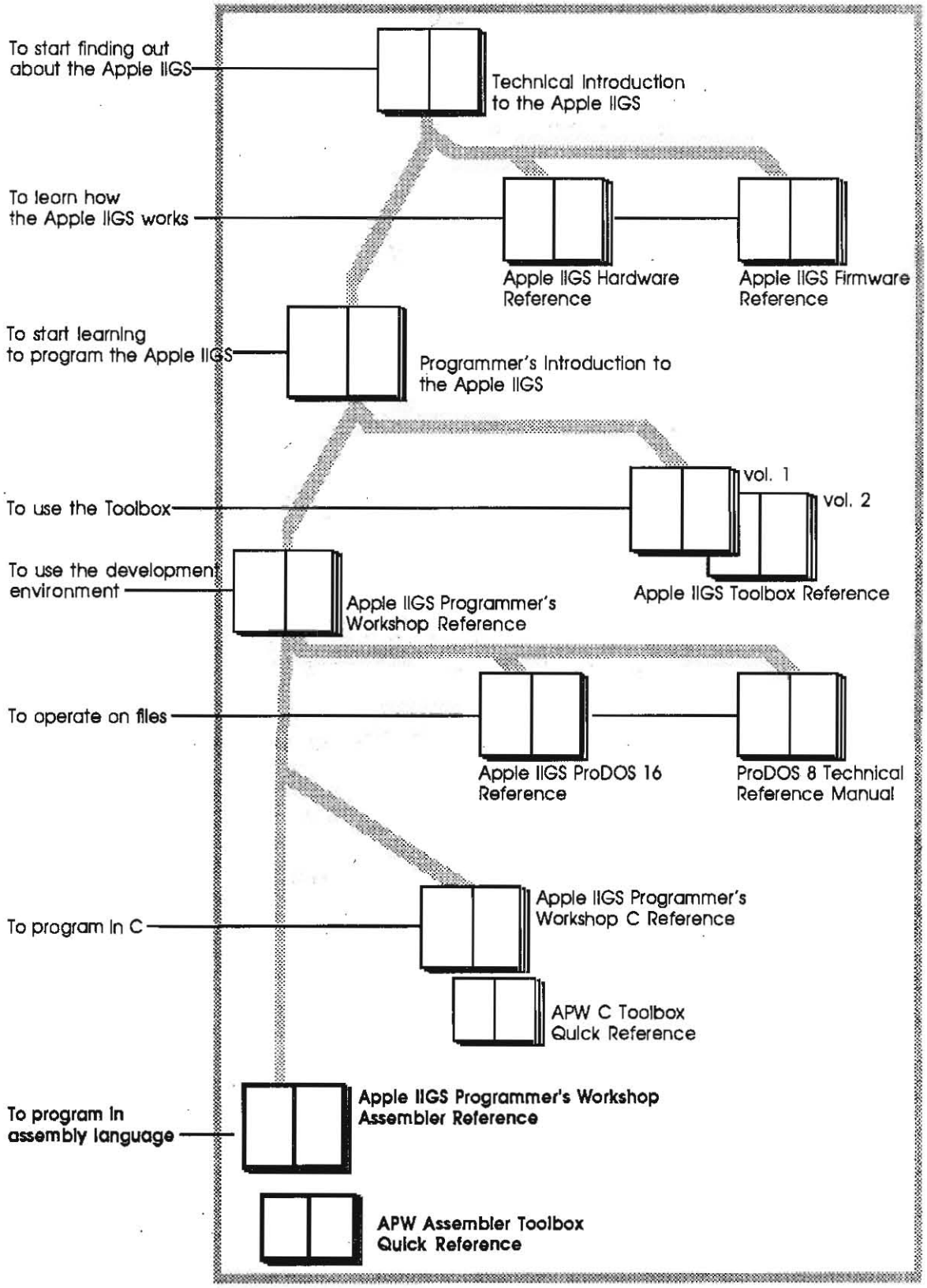
The Apple IIGS computer has many advanced features, making it more complex than earlier models of the Apple II. To describe it fully, Apple has produced a suite of technical manuals. Depending on the way you intend to use the Apple IIGS, you may need to refer to a select few of the manuals, or you may 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 among 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 Technical Reference Manual	Standard Apple II operating system
Apple IIGS ProDOS 16 Reference	Apple IIGS operating system and loader
IIuman Interface Guidelines	Guidelines for the desktop interface
Apple Numerics Manual	Numerics for all Apple computers

Figure P-1. Roadmap to the Technical Manuals



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, you may 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.

The Programmer's Introduction

When you start writing Apple IIGS programs, 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 that use the Apple desktop interface (with windows, menus, and the mouse). 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.

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 and 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 Apple IIGS Programmer's Workshop (APW) is the development environment for the Apple IIGS computer. APW is a set of programs that enables developers to create and debug application programs on the Apple IIGS. The *Apple IIGS Programmer's Workshop Reference* (APW Reference) includes information about the APW Shell, Editor, Linker, Debugger, and utility programs; these are the parts of the workshop that all developers need, regardless of which programming language they use.

The APW Reference manual describes the way you use the workshop to create an application and includes a sample program to show how this is done. In addition, this manual documents the APW Shell to provide the information necessary to write an APW utility or a language compiler for the workshop.

Included in the APW Reference manual are complete descriptions of two standard Apple IIGS file formats: the text file format and the object module format. The text file format is used for all files written or read as "standard ASCII files" by Apple IIGS programs running under ProDOS 16. The object module format is used for the output of all APW compilers and for all files loadable by the Apple IIGS System Loader.

Programming Language Manuals

Apple currently provides a 65816 assembler and a C compiler. Other compilers can be used with the workshop, provided that they follow the standards defined in the *Apple IIGS Programmer's Workshop 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 use the assembler or compiler for that language. The manuals for the languages Apple provides are the *Apple IIGS Programmer's Workshop Assembler Reference* (this book) and the *Apple IIGS Programmer's Workshop C Reference.*

Note: The *Apple IIGS Programmer's Workshop Reference* and the two programming language manuals are available through the Apple Programmer's and Developer's Association.

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 Technical Reference Manual* and *Apple IIGS ProDOS 16 Reference.* ProDOS 16 uses the full power of the Apple IIGS. The ProDOS 16 manual describes its features and 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 the standard operating system for most Apple II computers with 8-bit centrol processors. It also runs on the Apple IIGS. As a developer of Apple IIGS programs, you need the *ProDOS 8 Technical Reference Manual* 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 *Apple Numerics Manual.* If you develop programs for any Apple computer, you should know about those manuals.

The *Human Interface Guidelines* describes Apple's standards for the desktop interface of any program that runs on Apple computers. If you are writing a commercial application for the Apple IIGS, you should be 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 Binary Floating-Point Arithmetic* (IEEE Std 754-1985). The functions of the Apple IIGS SANE tool set match those of the Macintosh SANE package 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 routines in your programs. The *Apple Numerics Manual* is the comprehensive reference for the SANE numerics routines.

How to Use This Book

This section describes the contents of the *Apple IIGS Programmer's Workshop Assembler Reference* manual. Following a brief chapter-by-chapter description of this book's contents, the section "What to Read When" gives you guidelines about which sections you should read for specific purposes. Finally, the section called "Visual Cues" describes the notations used in this book to alert you to important material, or material with special significance.

What This Manual Contains

This manual is divided into three parts: an introduction to the APW Assembler containing two chapters; a four-chapter reference section to the APW directives and macros; and three appendixes with more reference material that you will need occasionally. There is also a glossary and an index.

Part I, "Programming Guide," gives you the minimum information that you need to be able to use the APW Assembler.

- Chapter 1, "Overview," introduces the environment in which youll use the APW Assembler. It discusses the Apple IIGS Programmer's Workshop (APW), ProDOS 16, the Apple IIGS Toolbox, and lists the hardware and software you need.
- Chapter 2, "Using The APW Assembler," steps through a sample session with the assembler, describes the assembly process, lists the shell commands you need to work with the assembler, and discusses the linker and other utilities.

Part II, "Language Reference," is a detailed description of the Apple IIGS Programmer's Workshop Assembler directives and macros.

- Chapter 3, "APW Assembly Statements," describes the syntax of APW Assembler labels, operation codes, operands, and comments.
- Chapter 4, " Assembler Directives," describes each of the APW Assembler directives, with their syntax and some examples of how they are used.
- Chapter 5, "Using Macros in Assembly Language Programs," tells you how to use macros provided with APW in your assembly-language programs; how to use the MacGen utility to build your own macro library; and how to use the macro directives: MCOPY, MDROP, and MLOAD.
- Chapter 6, "Writing Macros," tells you how to write your own macros and include them in your source text.

Part III includes three appendixes of reference material.

- Appendix A is a summary of the 65816 mnemonics and addressing modes. This appendix also gives you the number of bytes that each mnemonic requires for each addressing mode.
- Appendix B is a table of ASCII characters.
- Appendix C a list of the error messages that can be generated by the APW Assembler. The appendix includes a brief description of the most probable cause for the error and some hints about what you can do to correct the problem.

The Glossary defines all of the technical terms listed in boldface type throughout this manual.

What To Read When

To get the most out of the APW Assembler, you should use this manual as efficiently as possible. Here are some suggestions on how to proceed.

1. Whatever your background and experience, start with the next section, "Other Materials You Will Need" and Chapter 1. The Apple IIGS is not quite like any other computer, so you need to become familiar with the peculiarities of the Apple IIGS and the APW concepts that the assembler uses before you proceed.
2. Read through Chapter 2 to get an idea of the assembly process and the APW Shell commands that you want to use, then go to Chapter 4 for complete descriptions of the APW Assembler directives.
3. If this is your first experience with assembly language programming, go to Chapter 1 first for the concepts and a description of how it all fits together. Then look at one of the commercially available texts on the instruction sets. Chapter 3 of this manual contains a summary of the instructions that you can use as a reference once you understand how they work. When you understand the instruction set, go to Chapter 2 for the assembly process and then to Chapter 4 for the APW Assembler directives.
4. If you are doing complex programming and writing your own macros, read Chapter 6, "Writing Macros." This chapter contains information on macro formats, addressing modes, and data formats. For routine programming, the material in Chapter 5, "Using Macros in Assembly Language Programs," is all that you need to read about macros.

Visual Cues

Certain conventions in this manual provide visual cues that alert you to special material, for example the introduction of a new term or especially important information. Look for these visual cues throughout the manual:

> **Note:** Text set off in this manner presents sidelights or interesting points of information.

> **Important:** This heading tells you that there is important information that you should read before proceeding.

> **Warning:** A message such as this directs your attention to something that could cause loss of data or damage to the software.

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 previously and that there is an entry for it in the glossary.

Language Notation

A special typeface is used for characters that you type, or that can appear on the screen, such as commands, instructions, directives, and macro calls.

```
IT LOOKS LIKE THIS.
```

> **Note:** APW Assembler directives and macros are presented in upper case in this manual. This is only so that you can find them easily; the APW Assembler is not sensitive to case unless you use the `CASE` directive described in Chapter 4. Otherwise, `PARMBLOCK` and `ParmBlock` mean the same thing.

Italics are used in commands to indicate parameters that must be replaced with a value; for example, in the command

`COPY` *filename*

The word *filename* refers to any valid APW Assembler filename.

Apple IIGS Upgrade: The Apple IIGS Apple key, indicated by the open Apple icon (), corresponds to the Open-Apple key on the Apple IIe keyboard. The Apple IIGS Option key corresponds to the Solid-Apple key () on the Apple IIe keyboard. The Clear and Enter keys on the Apple IIGS keyboard have no Apple IIe equivalents.

> **Important:** The Apple IIGS keyboard Reset key has a triangle on it rather than the word *reset*.

Conventions

The following additional conventions are observed in this manual:

[]	Square brackets indicate that the enclosed item is optional.
|	A vertical bar indicates a choice. For example, `+L | -L` indicates that a command can be entered as either `+L` or as `-L`.
. . .	A horizontal ellipsis indicates that the preceding item(s) can be repeated as necessary.
. . .	A vertical ellipsis, or a pair of horizontal ellipses, indicate that not all of the statements in an example or figure are shown.

Part I

Programming Guide

Chapter 1

Overview

This chapter introduces you to the APW Assembler and describes a variety of features and concepts that you must understand in order to write application programs for the Apple IIGS computer. The material that follows

- introduces the Apple IIGS Programmer's Workshop
- tells you about some of the features of the APW Assembler
- describes how important concepts are implemented on the Apple IIGS
- shows you how the programs in the workshop interact
- lists system requirements
- tells you how to back up your system
- tells you how to begin running APW

The Apple IIGS Programmer's Workshop

The Apple IIGS Programmer's Workshop (APW) is the development environment in which you write your Apple IIGS application programs. This development environment includes the **APW Shell, APW editor, APW linker,** and a set of utilities. APW supports both assembly language and C. Additional development support is provided by a comprehensive set of routines known as the **Apple IIGS Toolbox.** These routines can be accessed by a program running under APW; however, the Toolbox is not considered part of APW. For a comprehensive description of APW, refer to the *Apple IIGS Programmer's Workshop Reference* manual. For more information on the Apple IIGS Toolbox, refer to the *Apple IIGS Toolbox Reference:* Volumes 1 and 2.

About The APW Shell

The APW Shell provides the interface that allows you to execute commands and call programs. It does the file management tasks of copying and deleting files and listing directories. You can use shell commands to perform the following functions:

- call the APW Editor to create new files or edit your source code
- call the APW Assembler
- override some APW Assembler source code directives
- call the APW Linker
- call APW utility programs
- call the Apple IIGS Debugger program

The most important shell commands to assembly-language programmers are summarized in Chapter 2 of this manual. Complete descriptions of all the APW Shell commands are contained in the *Apple IIGS Programmer's Workshop Reference* manual.

About the APW Editor

The Apple IIGS Programmer's Workshop Editor is a full screen text editor that can perform such tasks as

- deleting, copying, and moving text
- searching for a text string
- jumping from one position in the file to another
- scrolling the screen up or down
- setting and clearing tab stops
- restoring accidentally deleted text

The Apple IIGS Programmer's Workshop Editor is fully described in the *Apple IIGS Programmer's Workshop Reference.*

About the APW Assembler

The APW Assembler is a powerful macro assembler. Among the features of this assembler are

- support for source code in 65816, 65C02, and 6502 instruction sets
- an extensive group of assembler directives
- a comprehensive set of macros to access the Apple IIGS Toolbox, interface with ProDOS and the shell, and perform I/O
- support for user defined libraries and macros
- full compatibility with the Standard Apple Numeric Environment (SANE).
- conformity with the Apple IIGS object module format, which allows you to link a **program segment** written in assembly language with segments written in high-level languages
- support for partial assemblies
- backward compatibility with ORCA/M for 8-bit Apple II computers

Because of these features, the APW Assembler gives you the means to write relocatable code, divide your assembly-language programs into segments, link your assembly-language segments to segments written in high-level languages, and use macros.

Instruction Sets

The 65C816 processor of the Apple IIGS computer can operate in either native or emulation mode. The full instruction set of the 65C816 processor is available in **native mode**. The 91 instructions combined with 25 addressing modes make 255 operation codes available to

programs. The register set can be used for either 8-bit or 16-bit operations. The accumulator can be set to either a 16-bit or 8-bit register. The advantage of using a processor with 16-bit registers over a processor with 8-bit registers is shorter programs that run faster. Such programs cannot be run on 8-bit Apple II computers, however. This book assumes that you are writing programs to be run in native mode under ProDOS 16 on the Apple IIGS computer.

In **emulation mode**, the 65C816 processor behaves exactly like a 6502 processor, including register configurations, stack location, and instruction timing. This means that you can run programs written for 8-bit Apple computers (such as the Apple IIe and Apple IIc) on the Apple IIGS computer. Native and emulation modes are discussed in the *Technical Introduction to the Apple IIGS*, and described in detail in the *Apple IIGS Hardware Reference* manual.

Directives

The APW Assembler **directives** allow you to perform a variety of functions. These include

- data definition
- symbol definition
- space allocation
- output listing options
- conditional assemblies
- macro expansion listings

Macros

The Apple IIGS Programmer's Workshop provides a comprehensive set of **macros** that you can use to access the Apple IIGS Toolbox, make calls to ProDOS 16 and the shell, and perform I/O. The Apple IIGS Toolbox includes routines to handle memory management, menu management, QuickDraw II routines, support for the **Standard Apple Numeric Environment**, and more. The Apple IIGS Programmer's Workshop also provides full support for user-defined libraries and macros.

Standard Apple Numeric Environment

The APW Assembler supports the Standard Apple Numeric Environment (SANE). SANE is based on the IEEE Standard 754 for Binary Floating-Point Arithmetic, which 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.

APW Assembler Concepts

Many of the concepts described here will seem familiar to you from work with other computers, but be careful: the way in which they are implemented on the Apple IIGS may be different. To get the most out of the Apple IIGS Programmer's Workshop Assembler and to use the operating system and the memory of the Apple IIGS efficiently, you must understand and be able to use them in a new context. The descriptions that follow here concern only the APW Assembler; if you are writing a program that contains segments in high-level languages, see the *Apple IIGS Programmer's Workshop Reference* for directions on how to combine the segments.

Source, Object, and Load Files

The APW Assembler uses three types of files: **source files**, **object files**, and **load files**. Source files are ASCII files containing 65816 or 65C02 assembly-language code and data. A source file consists of either 65816 instructions or 6502 instructions, APW Assembler directives, macros, and the data needed by the program. The APW Assembler uses source files to produce object files.

Object files are binary files created by the APW Assembler and used by the APW Linker. They are created by the APW Linker and loaded into computer memory by the System Loader. Object files contain global symbol definitions, references to symbols in other object files, the machine code translation of the source program, and the information needed by the APW Linker to combine multiple object files.

Load files are also binary files. Load files are executable files consisting of the combined object files and segments from library files.

Symbolic References

The APW Assembler supports labeling specific instructions, subroutines, or blocks of data with names. You can then refer to the name in another part of the program. For example, when you want to execute a subroutine, you generally refer to the subroutine by name. A name or label of code or data used in this way is called a symbolic reference (that is, a *symbol* that can be *referenced* or referred to). The assembler provides three kinds of symbolic references: local labels, global labels, and symbolic parameters. See the description of labels in Chapter 3 and the description of symbolic parameters in Chapter 6 for more information.

Absolute and Relocatable Load Files

The APW Assembler ORG directive allows you to specify actual locations in computer memory where you want source code to execute. A `JMP $2000` causes the instruction at this location to be executed. Files that contain source code whose location in memory is specified when the program is written or linked are called *absolute load files*. For program code like this to run, the loader must load the instructions and data at the given location or not at all.

Relocatable load files, on the other hand, result when you write a program in which every reference to a location in the program is relative to another location, or is made through a symbolic reference. Such program code is loaded into memory by the System Loader. The actual memory addresses to which jumps (and other instructions that require absolute locations) must be made are patched into the program by the System Loader.

The Apple IIGS System Loader and **Memory Manager** are designed to support relocatable load files. Because desk accessories, shell programs, RAM-based tools, and so forth, are placed in memory by the System Loader and Memory Manager, absolute code is likely to conflict with other code already in memory. The APW Assembler is designed to work with relocatable code. Do not write absolute code unless you want to cause untold grief to yourself and the people who use your program.

Converting Source Code into Executable Code

Converting source code into 65816 machine-language instructions and data that is resident in memory is done in the three main steps, shown in Figure 1-1 and described in the material that follows.

1. The source code is assembled. The APW Assembler processes the source file in two passes, as described in Chapter 2. The output from the assembler is an object file consisting of 65816 machine-language instructions, data, and unresolved symbolic references.

   Your program can consist of several source files, each of which is converted into one or more object files by the APW Assembler.

2. The object files are passed to the APW Linker, which can combine them into a single load file and resolve the 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, it either automatically searches all of the library files with APW library prefixes, or searches the libraries that you specify with a LinkED command file. The linker replaces symbolic references with entries in relocation dictionary tables. The load file consists of blocks of machine language code that can be loaded directly into memory, 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 System 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 put 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 that can be processed independently.

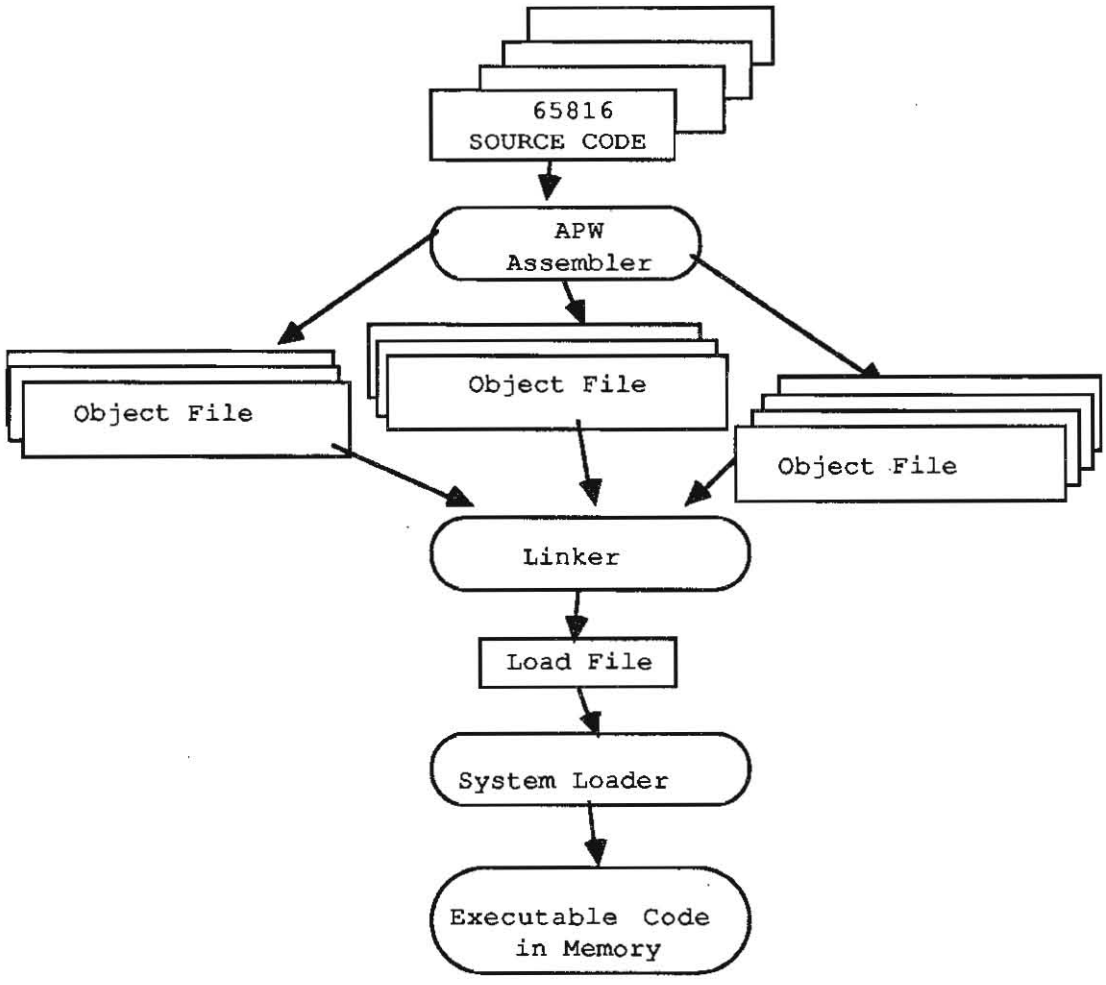


Figure 1-1. Creating and Assembling an Executable Program on the Apple IIGS

Program Segments

The APW Assembler allows you to segregate large blocks of code or data into segments. Individual segments can then be assembled and linked independently. Source code segments are defined by either START, END, or PRIVATE, END assembler directives. Data segments are defined by either DATA, END directive pairs or PRIVDATA and END directives. The code between the START, PRIVATE, PRIVDATA or DATA directive and the next END directive defines a segment. The APW Assembler allows you to assign unique labels to each START, PRIVATE, DATA, or PRIVDATA directive. The label is the source code segment name.

When you assemble source code, each source code segment becomes an object segment and the source code segment name is carried to the object segment. You can determine which object segments you want to include in any load segment by specifying a load segment name in your START, DATA, PRIVATE, or PRIVDATA directive. All object segments with the same load segment names are automatically included in the same load segment, unless you are using a LinkED command file. In this case you must assign segments as described in the *Apple IIGS Programmer's Workshop Reference* .

The advantage of dividing your code and data into segments is that it may be possible to load a program that is divided into several small segments into memory when the same program in one or two load segments would not 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 is loaded, and patches intersegment calls in each segment as it is loaded.

The other advantage of dividing your code and data into segments is that you can take advantage of the APW Assembler's ability to do **partial assemblies.**

File Segments and Partial Assemblies

Each source file segment can be assembled independently by the APW Assembler in a process referred to as a partial assembly. Each object file segment can be linked independently by using a LinkEd command file. LinkEd command files are described in the *Apple IIGS Programmer's Workshop Reference.*

OMF File Format

All APW Assembler object and load files are in object module format (OMF). Every OMF file consists of one or more segments; each segment consists of a segment header and the segment body. The segment header is divided into fields, one of which contains the name of the segment and another that specifies the type and attributes of the segment. The type and attribute field can be set with the assembler `KIND` directive described in Chapter 4 of this book. The complete description of object module format is contained in the *Apple IIGS Programmer's Workshop Reference.*

Static and Dynamic Segments

Static and dynamic segment attributes can be set with the `KIND` directive. The default setting for all assembler segments is static. A **static segment** is loaded into memory at program boot time, and is not unloaded or moved during execution. The first segment of all programs must be static. Other segments of the same program may be static, but (especially for large programs) the system will use memory more efficiently if infrequently used segments are dynamic.

A **dynamic segment** is a load segment that can be loaded automatically by the System Loader and Memory Manager while your program is executing. A dynamic segment can also be removed from memory if the space is needed to load another segment. See the *Apple IIGS ProDOS 16 Reference* for directions.

Library Files

Library files contain routines that are useful to many different programs. All APW library files are in object module format. 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. The linker either automatically searches all of the library files with the APW library prefix or searches the library files that you specify, if you use a LinkEd command file. You can create your own library files from one or more object files by using the MakeLib APW utility program.

Backing up Your APW Disk

It is important to make a copy of your APW disk and to run APW from the copy only. Keep the original disk in a safe place so you can make a new copy if something happens to the one you have been using.

You can make a copy of your APW disk by using any system utility program or desktop program you prefer, or you can use APW commands to do the job.

Chapter 2

Using The APW Assembler

This chapter consists of four sections that describe how to use the Apple IIGS Programmer's Workshop Assembler. The first section, "Writing and Running a Sample Program," steps through a sample session, giving you a fast way to become acquainted with assembling, linking, and executing a program. The second section, "The Assembly Process," describes the assembler listing formats, when error messages are generated, and how to redirect the program assembly output from the screen to the printer. The third section, "APW Shell Commands," describes the shell commands and options you'll use to assemble, link, and run your program; build a macro library; make a dictionary segment; and do partial assemblies. The fourth section describes the MakeLib utility program and the Apple IIGS Debugger.

Writing and Running A Sample Program

When you edit an existing source file, the editor is automatically set to the language in which the file was originally written. To edit the file, type the following command at the shell prompt, #:

```
EDIT filename
```

When you open a new file, the editor is set to the last language you used. To change the language type to 65816 instructions and open a new file named `MYPROG`, type the following commands:

```
ASM65816
EDIT MYPROG
```

Use the APW Editor to type in this source code, which when assembled, linked, and run, displays a simple message.

```
            KEEP   MYPROG.1          New name so linker doesn't
;                                    overwrite source file.
            MCOPY  HW.MACROS         Make macro file accessible
;                                    to program.
            LIST   ON                Show source code.
MAIN        START                    Begin the segment.
            PHK                      Push code bank.
            PLB                      Data bank=code bank
            WRITELN #'Hello World!'  Macro call
            LDA    #0                Clear error return.
            RTL                      Return to the shell.
            END                      End the segment.
```

To exit from the editor and save your program after you have typed in the code, do the following:

Press ⌘-Q or Control-Q.	Causes the editor's Quit menu to appear.
Press S	Saves the file to the filename used when the editor was entered.
Press E	Exits from the editor and returns to the shell.

To create your own macro file `HW.MACROS` containing the single macro `WRITELN`,type in the following command:

```
MACGEN MYPROG HW.MACROS 2/AINCLUDE/M16.UTIL
```

Now, when you run your program, the assembler will search your macro file instead of the larger system `M16.UTIL` file. Chapter 5 contains more information on how to build custom macro library files.

To assemble, link, and run the program you have just entered, type the following command:

```
RUN MYPROG
```

The shell checks the language type of `MYPROG` and calls the APW Assembler. The APW Assembler uses your source code as input and produces an object file as output. When the assembly is finished, control returns to the shell. The shell calls the linker to resolve all references and write the load file. The shell then executes the program, and the following message appears on the screen:

```
Hello, World!
```

The Assembly Process

The APW Assembler processes the source code in your program one segment at a time. Each segment goes through two passes. During the first pass, the assembler resolves **local labels** and lines that appear outside of program segments that do not contain labels. When the assembler encounters an `END` directive, it begins pass two.

When the APW Assembler begins pass two, it starts over at the beginning of your segment, as defined by a `START`, `PRIVATE`, `PRIVDATA`, or `DATA` directive. During this pass, both the object code and the assembly listing are produced, local labels having already been resolved in pass one. External labels are resolved at $8000, possibly with some offset value. External **direct page** offset labels, indicated in the source listing by a < character before the expression, are resolved at $80.

The output file from the second pass of the APW Assembler is in object module format (OMF). Each APW language compiler produces object code in object module format, allowing you to link together segments written in different languages. Object module format is discussed in detail in the *Apple IIGS Programmer' s Workshop Reference.*

When the APW Assembler finishes processing a segment, it generates an alphabetic list of the local labels in the segment. The assembler then begins processing the next segment. When all segments have been processed, the assembler lists the **global labels** and returns control to the shell. Depending on the command you used to invoke the Assembler, the shell will either pass control on to the linker or prompt you for your next input. If the linker is called, it uses the object files produced by the APW Assembler, plus any other object files that you specified on the shell command line or in a LinkEd file, as input. The linker combines object files and resolves global labels, giving you an executable load file as output.

The Assembly Listing

The APW Assembler default does not produce a listing on the screen during pass two. If the `LIST` directive is set to `OFF` in the source code, or the shell command-line parameter `-L` is used, each segment as it is processed produces two messages announcing the segment name and pass number. Any error messages will also be produced. If you instruct the assembler to list the output by using a `LIST ON` directive in the source file (and do not have a `-L` in the shell command line), or by setting the `+L` command line parameter, the assembled code is listed during pass two. Be aware that the `-L` parameter is not the same thing as a default; see the *Apple IIGS Programmer's Workshop Reference* for a discussion of the differences between the `-L` parameter and the shell command line defaults.

Each output line has four parts: a line number, the current relative address, the code that the Assembler generates (printed in hexadecimal as a sequence of up to four bytes, each byte separated by a space), and the source statement that generates the code. For example, if you have a line in your program like one of the lines used to set the data bank equal to the code bank in the earlier example, the assembled output line might look like this:

```
Line  Address  Code          Source Statement
0007  0001     AB            PLB
```

The first column is the four-digit decimal output line number. The line numbers start at 0001 and increment for each source line whether or not the output line is listed. This means that even if the listing is turned off for part of the assembly, you can determine how many lines the APW Assembler has processed. Lines generated by macros are not considered source lines, so they do not have line numbers, but they are identified by plus signs (+), as you can see in the fragment that follows.

```
Line  Address  Code              Source Statements
0003  0000                       LIST ON
0004  0000                       GEN ON
0005  0000                       MAIN  START
0006  0000   4B                  PHK
0007  0000   AB                  PLB
0008  0002                       WRITELN #'Hello World!'
      0002                +      ANOP
      0002   80 0D        +      BRA   ~B2
      0004   0C 48 65 6C +~A2    DC I1'1:&STR',C'Hello World!'
      0011   F4 00 00     +~B2    PEA   ~A2|-16
      0014   F4 04 00     +       PEA   ~A2
```

```
     0017   A2 0C 1A    +       LDX #$1A0C
     001A   22 00 00 E1 +~C2     JSL   $E10000
0009 001E   A9 00 00             LDA   #0
0010 0021   6B                   RTL
0011 0022                        END
```

Pausing in The Assembly

You can cause the assembly to pause at any time during pass two by pressing any character key Apple-period. When you press the next key, the assembly resumes. You can cause the assembler to pause and wait for a key press when it encounters an error by using the `+W` shell command line parameter described later in this chapter.

> **Note:** If there is no error, the assembler will pause only if a line or symbol table is being printed. It will not stop while pass headings containing the segment names are being printed.

Stopping The Assembly

You can stop the assembly by pressing Apple-period. The APW Assembler responds to this keyboard input by returning control to the editor. If the assembler happens to be listing output when you decide to stop, the listing is terminated.

If you have set one or more of the shell command-line parameters, their effect may override this description of what happens when you stop the assembler. See the descriptions of these parameters later in this chapter.

Assembler Error Messages

If the APW Assembler detects an error in the source statement and you are printing out the source, the error is printed on the next line. All error messages are text messages. The error messages are described in Appendix C.

`FILE` *filename source line number error message*

If the APW Assembler encounters any error and the +T shell command line parameter is set, the assembler returns control to the shell. The shell then calls the editor and displays the error message. Setting shell command-line parameters is described later in this chapter.

If the error is in a macro, the source code line that calls the macro is at the top of the screen. This allows you to identify the line containing the error, even if pass two has not started and no listing has been produced.

Printer Listings

Specifying the `+L` option with the shell command you used to assemble your program sends a listing to the screen. Specifying the `-L` option on the shell command line stops a

listing from being produced. You can redirect program assembly output from the screen to the printer using the redirection character >:

```
ASSEMBLE TEST >.PRINTER
```

You can also code the PRINTER directive with the ON option in your source file, in which case subsequent lines are sent to the printer. Printed listings are the same as listings to the screen, except that a form feed character is generated for the printer at the end of each page and at the end of the listing. Specifying the -L option on the shell command line overrides both the LIST and PRINTER directives in the source code.

For information on redirecting output and configuring the shell for the printer you are using, refer to the *Apple IIGS Programmer's Workshop Reference* manual.

APW Shell Commands

This section discusses the shell commands that you will use most often when working with the APW Assembler. With these commands, you can perform the following tasks:

- edit new and existing files
- assemble, link, and execute your program
- perform partial assemblies
- call the APW Linker
- call APW utility programs

Editing Files

There are three shell commands you need to edit a new or existing file. They are

ASM65816	Change the default language to 65816 assembly language
EDIT	Transfer control to the editor so that you can edit an existing file or open a new file
CHANGE	Change the language type of an existing file

When you edit an existing program, the editor is set to the language in which the file is written. When you open a new file, the editor is set to the default language, which is the last language used or set with a language command. If you need to set the language to 65816 assembly language before calling the editor, use the ASM65816 command, as you did in the example at the beginning of this chapter. If you need to change the language of an existing source file, use the CHANGE command.

Assembling, Linking, and Running Programs

There are several shell commands you can use when you are assembling, linking, and running your program. The following pointers will help you use these commands:

- You must separate the command from its parameters by one or more spaces.

- You can use the Right Arrow key to expand command names as described in the *Apple IIGS Programmer's Workshop Reference*. You can use the Up Arrow and Down Arrow keys to scroll through previously entered commands.
- There are no abbreviations for command names (except for aliases that you have added to the system).
- All commands and parameters (unless otherwise noted in the command description) can be entered in any combination of uppercase and lowercase characters.
- When a parameter in a command line conflicts with a source code command, the command-line parameter takes precedence. When you use neither a source code command nor a command-line parameter, the default parameter, if one exists, 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 *Apple IIGS Programmer's Workshop Reference.*

The APW Assembler recognizes the following APW Shell commands and parameters. Additional APW Shell commands and parameters are described in the *Apple IIGS Programmer's Workshop Reference* .

ASSEMBLE

```
ASSEMBLE  [+E|-E]  [+L|-L]  [+S|-S]  [+T|-T]  [+W|-W]  file1 [file2 ...]
          [KEEP=outfile] [NAMES=(seg1[ seg2[ ...]])]
```

This APW Shell command calls the APW Assembler to assemble one or more source files and produces object files. To save the object files, you must either include a `KEEP` directive in the source code, or use the `KEEP` parameter in this command, or use the `KeepName` shell variable. The `ASSEMBLE` command does not call the linker, so no executable load file is generated. The command parameters are described in the section "Command Parameters," later in this chapter.

ASML

```
ASML  [+E|-E]  [+L|-L]  [+S|-S]  [+T|-T]  [+W|-W]  file1 [file2...]
      [KEEP=outfile]  [NAMES=(seg1[ seg2[ ...]])]
```

This shell command assembles one or more source code files and produces object files. If the maximum error level found by the APW Assembler is equal to, or less than, the maximum error level allowed, the assembler completes the assembly and control returns to the shell. The shell then calls the linker to link the object files and any library files into a load file. The link fails if you do not have a `KEEP` directive in the source code and you do not use the `KEEP` parameter in this command, or the `KeepName` shell variable. If the error level is exceeded, the assembly terminates and the linker is not called. The maximum error level allowed is 0, unless you specify otherwise with the `MERR` directive. The command

parameters are described later in this chapter; the assembler directives are described in Chapter 4.

ASMLG

```
ASMLG [+E|-E] [+L|-L] [+S|-S] [+T|-T] [+W|-W] file1 [file2...]
      [KEEP=outfile] [NAMES=(seg1[ seg2[ ...]])]
```

The command ASMLG assembles one or more source files, links one or more object files with any library files, and runs the resulting load file. ASMLG functions exactly like ASML, except that when the object files are successfully linked into a load file, the load file is automatically executed.

LINK

```
LINK [+L|-L] [+S|-S] [+W|-W] objectfile1 objectfile2...[KEEP=outfile]
```

The LINK shell command calls the APW linker to link together object and library files to create an executable load file. Use this command if you have assembled your source code with the ASSEMBLE command. See the *Apple IIGS Programmer's Workshop Reference* for a complete description of the command and the rules for using the KEEP directive in source code, or on the shell command line, or with the KeepName variable.

> **Important:** The LINK command can be used only to process object and library files; do not try to process a LinkEd file with the LINK command.

RUN

```
RUN   [+E|-E] [+L|-L] [+S|-S] [+T|-T] [+W|-W] file1 [file2...]
      [KEEP=outfile] [NAMES=(seg1[ seg2[ ...]])]
```

The command RUN assembles, links, and executes source and library files. RUN functions exactly like ASMLG.

> **Note:** To automatically assemble and link two or more object files, use the ASSEMBLE and LINK commands in an Exec file as described in the *Apple IIGS Programmer's Workshop Reference*.

Command Parameters

In general, command-line parameters (those described here) override source code options where there is a conflict.

+E|-E — If you specify +E, the assembler calls the APW Editor when it encounters a fatal error. The editor displays the error message and the source file at the line where the error occurred. If you specify -E and a fatal error occurs in

the assembly, you are returned to the shell command line or the Exec file from which the command was executed.

`+L|-L` If you specify `+L`, the assembler generates a source listing, or if you are using `LINK`, a **link map** of the segments in the object files is produced (including the starting address, the length in hexadecimal of each segment, and the segment type). If you specify `-L`, the assembler does not produce either a listing or a link map. The default is `-L`. The `L` parameter overrides the `LIST` directive in the source file.

`+S|-S` If you specify `+S`, the APW Assembler produces an alphabetical listing of all local symbols following each `END` directive, and the linker, if it has been called, produces an alphabetical listing of all global references in the object file, called a **symbol table**. If you specify `-S`, these symbol tables are not produced. The default is `-S`. The `S` parameter in this command overrides the `SYMBOL` directive in the source file.

`+T|-T` If you select `+T`, the assembly terminates on any error. The `+T` parameter overrides the `MERR` directive in the source code. If you omit the `+T` parameter or select `-T`, only fatal errors cause immediate termination of the assembly. If you select both `+T` and `+E`, any error causes the assembler to terminate the assembly and call the APW Editor to display the error and message.

`+W|-W` If you select `+W`, the assembler stops on encountering an error and waits for a key press. Press Apple-Period (⌘-.) to terminate the assembly, or press any character key or the Space bar to continue. If you select `-W`, execution continues without pausing when an error is encountered.

file1 file2... The full or partial pathnames (including the filenames) of the source files to be assembled. You can also include the full pathnames or partial pathnames, minus filename extensions, of additional object files to be passed to the linker. You can include as many source, object, and library files as you wish, but be sure that at least one of the files is a source file, except when using the `LINK` command. The `LINK` command requires that *none* of the files be source files. Separate the filenames with spaces. See the *Apple IIGS Programmer's Workshop Reference* for a discussion of object files and the order in which they are passed to the linker.

Remember that any library files that you specify are searched in the order in which you list them on the command line. You should also note that if you list a library file before an object file, the library file is searched before the object file is linked. Finally, only the segments that are needed to resolve references that have not already been resolved are extracted from the library files.

`KEEP=`*outfile* Use this parameter to specify the pathname or partial pathname (including the filename) of the output file, or the executable load file if you are using `LINK`. Be sure that there are no spaces between `KEEP` and the equal sign (=).

See the *Apple IIGS Programmer's Workshop Reference* for a complete description of the KEEP parameter, what to do with partial assemblies, and how to compile source files in other languages.

NAMES=(*seg1 seg2* ...) This parameter causes the APW Assembler to perform a partial assembly. The operands *seg1 seg2* ... specify the names of the segments to be assembled. The segment names are separated by one or more spaces. There must not be any spaces between NAMES and the equal sign (=). The APW Linker automatically selects the latest version of each segment when the program is linked.

You assign names to object segments with START, PRIVATE, PRIVDATA, or DATA directives. The object file created when you use the NAMES parameter contains only the object segments that you list in this parameter. When you link a program, the linker scans all the files whose filenames are identical except for their extensions, and takes the latest version of each segment. Therefore, you must use the same output filename for every partial assembly of a program. See the *Apple IIGS Programmer's Workshop Reference* for examples and information on using the NAMES parameter with multiple source filenames on the shell command line

Note: Segment names are case sensitive

objectfile1 objectfile2... The full pathname, including filename, minus filename extensions, of the object files to be linked. You can link several object files into one load file with a single LINK or ASML command. You can also search several library files. They are linked in the order in which they are listed. Use the full or partial pathnames, minus filename extensions, of all the object files to be included. Separate the pathnames with spaces. See the *Apple IIGS Programmer's Workshop Reference* for a description of full and partial pathnames and filename extensions.

You can set the KeepName shell variable instead of using the KEEP assembler directive, or the KEEP parameter on the command line. If you use KeepName, the shell uses the value you set to name the output files. This name has precedence over any value set with the KEEP assembler directive. KeepName is fully described in the *Apple IIGS Programmer's Workshop Reference*.

If you use neither the KEEP parameter, nor the KEEP assembler directive, nor the KeepName shell variable, then the object modules are not saved at all. In this case, the link cannot be performed, because there is no object module to link.

Appending Files

When the assembler recognizes an APPEND directive in an assembly source file, it returns control to the shell which opens the file named in the APPEND operand, bringing in a language translator, if needed. If the current file and the new file are both in assembly language, the effect is the same as if the two files were concatenated into a single file. This means that flags that are set in the first file remain set in the second file and global EQU directives set in the first file keep the symbols available to the second file. See the *Apple IIGS Programmer's Reference* for information about appending assembly-language source files to source files in other languages.

Partial Assemblies

If you are using partial assemblies, note the availability of the shell command CRUNCH. CRUNCH combines all the object modules created by partial assemblies into a single file.

Assembler directives that are global in scope are resolved whether or not they are in one of the subroutines assembled. These directives are

ABSADDR	Allow absolute addresses
APPEND	Append a source file
CASE	Specify case sensitivity
COPY	Copy a source file
DIRECT	Promote direct page locations to absolute addresses
ERR	Print errors
EXPAND	Expand DC statements
GEN	Generate macro expansions
GEQU	Define a global symbolic constant
IEEE	Enable IEEE format numbers
INSTIME	Show instruction times
KEEP	Keep an output file
LONGA	Select accumulator size
LONGI	Select index register size
LIST	List output
MCOPY	Copy a macro file
MDROP	Drop a macro file
MERR	Maximum error level
MLOAD	Load a macro file
MSB	Set or clear most significant bit
NUMSEX	Set byte order in floating point numbers
OBJCASE	Specify case sensitivity in object files
PRINTER	Send output to the printer
RENAME	Rename an operation code
SETCOM	Set comment column number
SYMBOL	Print symbol tables
65816	Enable 65816 operation codes
65C02	Enable 65C02 operation codes
TITLE	Print header
TRACE	Observe assembler processing

The operands of these directives cannot contain labels unless they appear inside a program segment, and the segment that they appear in is assembled. If you do not follow these rules, an invalid operand error will result.

Listings and error messages are sent to the screen unless you include a `PRINTER` directive with the `ON` option in your source file, redirect output to a disk file or the printer in the command line, or use command-line parameters to suppress the output. No listing is the default. Error messages cannot be suppressed.

The APW Linker

The APW Linker takes object files created by the APW Assembler and library files 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 dynamic and static data segments, dynamic and static code segments, and library files among other things.

The linker is called automatically by the shell commands `ASML`, `ASMLG`, and `RUN`. The linker can also be called directly by using the `LINK` command. A **LinkEd** command file can be used to create dynamic segments, search specified library files, link object files in a given order, extract specific segments to link, and set load addresses for nonrelocatable code. LinkEd commands can be appended to the last file of the source code, or can be assembled and executed separately using the shell commands `ASSEMBLE` or `ALINK`. The linker and its uses are described in detail in the *Apple IIGS Programmer's Workshop Reference* manual.

Making Library Files

The MakeLib utility creates a library file (ProDOS 16 filetype $B2) from one or more object files. Each library file consists of one or more segments, and each segment can contain as many subroutines as you care to put in it. 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 would be able to 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 use several library files, on the other hand, the order in which the files are searched *is* 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.

Note: The linker does not recognize *file*.ROOT and *file*.A as related to each other in a library. They are treated as separate and equal files rather than two files in a sequence. This means that private labels cannot be accessed because they are not available outside of the object file in which they are defined.

The MakeLib utility is invoked by the APW Shell command MAKELIB. The syntax for this command is

MAKELIB [-F] [-D] *libfile* [+ *objectfile...*] [- *objectfile...*] [^ *objectfile...*]

You can use this command to either create or modify a library file. The parameters for this command are as follows:

-F — If you specify -F, a list of filenames included in *libfile* is produced. If you omit this option, no filename list is produced.

-D — If you specify -D, the dictionary of symbols in the library is listed. Each of the listed symbols is a global symbol occurring in the library file. If you omit this option, no dictionary is produced.

libfile — The full pathname or partial pathname (including the filename) of the library file to be created, read, or modified.

+ *objectfile* — The full pathname or partial pathname (including the filename) of an object file to be added to the library. You may add as many object files as you wish. Separate object filenames with spaces.

- *objectfile* — The filename of a component file to be removed from the library. This parameter is a filename only, not a pathname. You may remove as many files as you wish. Separate the filenames with spaces.

^ *objectfile* — The full pathname or partial pathname (including the filename) of a component file to be removed from the library and written out as an object file. If you include a prefix in this pathname, the object file is written to that prefix. You may specify as many files as you wish to be written out as object files. Separate the filenames with spaces.

Note: You must specify at least one object file or option, otherwise, the message `No action requested` appears on the screen.

To create a library file using the APW Assembler, follow this procedure:

1. Write one or more source files in which each library subroutine is a separate segment. The first routine should be a dummy segment (the segment consists of a START directive, followed by an END directive).
2. Assemble the segments, specifying a unique name for each with, for example, the KEEP parameter in the ASSEMBLE command. If you have a multisegment program (a dummy segment followed by all of the rest of your library routines), it is saved as two object files, the first file has the extension .ROOT appended to its name and the

second file (containing the library routines) has the extension .A appended to its name.

3. Run the MakeLib utility, specifying each object file to be included in the library file. For example, if you assembled two files, you would create the object files LIBOBJ1.ROOT, LIBOBJ1.A, LIBOBJ2.ROOT, and LIBOBJ2.A. Because LIBOBJ1.ROOT and LIBOBJ2.ROOT contain dummy segments, you do not want to include them in your library. If your library file is named LIBFILE, then your command line will be

```
MAKELIB LIBFILE +LIBOBJ1.A +LIBOBJ2.A
```

4. Place the new library file in the library prefix (prefix 2). You can accomplish this in step 3 by changing your command line to

```
MAKELIB 2/LIBFILE +LIBOBJ1.A +LIBOBJ2.A
```

You could also put your new library file in the subdirectory by using the MOVE command after the file is created.

The Apple IIGS Debugger and DumpOBJ

The Apple IIGS Debugger and the DumpOBJ utility program are both useful tools for debugging assembly-language programs.

With the Apple IIGS Debugger, you can trace the execution of your program, stepping through the code one instruction at a time or executing at full speed. In either case, the debugger will display the contents of the registers, the stack, the direct page, and 384 bytes of RAM at any breakpoint you specify. The Apple IIGS Debugger displays in 80-column mode only, but allows you to switch between its own display and the display of the program being tested. The Apple IIGS Debugger is described in the *Apple IIGS Debugger Reference*.

The DumpOBJ utility program is useful for debugging assembly-language code and for whenever you want to see the contents of an object file. DumpOBJ writes the contents of an object file to standard output (usually the screen). You can list either in object module format (OMF), 65816 machine-language disassembly, or hexadecimal. You also have the choice of listing only segment headers, or only the names of segments and their types, or only operation codes and operands, or a variety of other options. The complete description of DumpOBJ is found in the *Apple IIGS Programmer's Workshop Reference*.

Part II

Language Reference

Chapter 3

APW Assembly Statements

Each line of an assembly-language source file is either an assembly statement or a comment. The APW Assembler allows only one assembly statement per source line. Assembly statements can consist of any of the following:

- a 65816 instruction (this includes 6502 and 65C02 instructions)
- an APW Assembler directive
- an APW macro

All of the APW Assembler statement types have the same syntax.

Assembly Statement Formats

An assembly statement can consist of up to four fields:

[label] OPERATION [operand] [comment]

The operation field is usually the only one required. Any time two or more fields appear in one assembly statement, they must be separated by at least one space.

An APW Assembler source file line can be up to 255 characters long. The APW Editor, however, allows only 80-character lines. If you are using an 80-column printer, source file lines longer than 57 columns will cause printed assembler output to wrap around to the next line. To prevent this from happening, you can reset the APW Editor's end-of-line marker from its default of 80 columns to 57 columns, or change the font on your printer. See the *Apple IIGS Programmer's Workshop Reference* manual for more information on the APW Editor.

Labels

Labels can represent either the address of data or the address of an instruction. Labels are usually optional. Exceptions are the `DATA`, `ENTRY`, `EQU`, `GEQU`, `PRIVATE`, `PRIVDATA`, and `START` directives, where labels are required. If present, the label must begin in column 1 with an alphabetic character, an underscore, (_), or a tilde (~). Labels may be 1 to 255 characters in length and consist of letters, digits, underscores (_), and tildes (~); they may not contain imbedded spaces.

Note: Labels beginning with a tilde character (~) are reserved for system use.

The APW Assembler is case insensitive by default, so that the labels, `Console` and `CONSOLE` mean the same thing. The underscore (_) is significant in labels. This means that `THISLABEL` and `THIS_LABEL` are different.

> **NOTE:** It is best not to use `A` as a label, because it can cause confusion between absolute addressing using the label `A` and accumulator addressing.

Label Scope

A label may be private, global, or local in scope. A private label is not available outside of the object file in which it is defined. A global label can be referenced from any segment in the program, while a local label has validity only within the segment where it is defined. You can define a local label with the same name as a global label. The APW Assembler chooses the local label in preference to the global label. A particular label can be defined in each code segment in the program, but only once per segment. Labels within data segments can be defined once per program. They have local scope, but will be flagged as errors if duplicates appear within other data segments.

The APW Assembler directives that define global labels are `START`, `DATA`, `GEQU`, `ENTRY`, `PRIVATE`, and `PRIVDATA`.

A label defined by the `GEQU` directive is available at assembly time to source code that follows it, while the actual values of all other global labels are available only at link time. Consequently, you should always use a `GEQU` directive to define a direct page or long address label that will be used in more than one segment. That way, the assembler can automatically determine which addressing mode is most appropriate.

The following example shows how both local and global labels can be used. This code would produce an error in `SEG1` because the label `LAB1` is not defined globally. It is defined locally only for `SEG2`. It is legal, however,for both segments to use the local label `LAB2`, which has a different value in each case.

```
SEG1    START
LAB2    LDA     LAB1
        END

SEG2    START
LAB2    LDX     LAB2
LAB1    LDY     LAB1
        END
```

Case Sensitivity in Labels

You can make a label case sensitive to the APW Assembler by specifying the `ON` option with the `CASE` directive in your source code. `CASE OFF` reverses the effect. The directive `OBJCASE ON` causes labels sent to the object file to be case sensitive, whether or not they are treated as case sensitive inside the assembler. Specifying the option `OFF` with the directive `OBJCASE` makes exported labels case insensitive, whether or not they are treated as case sensitive inside the assembler. The default is `OBJCASE OFF`. Setting

CASE also sets OBJCASE, so if the exported behavior is to be different from the local behavior, you must specify the OBJCASE directive last.

Attributes and Labels

Attributes can tell you whether or not a label has been defined, whether or not a parameter has been passed, or the kind of statement that generated the label. Attributes may be thought of as functions that return information about labels. Attributes are resolved during expression evaluation.

The form of an attribute used with a label is

X: *Label*

where the X stands for C, L, T or S. These attributes are defined as follows:

C	Count attribute
L	Length attribute
T	Type attribute
S	Settings attribute (described in Chapter 4)

Count Attribute

The count attribute is used to tell whether or not a label has been defined. The count attribute of an undefined label is zero. The count attribute of a defined label is one. Several examples of the count attribute are given in Chapter 6.

Length Attribute

The length attribute of a label is the number of bytes created by the line where the label was defined. This makes counting characters very easy as shown by the example of how to count characters in a string:

```
STRING   DC          C'Hello, World'
         DC          I1'L:STRING'
```

The length of the string in the first line will be used as the value for the second DC directive. The second DC directive will contain a value equal to a decimal 12: 10 letters, plus 1 comma, plus 1 space.

Note: The length attribute returns a valid value only if the length is in the range 0-255.

Type Attribute

The type attribute is used to determine the kind of statement that generated the label. The character that is returned for each type is indicated in the table below.

Character	Meaning
A	Address-type DC directive
B	Boolean-type DC directive
C	Character-type DC directive
D	Double-precision floating-point type DC directive
E	Extended floating-point DC directive
F	Floating-point type DC directive
G	EQU or GEQU directive
H	Hexadecimal-type DC directive
I	Integer-type DC directive
K	Reference-address type DC directive
L	Soft reference type DC directive
M	Instruction
N	Other assembler directives
O	ORG directive
P	ALIGN directive
S	DS directive

If a DC directive contains more than one type of variable, the first type in the line determines the type attribute. For example, the LDA instruction in this example is not assembled because its type attribute is C (character) rather than A (address).

```
LAB   DC    A'ABCD', H'ABCD'
      AIF   T:LAB='A',.A
      LDA   #10
.A
```

Operation Code

The operation code is required in all assembly statements. It can contain either an assembly-language instruction to the 65C816 processor, a directive to the APW Assembler, or a macro call.

Normally, the operation code starts in column ten. The editor has a tab stop set to this column for your convenience. If there is no label, however, the operation code can start in any column greater than two.

65816 Instruction Set

The operation codes for the 65816 instruction set used by the Apple IIGS are three-character alphabetic strings. While this manual assumes that you are familiar with the 65816 instruction set, it is summarized here for your reference. Appendix A contains a list of the instructions, their addressing modes, and the number of bytes required by each.

The Apple IIGS can also operate in 6502 emulation mode. In this mode, the Apple IIGS emulates the 6502 processor exactly, including the configuration of the registers, stack location, and instruction timing.

ADC Add With Carry adds the data located at the address specified by the operand to the contents of the accumulator; adds one to the result if the carry flag is set, and stores the final result in the accumulator.

AND Logically ANDs accumulator contents with the contents of memory specified by the operand. Stores the result in the accumulator.

ASL Shift Memory Or Accumulator Left shifts the contents of the location specified by the operand left one bit. The arithmetic result of the operation is an unsigned multiplication by two.

BCC Branch If Carry Clear tests the carry flag in the P status register. If the flag is clear, a branch is taken to the displacement specified in the operand field; if the flag is set, the instruction immediately following BCC is executed.

BCS Branch If Carry Set tests the carry flag in the P status register. If the flag is set, a branch is taken to the displacement specified in the operand field; if it is clear, the instruction immediately following the BCS instruction is executed.

BEQ Branch If Equal tests the zero flag in the P status register. If it is set, the last value tested was zero and a branch is taken to the displacement specified in the operand field; if it is clear, the instruction immediately following BEQ is executed.

BIT Test Memory Bits Against Accumulator sets the P status register flags based on the result of two different operations. First, it sets or clears the n flag to reflect the high-bit value of the data at the location specified by the operand, and sets or clears the v flag to reflect the contents of the next-to-highest bit of the data. Second, it logically ANDs the data with the contents of the accumulator and sets the z flag if the result is zero; clears it if the result is not zero. The contents of the accumulator are unaffected.

BMI Branch If Minus tests the negative flag in the P status register. If it is set, the high bit of the value which most recently affected the n flag was set, and a branch is taken. If the negative flag is not set, the instruction following BMI is executed.

BNE Branch If Not Equal tests the zero flag in the P status register. It is clear if the value just tested was not zero, and a branch is taken to the displacement specified in the operand field. If the zero flag is set, the value tested was zero, and the instruction following BNE is executed.

BPL Branch If Plus tests the negative flag in the P status register. It is clear if the last value that affected the negative flag had its high bit clear (a two's complement positive number), the branch is taken. If the flag is set, the high bit of the last value was set (a two's complement negative number), the instruction immediately following BPL is executed.

BRA Branch Always branches with no tests. BRA, is in effect, a two byte unconditional JMP that is relocatable. Because the BRA instruction uses displacements from the program counter, its branch targets are limited to plus or minus 128 bytes from the first byte following the BRA instruction.

BRK Software Break forces a software interrupt. BRK is not affected by the interrupt disable flag.

BRL Branch Always Long is a three-byte relocatable instruction; the two bytes immediately following the operation code form a sixteen-bit signed displacement from the program counter. Once the branch address is calculated, the result is loaded into the program counter and control is transferred to the new location.

BVC Branch If Overflow Clear tests the overflow flag in the P status register. If it is clear, a branch is taken; if it is set, the instruction immediately following BVC is executed.

BVS Branch If Overflow Set tests the overflow flag in the P status register. If the flag is set, a branch is taken; if it is clear, the instruction following BVS is executed.

CLC Clear Carry Flag clears the carry flag in the status register.

CLD Clear Decimal Mode Flag clears the decimal mode flag in the status register.

CLI Clear Interrupt Disable Flag clears the interrupt flag in the status register.

CLV Clear Overflow Flag clears the overflow flag in the status register.

CMP Compares the contents of the accumulator with the data specified by the operand. Sets the carry, zero, and negative flags based on the result. The comparison is of unsigned binary values only (except for signed comparison for equality).

COP Co-Processor Enable causes a software interrupt. The COP instruction must be followed by a signature byte.

CPX Compare X Register compares the X register with the data specified by the operand. Sets the carry, zero, and negative flags based on the result. The comparison is of unsigned values only (except for signed comparison for equality).

CPY Compare Y Register compares the Y register with data specified by the operand. Sets the carry, zero, and negative flags based on the result. The comparison is of unsigned values only (except for signed comparison for equality).

DEC Decrement subtracts one from the value at the location specified by the operand.

DEX Decrement X subtracts one from the value in the X register.

DEY Decrement Y subtracts one from the value in the Y register.

EOR Exclusive-ORs accumulator with the data specified by the operand. Stores the result in the accumulator.

INC Increment adds one to the contents of the location specified by the operand.

INX Increment X adds one to the value in the X register.

INY Increment Y adds one to the value in the Y register.

JML Jump Long transfers control to a 24-bit (long) address specified by the operand field.

JMP Jump transfers control to the address specified by the operand field.

JSR Jump To Subroutine transfers control to the subroutine at the address within the current program bank specified by the operand, after first pushing the current program counter onto the stack, as a return address.

JSL Jump To Subroutine Long transfers control to the subroutine at the 24-bit address that is the operand, after first pushing a 24-bit (long) return address onto the stack. This return address is the address of the last instruction byte (the fourth instruction byte, or the third operand byte), not the address of the next instruction.

LDA Load Accumulator loads the accumulator with the data specified by the operand.

LDX Load X Register loads the X register with the data specified by the operand.

LDY Load Y Register loads the Y register with the data specified by the operand.

LSR Logical Shift Right logically shifts the contents specified by the operand right one bit.

MVN Move Next copies a block of memory to a new location. The beginning address of the source for the move is in the X register. The beginning address of the destination is in the Y register, and the length of the move is in the C (double accumulator) register.

Note: The MVN instruction always copies one more byte than is specified in the accumulator.

MVP Move Previous copies a block of memory to a new location. The ending address of the source for the move is in the X register. The ending address of the destination is in the Y register, and the length of the move is in the C (double accumulator) register.

Note: The MVP instruction always copies one more byte than is specified in the accumulator.

NOP No Operation increments the program counter once to point to the next instruction.

ORA OR Accumulator ORs the accumulator with the date specified by the operand. Each bit in the accumulator is ORed with the corresponding bit in memory. The result is stored in the accumulator.

PEA Push Effective Absolute Address pushes the two bytes in the operand onto the stack. This operation always pushes sixteen bits of data (typically an absolute address), regardless of how the m and x mode select flags are set.

PEI Push Effective Indirect Address pushes two bytes to the stack. The location of the two bytes is determined by adding the contents of the data byte immediately following the operation code to the direct register (D). PEI implies that the sixteen-bit data pushed is an address, although it can be any sixteen bits of data. This operation always pushes sixteen bits of data, irrespective of the settings of the m and x mode select flags.

PER Push Effective Relative pushes the effective relative indirect address onto the stack. This instruction adds the current value of the program counter to the sixteen-bit signed displacement in the operand, and pushes the result onto the stack. This operation always pushes sixteen bits of data, no matter how the m and x mode select flags are set.

PHA Push Accumulator pushes the contents of the accumulator onto the stack. This instruction pushes 8-bits if m is equal to 1, or 16-bits if m is set to 0.

PHB Push Data Bank Register pushes the contents of the data bank register onto the stack. This instruction always pushes 8-bits or one byte.

PHD Push Direct Page Register pushes the contents of the direct page register onto the stack. This instruction always pushes 16-bits or two bytes.

PHK Push Program Bank Register pushes the contents of the program bank register onto the stack. This instruction always pushes 8-bits or one byte.

PHP Push P register pushes the contents of the processor status register onto the stack. This instruction always pushes 8-bits or one byte.

PHX Push X Register pushes the contents of the X register onto the stack. This instruction pushes 8-bits if m is equal to 1, or 16-bits if m is set to 0.

PHY Push Y Register pushes the contents of the Y register onto the stack. This instruction pushes 8-bits if m is equal to 1, or 16-bits if m is set to 0.

PLA Pull Accumulator pulls the value on the top of the stack into the accumulator. This instruction pulls 8-bits if m is equal to 1, or 16-bits if m is set to 0.

PLB Pull Bank pulls the eight-bit value on top of the stack into the data bank register B and switches the data bank to the new value. All instructions that reference data specifying sixteen-bit addresses get their bank address from the value in the data bank register. This is the only instruction that can modify the data bank register.

PLD Pull Direct pulls the sixteen-bit value on top of the stack into the direct page register D. Switchs the direct page to the new value.

PLP Pull Processor pulls the eight-bit value on top of the stack into the processor status register P.

PLX Pull X Register pulls the value on the top of the stack into the X register. This instruction pulls 8-bits if m is equal to 1, or 16-bits if m is set to 0.

PLY Pull Y Register pulls the value on the top of the stack into the Y register. This instruction pulls 8-bits if m is equal to 1, or 16-bits if m is set to 0.

REP Reset Status Bits scans bits set to one in the operand byte, and resets the corresponding bits in the status register to zero. Zeros in the operand byte cause no change to their corresponding status register bits.

ROL Rotate Memory Or Accumulator Left moves the contents of the location specified by the operand left one bit. The leftmost bit is transferred into the carry flag; the rightmost bit takes the value in the carry flag. When m is equal to 0, the leftmost bit is bit 7; when m is set to 1, the leftmost bit is bit 15.

ROR Rotate Memory Or Accumulator Right moves the contents of the location specified by the operand right one bit. The leftmost bit takes the value in the carry flag; the rightmost bit, bit zero, is transferred into the carry flag. When m is equal to 0, the leftmost bit is bit 7; when m is set to 1, the leftmost bit is bit 15.

RTI Return From Interrupt pulls the status register and the program counter from the stack. If the processor is set to native mode (e=0), the program bank register is also pulled from the stack.

RTL Return From Subroutine Long pulls the program counter (first incrementing the sixteen-bit stack value by one), then the program bank register from the stack.

RTS Return From Subroutine pulls the program counter from the stack, after incrementing it by one.

SBC Subtract With Borrow From Accumulator subtracts the data located at the address specified by the operand from the contents of the accumulator (subtracting one more if the carry flag is clear), and stores the result in the accumulator.

SEC Set Carry sets the carry flag in the status register.

SED Set Decimal sets the decimal mode flag in the status register.

SEI Set Interrupt sets the interrupt disable flag in the status register.

SEP Set Status Bits sets each bit in the status register to the corresponding bit in the operand byte. Zeros in the operand byte cause no change to their corresponding status register bits.

STA Store Accumulator stores either the 8 or 16-bits of the accumulator at the address specified by the operand.

STP Stop Processor shuts down the processor until a reset occurs.

STX Store X stores the contents of the X register at the memory address specified by the operand. This instruction stores either 8 or 16-bits, depending upon the size of the X register.

STY Store Y stores the contents of theY register at the memory address specified by the operand. This instruction stores either 8 or 16-bits, depending upon the size of the Y register.

STZ Store Zero stores zero in the memory location specified by the operand.

TAX Transfer Accumulator To X transfers the contents of the accumulator to the x register. If the registers are different sizes, the size of the X register determines the number of bits transferred.

TAY Transfer Accumulator To Y transfers the contents of the accumulator to the Y register. If the registers are different in size, the size of the Y register determines the number of bits transferred.

TCD Transfer C To D transfers the contents of the 16-bit accumulator to the direct page register, regardless of how the accumulator memory mode flag is set.

TCS Transfer C To S transfers the contents of the 16-bit accumulator to stack pointer.

TDC Transfer D To C transfers the contents of the direct page register to the 16-bit accumulator. The transfer is always sixteen bits, regardless of the setting of the accumulator memory mode flag.

TRB Test And Reset Bits ANDs to clear any bit in the memory operand that is set in the A accumulator.

TSB Test And Set Bits sets any bit in the memory operand that is set in the A accumulator.

TSC Transfer S To C moves the value in the sixteen-bit stack pointer S to the sixteen-bit accumulator C.

TSX Transfer S To X transfers the contents of the stack pointer S to the X register.

TXA Transfer X To A transfers the contents of the X register to the accumulator. If the registers are different sizes, the number of bits transferred is determined by the accumulator.

TXS Transfer X To S transfers the value in the X register to the stack pointer

TXY Transfer X To Y transfers the value in the X register to the Y register.

`TYA` Transfer Y To A transfers the value in the Y register to the accumulator. If the registers are different sizes, the number of bits transferred is determined by the size of the accumulator.

`TYX` Transfer Y To X transfers the value in the Y register to the X register.

`WAI` Wait For Interrupt pulls the RDY pin low. Power consumption is reduced and RDY remains low until an external hardware interrupt (NMI, IRQ, ABORT, or RESET) is received.

`WDM` Reserved for future expansion.

`XBA` Exchange B And A exchanges the contents of the low-order and high-order bytes of the 16-bit accumulator, where B is the high-order byte and A is the low-order byte.

Important: All flags are set according to the value of the low byte or accumulator after the exchange

`XCE` Exchange Carry And Emulation Bits shifts the processor between 6502 emulation mode and 16-bit native mode. The operation exchanges the carry bit in the P register with the value of the e bit. If e = 1, the processor executes in emulation mode; if e = 0, it executes in native mode.

The APW assembler allows substitutions for the following standard operation codes:

Standard	Also Allowed
BCC	BLT
BCS	BGE
CMP	CPA
DEC A	DEA
INC A	INA

APW Assembler Directives

The APW Assembler directives, which are listed in Chapter 4, tell the APW Assembler to do such things as define program data, reserve space in memory, or send output to a printer.

APW Assembler Macro Calls

Macro calls tell the APW Assembler to insert a group of instructions into the code. The assembler directives that you need to use the predefined macros in the macro libraries are described in Chapter 5. The directives that you need to write your own macro calls are given in Chapter 6.

Operands

An operand provides information that the operation code uses to perform its function. In the case of 65816 instructions (including 6502 and 65C02 instructions), an operand is generally an expression that resolves to an address. There must be at least one space between the operation code and the operand. The operand normally starts in column sixteen; the editor provides a tab stop there for your convenience. The operand may not start in any column at or beyond that specified by the SETCOM directive (normally 41); otherwise, the operand will be considerd a comment.

Instruction Operand Formats

When the operation code is a 65816 instruction, the operand that follows it consists of an expression, and, if the addressing mode requires it, an addressing mode indicator. Together they tell the assembler which addressing mode you want to use with the instruction.

The choice of addressing mode determines where an instruction gets its data. If no addressing mode indicator is present, the default addressing mode for the instruction is determined based on the value of the expression.

For example, the difference between two-byte indirect addresses and three-byte indirect addresses is represented by the presence of the following indicators:

operands enclosed by ( ) are 2-byte indirect addresses

operands enclosed by [ ] are 3-byte (long) indirect addresses.

Immediate addressing operands are indicated by a number sign (#) prefix. If the operand has a # operator, or #<, it means to use the least significant byte, or bytes, from the expression for the address when forming immediate operands. For example:

```
LDA     #$123456
```

causes $56 to be generated as the immediate operand in eight-bit mode and $3456 to be generated in 16-bit mode.

The operators #> or / mean shift the expression value right by one byte (divide by 256). For example:

```
LDA     #>$123456
```

causes $34 to be generated in eight-bit-mode and $1234 to be generated in sixteen-bit-mode.

The ^ operator means to shift the expression right by two bytes (divide by 65536). For example

```
LDA     #^$123456
```

causes $12 to be generated in 8-bit mode and and $0012 to be generated in 16-bit mode.

```
LDA     #^$12345678
```

causes `$34` to be generated in 8-bit mode and and `$1234` to be generated in 16-bit mode.

Finally, instructions that use registers, such as the index registers or the stack pointer, generally include the register letter (`X`, `Y`, or `S`) in the operand. The register letter is separated by a comma from the operand value and positioned relative to other address mode indicators such as parentheses and brackets to signify the kind of effective address generation desired.

Table 3-1 shows the legal operand syntax for the 65816 addressing modes. It uses the symbols *expr* to refer to any expression, and *dp, abs,* and *long,* to refer to constant expressions that resolve to one, two, or three bytes respectively.

Table 3-1. Operand Symbol Types

This table summarizes the abbreviations used for legal values in the addressing mode syntax examples:

expr	An expression without regard to magnitude.
dp	An expression that resolves to a number in the range 0-255 for the direct page or stack offset. Addresses of other lengths can be forced to one-byte with the less than (<) prefix.
abs	An expression that resolves to a two-byte address. A two-byte address can be forced (regardless of the actual length of the value) by preceding the expression with a vertical bar (|).
long	An expression that resolves to a three-byte address. Addresses of other lengths can be forced to three-bytes with the greater than (>) prefix

The actual value, or if that is not known, the presumed range of an expression is used in addition to other address mode indicators to determine the appropriate operand size so that the operation code with the correct addressing mode for the instruction will be generated. When the < character is prefixed to an expression, it forces evaluation as an 8-bit (one byte) value (less than $100). When a > is prefixed to an expression, it forces evaluation as a 24-bit value (greater than $FFFF). When > is used, for example, as a prefix to a value such as $36 (for example, `LDA >$36`), the absolute long addressing mode is used rather than the direct page mode that would normally be used with an operand value of $36. When the | operator is prefixed to an expression, it forces evaluation as a 16-bit value (greater than $FF but less than $10000). If the assembler does not know the actual value of a relocatable or external expression, it assumes a 16-bit value. If a modifier is used and the value is greater than will fit in the size indicated, any extra bytes are ignored. If the size is greater than required to represent the value, additional zero-bytes are added.

The `JMP` indirect absolute instruction needs a sixteen-bit address as the target for the jump:

```
JMP ($2000)
```

This syntax will result in an error message if an expression that resolves to one byte is used instead:

```
JMP (|$36)
```

The use of the vertical bar before the expression forces the extension to a 16-bit operand size (no matter what value is given within the parentheses):

```
JMP (|$36)
```

resolves to `$0036`. Long addressing is forced in the same way with the > operator:

```
LDA >$2300
```

This generates a three-byte operand which *always* loads at `$2300` in bank zero, regardless of the data bank value. The same instruction without the > operator

```
LDA $2300
```

generates a 2-byte operand that loads from offset `$2300` in the current data bank.

The < operator is used to force evaluation as a one-byte (8-bit) value as with

```
LDA <$2034
```

This address mode syntax causes the operand to evaluate to the one byte value `($34)`; therefore, the direct page addressing form of the `LDA` instruction is generated by the assembler. When the instruction is executed, data is loaded from the direct page offset of `$34`. The high byte (`$20`) is ignored by the assembler.

An example of the legal format for each of the addressing modes is shown in Table 3-2

Table 3-2. The 65816 Addressing Mode Operand Syntax

Addressing Mode	Operand Format Example
Absolute	`LDA` *abs* or `LDA` `|`*expr*
Absolute Long	`LDA` *long* or `LDA` `>`*expr*
Absolute Indexed	`LDA` *abs*`,X` or `LDA` `|`*expr*`,X` `LDA` *abs*`,Y` or `LDA` `|`*expr*`,Y`
Absolute Long Indexed	`LDA` *long*`,X` or `LDA` `>`*expr*`,X`
Absolute Indexed Indirect	`JMP (`*abs*`,X)` or `JMP (|`*expr*`,X)`
Absolute Indirect	`JMP (`*abs*`)` or `JMP (|`*expr*`)`

Absolute Indirect Long	`JMP [abs]` or `JMP [lexpr]`
Accumulator	`ASL A`
Block Move	`MVN long,long`
Direct Page	`LDA dp` or `LDA <expr`
Direct Page Indexed	`LDA dp,X` or `LDA <expr X` `LDA dp,Y` or `LDA <expr,Y`
Direct Page Indirect	`LDA (dp)` or `LDA (expr)`
Direct Page Indirect Long	`LDA [dp]` or `LDA [<expr]`
Direct Page Indirect Indexed	`LDA (dp),Y` or `LDA (<expr),Y`
Direct Page Indirect Indexed Long	`LDA [dp],Y` or `LDA [<expr],Y`
Direct Page Indexed Indirect	`LDA (dp,X)` or `LDA (<expr,X)`
Immediate	`LDA #expr`
Implied	`INY`
Program Counter Relative	`BRA expr`
Program Counter Relative Long	`BRL expr`
Stack (Absolute)	`PEA abs` or `PEA #expr`
Stack (Direct Page Indirect)	`PEI dp` or `PEI (dp)` or `PEI (<expr)`
Stack (Program Counter Relative)	`PER expr` or `PER abs`
Stack Relative	`LDA dp,S` or `LDA <expr,S`
Stack Relative Indirect Indexed	`LDA (dp,S),Y` or `LDA (<expr,S),Y`

Expressions

An expression is a logical or mathematical formula that resolves to a number. The expression can contain both labels and constants. In general, expressions resolve to integers in the range -2147483648 to 2147483647. If the expression is a logical operation, its result is always 0 or 1, corresponding to false or true. If an arithmetic value is used in an assembler directive that expects a Boolean result, 0 is treated as false, and any other value is treated as true.

An expression can be either simple or compound. A compound expression consists of two simple expressions separated by one of the logical operators shown in Figure 3-1.

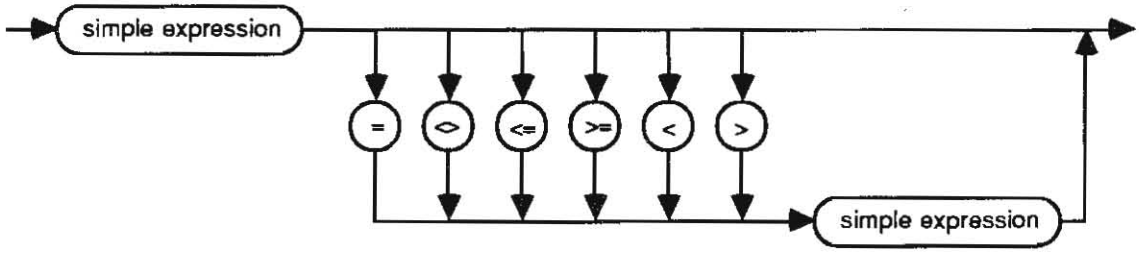


Figure 3-1. Syntax of a Compound Expression

Here are some examples of compound expressions that follow these rules:

Expression	Result
`2<4`	1
`2+1<>$FFF`	1
`LOOPCOUNT=LOOPCOUNT+1`	0

The syntax for a simple expression is shown in Figure 3-2. It consists of

1) an optional leading sign,

2) a term, and optionally,

3) a +, -, .OR., or .EOR. followed by another term.

Logical comparisons have the lowest priority.

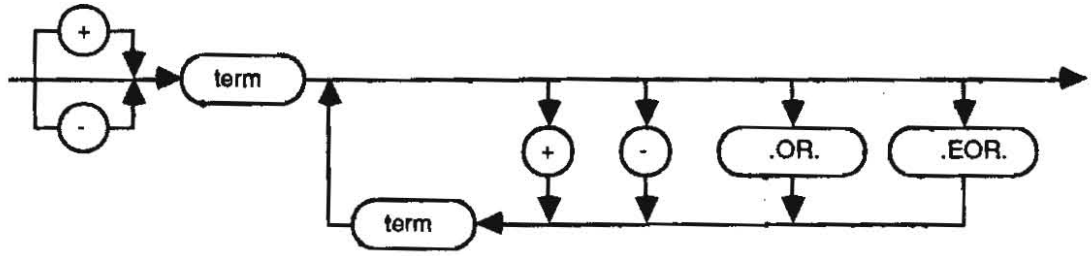


Figure 3-2. Syntax of a Simple Expression

Here are some examples of a simple expression made up of a term, an operator, and another term:

Expression	Result
`5+6`	`11`
`-3+2`	`-1`
`1.OR.0`	`1`
`3/4+6*2`	`12`

A term is a factor, optionally followed by one of the operators *, /, .AND., or | (the bit shift operator) and another factor. Figure 3-4 shows the syntax of a term. .AND. is a logical operator, asking if the terms on either side are true. If both are true, so is the result, otherwise the result is false. The vertical bar (or, optionally, !) is a bit shift operator. The first operand is shifted the number of bits specified by the right operand, with positive shifts shifting left and negative shifts shifting right. Thus, a|b is the same as a*(2^b). It is important to note that logical operators perform word comparisons, rather than bit-wise operations.

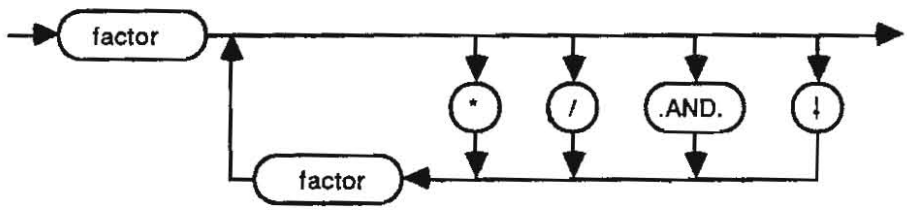


Figure 3-3. Syntax of a Term

Here are some examples of terms made up of factors, operators, and factors in an expression:

Expression	Result
`3/4`	`0`
`1+2*3`	`7`

A factor, as shown in Figure 3-4, is a constant, label, or expression enclosed in parentheses, or a factor preceded by .NOT.. .NOT. is the Boolean negation, producing true (1) if the following factor is false, and false (0) if it is true. Here, a label refers to a named symbol that cannot be resolved at assembly time. Constants are named symbols defined by a local EQU directive or global GEQU directive, or a decimal, binary, octal, or hexadecimal number, or a character constant.

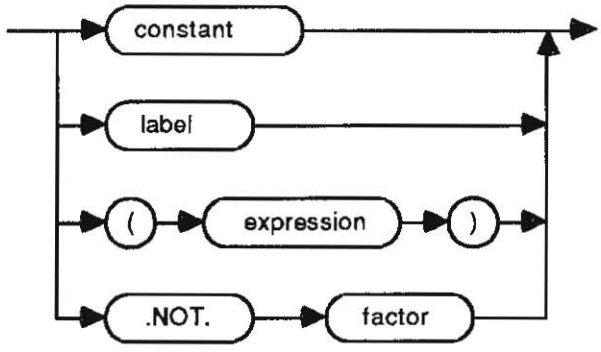


Figure 3-4. Syntax of a Factor

Here are some examples of factors in expressions:

Expression	Result
`6*7=42`	1
`.NOT.(4+6=10)`	0

The constants that are legal in a factor are shown in Figure 3-5.

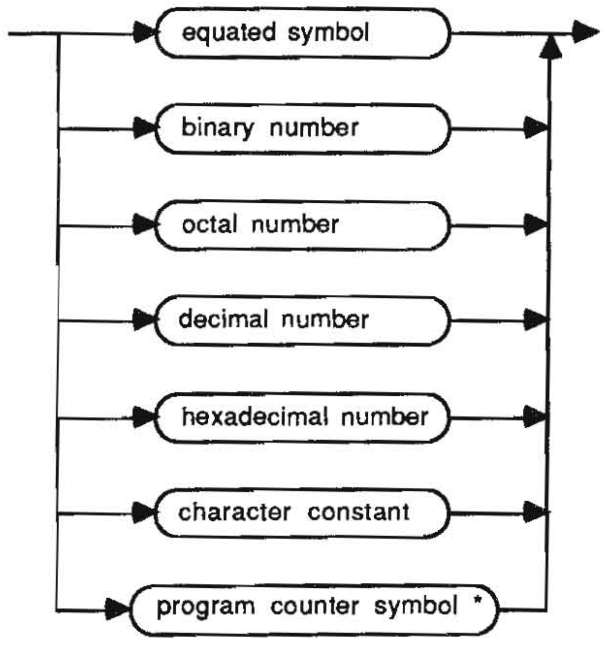


Figure 3-5. Syntax of a Constant

The syntax for a binary number is shown in Figure 3-6. It can consist of 1's and 0's and is prefixed with a percent sign (%).

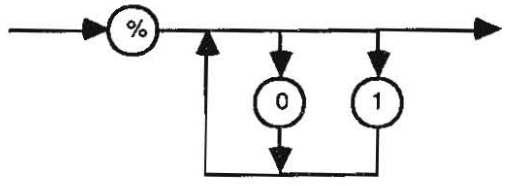


Figure 3-6. Syntax of a Binary Number

Here are some examples of binary constants and their decimal equivalents:

Binary constant	Decimal equivalent
%0	0
%1	1
%10	2
%10100101	165

The syntax for an octal number is shown in Figure 3-7. It consists of the numbers 0 through 7 and is prefixed by an at sign (@).

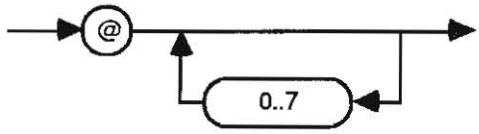


Figure 3-7. Syntax of an Octal Number

Here are some examples of octal constants and their decimal equivalents:

Octal constant	Decimal equivalent
@6	6
@7	7
@10	8

The syntax for a decimal constant is shown in Figure 3-8. It consists of the digits 0 through 9.

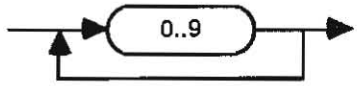


Figure 3-8. Syntax of a Decimal Number

Here are some examples of decimal constants:

```
3
457
```

The syntax for hexadecimal constants is shown in Figure 3-9. The constant can consist of the numbers 0 through 9 and the letters A through F and must be preceded by a dollar sign ($).

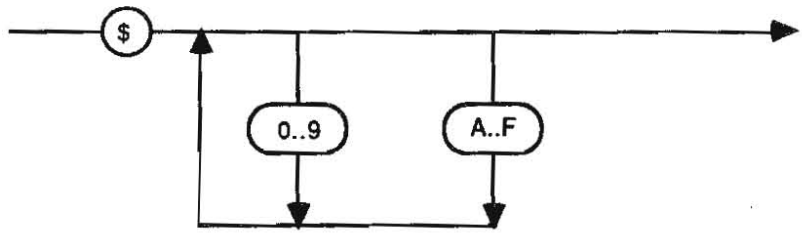


Figure 3-9. Syntax of a Hexadecimal Number

Here are some examples of hexadecimal constants and their decimal equivalents:

Hexadecimal constant	Decimal equivalent
$9	9
$A	10
$B	11
$89ABCD	9022413

The syntax for character constants is shown in Figure 3-10. A character constant may consist of any keyboard character except a single quotation mark ('), enclosed by single quotation marks, or any keyboard character except double quotation marks ("), enclosed by double quotation marks.

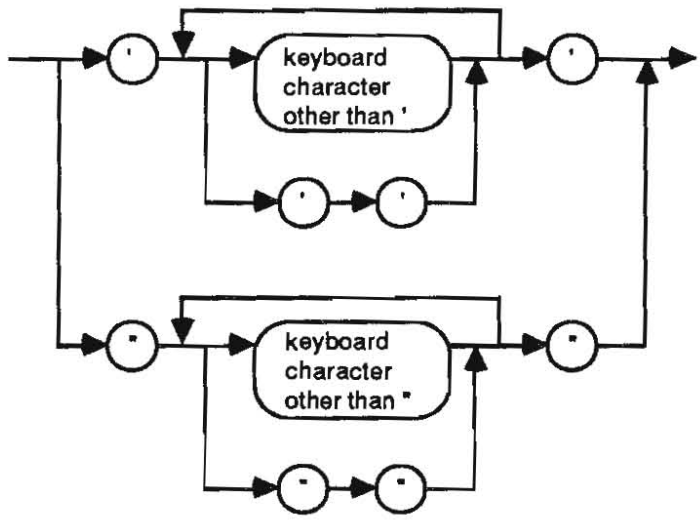


Figure 3-10. Syntax of a Character Constant

Here are some examples of character constants and their decimal equivalents:

Character constant	Decimal equivalent
`'a'`	97
`"A"`	65
`''''`	39

Comments

Comments can be contained in either comment fields or on separate lines of the source file.

The Comment Field

The comment field is optional. It is a place where you can document in English what your program is doing. The comment field exists entirely for your benefit; it does not affect object file generation.

There must be at least one space between the operand (or operation code, if there is no operand), and a comment. Some Assemblers require a semicolon before the comment; the APW Assembler requires a semicolon only if there is nothing else on the line. Comments usually start in column 41; the editor has a tab stop there, but the comment can start one space after the operand in any column. See the description of the `SETCOM` directive in the next chapter for more information.

Comment Lines

Each line of an assembly-language source file can be either an assembly statement or a comment. Comment lines have the same function as the comment fields in assembly statements, the lines are just longer. Comment lines are either blank lines, or lines beginning with asterisks (*), semicolons (;), or exclamation points (!). These statements appear in listings, but are otherwise ignored by the APW Assembler.

> **Important:** Do not use ampersands (&) in your comments. The & denotes the beginning of a symbolic parameter. See Chapter 6 for more information on this topic.

> **Note:** Comment lines must begin with either `*`, `;`, or `!` or the assembler will generate an error.

Chapter 4

APW Assembler Directives

A directive is a statement that tells an assembler to take some action. You can use APW Assembler directives to perform such tasks as

- controlling the program
- defining data
- defining symbols
- moving code to memory locations
- controlling files
- determining assembler options
- controlling output listing options

All directives except macro definition directives are valid within a source file; macro definition directives are valid only within macro files as described in Chapters 5 and 6.

Directive Formats

APW directives are coded in the same way as 65816 instructions, which means that they can have the fields shown here.

[label] *OPERATION* *[operand]* *[comment]*

As with instructions, the only directive field that is always required is the operation field. This field contains the name of the directive.

Directive Functions

The APW Assembler directives can be divided into the following functional groups: program control, data definition, symbol definition, memory designation, file control, APW Assembler options, and listing options.

Program Control Directives

The program control directives define code segments, data segments, and alternate entry points. These directives include

`DATA` Begin a data segment.

END	Complete a code or data segment.
ENTRY	Define an alternate entry point into a segment.
PRIVATE	Make an object code segment name unavailable to other object segments.
PRIVDATA	Make an object data segment whose name is unavailable to other object segments.
START	Begin a code segment.
USING	Make local data segment labels available to code segments.

Data Definition Directives

The data definition directives define constants, initialize memory, and reserve storage areas in code and data segments. They include

DC	Define a constant and initialize the value in memory.
DS	Define an area of storage in memory and initialize it to zeros.

Symbol Definition Directives

The symbol definition directives assign values to names. These directives let you assign names to numeric constants and expressions, so that you can use the names instead of the original values in your source text. The symbol definition directives include

EQU	Define a local label and set it equal to a value.
GEQU	Define a global label and set it equal to a value.

Code Location Directives

The code location directives move code to specified locations in memory. These directives include

ALIGN	Force a code or data segment to a boundary.
ORG	Designate program execution at a fixed location.
OBJ	Set the object code execution location, but don't force the code to load there.
OBJEND	Cancel the effect of the last OBJ.

File Control Directives

File control directives save assembled object files on disk and access files other than the current source file during assembly. These directives include

APPEND	Transfer processing to another file.
COPY	Transfer processing to another file and then return to the original file.
KEEP	Save assembled code on a disk.

APW Assembler Option Directives

The APW Assembler option directives control the assembly process. These directives include

CASE	Specify case sensitivity in source file labels.
CODECHK	Enable/disable linker check of JMP and JSR instructions.
DATACHK	Enable/disable linker check of data references.
DYNCHK	Enable/disable linker check of jumps to dynamic segments.
IEEE	Generate IEEE format numbers.
LONGA	Select accumulator size.
LONGI	Select index register size.
MERR	Set the maximum error level.
MSB	Set or clear the most significant bit of characters generated by DC directives.
NUMSEX	Set the order of bytes in floating-point numbers.
OBJCASE	Specify case sensitivity in object file labels.
SETCOM	Set comment column.
65C02	Enable/disable 65C02 code.
65816	Enable/disable 65816 code.

Listing Option Directives

The listing option directives allow you to specify certain options when you are listing the assembly to the screen, a file, or to the printer. The listing option directives include

ABSADDR	Enable/disable absolute addresses in output listings.
EJECT	Eject the page.
ERR	Print errors.
EXPAND	Expand DC directives.
INSTIME	Show instruction times.
LIST	Enable/disable list output.
PRINTER	Send output to printer.
SYMBOL	Enable/disable symbol table production.
TITLE	Generate a header.

The Comment Field and APW Assembler Directives

The same rules apply to comments in APW Assembler directives as in any other assembly statement. Comments are always optional (the field is shown in each directive for completeness). If present, there must be at least one space between the operand (or operation code, if there is no operand) and the beginning of the comment field. The comment can begin in any column. There are almost no restrictions on what you can put in this field. Symbolic parameters are expanded, so you should not follow an ampersand (&) with A . . . Z, ~, or _ unless you are aware of the consequences.

Settings Attribute and APW Assembler Directives

Several APW Assembler directives have operands which can be set to ON or OFF. You can use the settings attribute to find out how these operands are set at any time. The settings attribute is coded as S, followed by a colon and the name of the directive to be evaluated.

S:*directive*

If the operand is set to ON, a 1 is returned, if the operand is set to OFF, a 0 is returned. If you wanted to find out whether the 65816 instruction set was enabled so that you can do a 16-bit arithmetic operation, you could use S:65816. If the answer comes back as a 1, it means that the 65816 instruction set and addressing modes are enabled. The directives that accept ON or OFF operands follow:

ABSADDR	EXPAND	LONGI	65816
CASE	GEN	MSB	SYMBOL
CODECHK	IEEE	NUMSEX	TRACE
DATACHK	INSTIME	OBJCASE	
DYNCHK	LIST	PRINTER	
ERR	LONGA	65C02	

This example uses the settings attribute to determine which of two code sequences to assemble. In CASE=1, assemble the sequence of code that follows the label .A. Otherwise, fall through the settings test and assemble the sequence of code that follows the label .B. See Chapter 6 for descriptions of the conditional assembly directives AIF and AGO.

```
      AIF   S:CASE=1,.A
MAIN  START
      AGO   .B
.A
MAIN  START
      .
      .
      .
.B
```

APW Assembler Directives

This section describes the functions that can be performed with the APW Assembler directives.

ABSADDR Allow Absolute Addresses

[*label*] ABSADDR ON | OFF [*comment*]

The ABSADDR directive enables and disables absolute addresses in the APW Assembler output listing.

ON — Use this parameter to obtain a column of 6-byte addresses to the left of the relative offsets that the APW Assembler normally places in the output listing. The relative offsets still appear in the output.

OFF — The default is OFF. With this option, only relative offsets appear in the output.

The 'Hello World!' program with ABSADDR OFF shows the following relative offsets:

```
0003 0000                    LIST ON
0004 0000                    GEN OFF
0005 0000                    ORG $10000
0006 0000                    ABSADDR OFF
0007 0000             MAIN   START
0008 0000 4B                 PHK
0009 0001 AB                 PLB
0010 0002                    WRITELN #'Hello World!'
0011 001E A9 00 00           LDA   #0
0012 0021 6B                 RTL
0013 0022                     END
```

The 6-byte addresses produced with ABSADDR ON are the base number plus the number of bytes generated by the assembler since the last change in the base number.

```
0003 0000                              LIST ON
0004 0000                              GEN OFF
0005 0000                              ORG $10000
0006 010000 0000                       ABSADDR ON
0007 010000 0000           MAIN        START
0008 010000 0000 4B                    PHK
0009 010001 0001 AB                    PLB
0010 010002 0002                       WRITELN #'Hello World!'
0011 01001E 001E A9 00 00              LDA   #0
0012 010021 0021 6B                    RTL
0013 010022 0022                       END
```

If there is no ORG directive, the base number defaults to $000000. An ORG directive changes the base number to the value specified by the ORG's operand. The net effect is that this column shows the correct absolute memory location of the line, assuming that the listing is from a full assembly, and the file is loaded at the location specified in the ORG directive. It also assumes that you have not used the advanced linker, or the standard linker in some unusual way such as linking code in a library.

ALIGN Align to a Boundary

[*label*] ALIGN *anumb* [*comment*]

The ALIGN directive is used either prior to the start of a code or data segment, or within a segment. ALIGN directives used outside of segments must be restricted to page or bank alignments because these are the only kinds of alignments supported by the System Loader.

anumb This number is an absolute number and must be a power of 2.

When the ALIGN directive is used before a START, PRIVATE, DATA, or PRIVDATA directive, it directs the linker to align the segment to a byte boundary divisible by the absolute number in the operand of the ALIGN directive. To align a segment to a page boundary, for example, use the sequence

```
      ALIGN  256
ONCE START
       ...
       ...
       END
```

When an ALIGN directive is used within a segment, enough zeros are inserted to force the next byte to fall at the indicated alignment. The insertion is done at assembly time, so the zeros appear in the program listing. If an ALIGN directive is used within a segment, that segment must also be controlled by an ALIGN directive. The internal ALIGN directive must have equal or smaller boundaries than the external ALIGN directive.

ANOP Assembler No Operation

[*label*] ANOP [*comment*]

The ANOP directive allows you to define a label without an instruction. This directive does not cause the assembler to do anything.

label This parameter assumes the current value of the program counter.

APPEND Append a File

[*label*] APPEND *pathname* [*comment*]

The APPEND directive transfers processing to the beginning of the file specified by *pathname*. Any lines following the APPEND directive in the original file are ignored. The file indicated by *pathname* must be an APW source file, but does not have to be an

assembly-language file. Any language for which you have an assembler or compiler installed in your system will work if it conforms to the APW language standards.

pathname This parameter indicates a full pathname, including the prefix and filename, or a partial pathname, in which the current prefix is assumed. If you are using a full pathname, be sure that it begins with a slash (/). Do *not* precede *pathname* with a slash if you are using a partial pathname. No wild cards or device names may be used.

For example, inserting the APPEND directive in this code segment transfers processing to TEST.C so the remaining text is not assembled. If you want to finish assembling the text, use the COPY directive.

```
      GEN OFF
      ORG $10000
      ABSADDR OFF
MAIN  PHK
      PLB
      APPEND TEST.C
      WRITELN #'Hello World!'
      LDA   #0
      RTL
      END
```

CASE — Specify Case Sensitivity

[*label*] CASE ON | OFF [*comment*]

The CASE directive enables and disables case sensitivity in source file labels.

ON Use the ON option to make source file labels case sensitive.

OFF Use this parameter to make source file labels insensitive to case. The default is OFF. This means that LABEL and Label are treated as though they are the same.

If you are using CASE OFF and you have not set OBJCASE, all labels will be written to the object file in uppercase.

CODECHK Tell Linker to Check Jump Instructions

[*label*]	CODECHK	ON | OFF	[*comment*]

The CODECHK directive enables and disables the APW Linker check of 16-bit JMP and JSR instructions to insure that their targets are within the current load segment. The default is ON and it is unlikely that you will find a circumstance in which you will want to turn it off. With CODECHK ON, a jump out of range produces a linker error of:

```
Address not in the current bank
```

with a severity level of 8.

If you have a case, as in the example, where you need to turn CODECHK OFF no error will be generated for references outside of the current load segment.

In the example which follows, the two segments FOO and BAR are destined for different load segments.

```
      CODECHK OFF
FOO   START
      JSR   BAR
      RTL
      END

BAR   START FOOBAR
      LDA   #0
      RTS
      END
```

COPY Copy A File

[*label*]	COPY	*pathname*	[*comment*]

The COPY directive is used to transfer processing to the beginning of the file indicated by *pathname*. When the file named in *pathname* is completely processed, the assembly continues with the first line after the COPY directive in the original file.

pathname This parameter indicates a full pathname, including the prefix and filename, or a partial pathname, in which the current prefix is assumed. If you are using a full pathname, be sure that it begins with a slash (/). Do *not* precede *pathname* with a slash if you are using a partial pathname. No wild cards or device names may be used.

Compare COPY with the APPEND directive, which does not return to the original file. A copied file can copy another file; the depth is limited only by the amount of available memory.

DATA Define Data Segment

label DATA [*loadseg*] [*comment*]

DATA marks the beginning of a data segment. Its purpose is to store data definitions and any instruction found in this segment will be flagged as an error. The data segment continues until an END directive is reached.

label Each DATA directive requires a label, which functions as the data segment name in the object file. The label is global in scope. No more than 127 data segments may be defined in any one program. Each data segment must have a unique object segment name.

loadseg The optional load segment name may be up to 10 characters long. It may begin with an alphabetic character, an underscore (_), or a tilde (~). The name may contain letters, digits, underscores (_), and tildes (~), but no spaces. Several object segments may have the same load segment name. The linker places all object segments with the same name in the same load segment. If you use a LinkEd file to control the APW Linker, the load segment name you specify here will be ignored unless you ask for it specifically. By default, *loadseg* is case insensitive and uppercase in the output file. The OBJCASE value is respected.

Labels used within a data segment are local, but they cannot be duplicated in other data segments, or as global labels. They become available to outside segments that have USING directives that reference the data segment. The USING directive has the effect of making the labels local to both the data area and the segment where the directive appears.

Note: If you use a comment with the DATA directive, the comment must start at the SETCOM column (usually the forty-first). A comment that begins before the SETCOM column will be interpreted as a *loadseg* parameter.

DATACHK Check Data References

[*label*] DATACHK ON | OFF [*comment*]

The DATACHK directive enables and disables checking by the APW Linker for 16-bit data references outside of the current load segment. With DATACHK ON, such a reference causes a severity level 8 error:

```
Address is not in current bank
```

The following example code segment will be loaded under the default name of 10 space characters. The data segment will be loaded with the name FOOBAR. In order to access ITEM (which is a 16-bit data reference), you must have a USING directive and turn DATACHK OFF.

```
      DATACHK OFF
FOO   START
      USING  BAR          USING allows you to access the data
                            segment BAR
      LDA    ITEM         The 16-bit data reference
      RTL
      END

BAR   DATA   FOOBAR
ITEM  DS     2
      END
```

DC Declare Constant

[label] DC *condef[,condef,...]* *[comment]*

The DC directive defines constants within a program that initialize values in memory. The constants may be any of these value types:

A*x*	Address
B	Binary
C	Character
D	Double-precision floating point
E	Extended floating point
F	Floating point
H	Hexadecimal constant
I*x*	Integer
R	Reference an address
S*x*	Soft reference

condef The constant definition *condef,* begins with an optional repeat count *(rcount)*, which must be in the range 1 to 255 decimal, followed by an identifier describing the value type. The variable being defined is placed in the object file as many times as specified by the repeat count (*rcount*). The identifier is followed by values, enclosed in single quotation marks, separated from each other by commas. Optionally, additional constant definitions may follow the first, separated by commas. The format for *condef* is

condef = [rcount]identifier'value,value,...'

The following example has a repeat count of 2, an integer identifier (I), and two values followed by another repeat count, an integer identifier and another value. The result places four 16-bit integers and one 8-bit integer into memory.

```
LABEL DC   2I'2,3',1I'4'
```

The hexadecimal values resulting from this directive are

```
02  00  03  00  02  00  03  00  04
```

Some important points about the DC directive:

- You can mix value types in one line.
- Except for B, C, or H types, you can have multiple values of the same type on one line, each value separated by a comma.
- Character strings defined by DC directives have their most significant bits cleared. You must use the MSB directive if you want to change this.

Address (A*x*)

Address identifiers are used to build tables of addresses. The A DC statement generates a 1- to 4-byte integer address. If you omit the *x*, a 2-byte address is generated, by default.

Binary (B)

The value type B designates a binary number. Binary values consist of ones, zeros, and spaces, enclosed by single quotation marks. The spaces are removed by the assembler before the bit values are stored. If a byte is left partially filled, it is padded on the right with zeros, as shown in the second example:

Example Code	Hexadecimal Values
`DC B'01 01 01 10'`	`56`
`DC B'111111111'`	`FF80`

Character (C)

Character strings are indicated by the value type C. The string, enclosed in quotation marks, may contain any sequence of keyboard characters. If you use a quotation mark within the string, enter it twice to distinguish it from the end of the string:

Example Code	Hexadecimal Values

`DC C'HOW''S THE WEATHER?'`	`48 4F 57 27 54 20 54`
	`48 45 20 57 45 41 54 48`
	`45 52 3F`

Normally, strings are stored with the high-order bit off, corresponding to the ASCII character set. If you are writing characters directly to the Apple II screen, use the MSB directive to set the high bit on.

Hexadecimal (H)

The value type H indicates a hexadecimal value. Hexadecimal values consist of the digits 0 through 9, the hexadecimal digits A through F, and spaces, all enclosed by single quotes. The spaces are removed before the value is stored, two digits per byte. Hexadecimal values are stored as pairs of digits. If you code an odd number of digits, the assembler pads the last byte with 4 bits of zeros on the right. The first example shows what happens when you ask the assembler to store an odd number of digits:

Example Code	Stored Values
`DC H'01234ABCDEF'`	`01 23 4A BC DE F0`
`DC H'1111 2222 3333'`	`11 11 22 22 33 33`

Integers I(*x*)

Integers are indicated by a value type of I. Because integer sizes can vary between 1 and 8 bytes, the size is indicated by a digit from 1 to 8 following the I. If you omit the length, a 2-byte (16-bit) integer is generated. All integers are stored least significant byte first. Expressions can be used in DC statements to declare integers of 1 to 4 bytes. Expressions larger than 4 bytes are illegal.

This DC directive is illegal:

```
DC    I5'4+5'
```

This DC directive is fine:

```
DC    I5'9'
```

Integers longer than four bytes can be represented only as signed decimal constants.

The table that follows gives the valid range of signed integers through a length of eight bytes. See the M16.INTMATH macro file for macros to perform integer math operations:

Size in bytes	Smallest value	Largest value
1	-128	127
2	-32768	32767
3	-8388608	8388607
4	-2147483648	2147483647
5	-549755813888	549755813887
6	-14073748355328	14073748355327
7	-36028797018963968	36028797018963967
8	-9223372036854775808	9223372036854775807

Floating Point (F, D, E)

There are four kinds of floating-point numbers designated by three value types: F, D, and E. All of the floating-point values are entered as signed floating-point numbers, with optional signed exponents starting with E. The numbers are stored least significant byte first, by default.

Floating Point (F)

The value type F designates a floating-point number that is stored as a 4-byte (32-bit) floating-point number in a format compatible with the Standard Apple Numeric Environment (SANE). Floating-point numbers designated by F can range from approximately 1E-38 to 1E+38. The mantissa is accurate to over 7 decimal digits.

Example:

```
DC F'3,-3,.35E1,6.25E-2'
```

Double-Precision Floating Point (D)

Double-precision floating-point values are identical to F, except that an 8-byte (64-bit) number is generated. Numbers can range from about 1E-308 to 1E+308. The mantissa is accurate to slightly more than 15 decimal digits.

Example:

```
DC D'35E105,6.25E-202'
```

Extended Floating Point (E)

Extended floating-point values are identical to F and D, except that they generate 10-byte (80-bit) numbers. Numbers can range from about 1E–4931 to 1E+4931(they are calculated by formula). The mantissa (which is also calculated) is accurate to slightly more than 19 decimal digits.

Example:

```
DC E'42E1000,8.73E-1100'
```

Applesoft Floating Point (F)

To generate floating-point numbers that are compatible with the Apple II Applesoft floating-point firmware, turn off the IEEE format and use the F value type floating-point format. A 5-byte (40-bit) Applesoft-style floating-point number representation will be generated. These numbers are always stored most significant byte first.

Reference an Address (R)

The value type R generates a reference to an address in the object file without saving the address in the executable file. This allows a program to note that a subroutine will be needed from the subroutine library without reserving storage for the address. This reference does not take up space in the finished program.

Usually only subroutines in a library referenced by JSR, or JSL are added to the load file by the APW Linker. Sometimes code in the library is not referenced directly, but still needs to be added to the load file. The R value type can serve this purpose without generating any code in the load file.

Soft Reference (S*x*)

The value type S generates 1 to 4 bytes of storage for each address in the operand, but does not instruct the linker to link the segments into the final program. If the segment is not linked, the executable file produced by the linker will have $0000 as the 1- to 4-byte address. This allows a table of addresses to be built, but only those segments requested elsewhere in the program have their addresses placed in the table.

DIRECT — Set Direct Page Value

[label] DIRECT *expression* |OFF *[comment]*

The DIRECT directive controls the generation of errors and promotion to absolute addresses when no direct-page addressing mode is available. The DIRECT directive is either set to OFF or set to an expression which resolves to a 16-bit constant. If the operand is a number, the assembler promotes direct-page addresses to absolute addresses when no direct-page addressing mode is available. It does this by adding the 16-bit constant to the direct-page value. For example

```
LDA    4,Y
```

does not automatically promote to LDA |$0004,Y because the direct page may not be a 0. If, however you specify

```
DIRECT $1200

LDA    4,Y
```

the assembler can generate LDA $1204,Y because you set the location of the direct page.

DIRECT OFF causes the assembler to generate errors for direct-page addresses when no direct-page addressing mode is available.

DS — Define Storage

[label] DS *expression* *[comment]*

The DS directive defines areas of memory for program use. It also initializes these locations to zeros.

expression The number of bytes of storage to reserve.

The following directive defines 50 bytes of memory initialized to zeros:

```
FREE DS    50
```

DYNCHK — Check References to Dynamic Segments

[label] DYNCHK ON | OFF *[comment]*

The DYNCHK directive enables and disables a check on how jumps are made to dynamic segments. With DYNCHK set to ON, jumps are legal to dynamic segments only via JSL instructions. With DYNCHK set to OFF, you can jump to dynamic segments via JSRs without error as shown in the example. Remember that if you set DYNCHK to OFF the directive CODECHK is also set to OFF so that there is no longer a check on whether or not your jump target is within the current load segment.

```
      DYNCHK OFF
FOO   START
      JSR    BAR     Go to location in dynamic segment.
      RTL
      END

BAR   START FOOBAR
      KIND   $80     Set this segment header to dynamic.
      LDA    #0
      RTL
      END
```

EJECT — Eject the Page

[label] EJECT *[comment]*

The EJECT directive causes printer output to skip to a new page. The directive has no effect on output going to the console.

END

End Program Segment

[*label*] END [*comment*]

The END directive indicates the end of a code or data segment. It directs the APW Assembler to print the local symbol table (if the +S option is used on the command line, or if the SYMBOL ON directive is in effect) and delete the local labels from the symbol table. The END directive has no operand and usually no label. The END directive is required. If it is omitted, the assembler generates an error message.

ENTRY

Define Entry Point

label ENTRY [*comment*]

The ENTRY directive defines a global label that corresponds to a location within the segment. This label can be used as an alternate entry point into the segment, or for other reasons such as making data within a code segment visible to code in other segments.

label The required label is the name of the new entry point. The label is global.

The first example provides alternate entry points at SUB1 and SUB2.

```
MAIN START
          ...
          ...
 JSL      SUB1
          ...
          ...
 JSL      SUB2
          ...
          ...
          END
SUBS      START
          ...
          ...
SUB1      ENTRY
          ...
          ...
          RTL
SUB2      ENTRY
          ...
          ...
          RTL
          ...
          ...
          END
```

This example makes the address of the DC directive visible to other code segments.

```
MAIN    START
        ...
        ...
        LDA     VAR1
        ...
        ...
        END
SUB     START
        ...
        ...
VAR1    ENTRY
        DC I2'$FF'
        END
```

EQU Equate

label EQU *value* [*comment*]

The EQU directive defines the name in the label field and assigns the value in the operand to it. This allows you to assign a name to a numeric value and use the name instead of the number in further operands.

value The operand may contain a label that already has a value, an expression, or a constant defined by an earlier equate.

label The label is required. If the label field does not contain a value, an error is generated.

If you need to define a label without generating code, you can use the ANOP (No Operation) directive.

You should define constants before they are used. It is customary to put all equates in one data segment. Although this is a strict requirement only when the constant is used later as a direct-page or long address, it is good programming practice.

The following commands place a carriage return at the end of a line of characters. First, define the RETURN symbol and set it equal to the code for a carriage return using the EQU directive. Then use a DC directive to define a 1-byte integer to contain the value.

```
RETURN  EQU     $D
        DC      C'This line ends with a carriage return'
        DC      I1'RETURN'
```

Some additional examples of EQUATE directives follow:

```
ONE     EQU     1
TWO     EQU     1+1
FOUR    EQU     TWO*TWO
L1      EQU     L0/2+7
HERE    EQU     *
```

ERR Print Errors

[*label*] ERR ON | OFF [*comment*]

The ERR directive enables and disables error listings when the LIST directive is set to OFF.

ON — If ON is specified, error lines continue to be printed, with the LIST directive set to OFF. The default is ERR ON.

OFF — If this operand is specified, errors are no longer printed, but the number of errors found is still listed at the end of the assembly.

EXPAND Expand DC Statements

[*label*] EXPAND ON | OFF [*comment*]

The EXPAND directive enables and disables the listing of code generated by DC directives in output listings.

ON — The ON option causes all bytes generated by DC directives to be shown, four bytes per line, in the output listing. A maximum of sixteen bytes per DC directive, or four lines, can be displayed.

OFF — The OFF option limits code shown in the output listing to the first four bytes of a DC directive, or one line in the listing. The default is OFF.

With EXPAND set to OFF, the DC directive `L DC E'1.1'` generates the line

```
0006 3F FF 8C CC  L         DC E'1.1'
```

When EXPAND is set to ON, the same DC directive generates the lines

```
0006 3F FF 8C CC  L         DC E'1.1'
000A CC CC CC CC
000E CC CD
```

GEN Generate Macro Expansions

```
GEN          ON | OFF
```

If GEN is turned on, all lines generated by macro expansions are shown on the output listing. Each line generated by a macro has a + character to the left of the line. If GEN is turned off, only the macro call is printed in the assembly listing. Errors within the macro expansion are still printed, together with the line causing the error.

GEQU Global Equate

```
label        GEQU        expression        [comment]
```

Like the EQU directive, GEQU defines the name in the label field and assigns the value in the operand to it. Unlike EQU, however, GEQU's label is saved in the global symbol table. This makes the label available to all program segments.

label The label is required. They are included in the object file, so library routines can use global equates to make constants available to the main program.

expression The *expression* operand may contain a label that already has a value, an expression, or a constant defined by an earlier equate.

Labels defined by the GEQU directive are resolved at assembly time if they are defined before they are referenced; otherwise, they are resolved at link time. The following example will be resolved at assembly time:

```
A       GEQU   5
MAIN    START
        LDA    #A
        END
```

but the following code sequence will be resolved at link time:

```
MAIN    START
        LDA    #A
        END
D       DATA
A       GEQU   5
        END
```

IEEE — Generate IEEE Format Numbers

[*label*]	IEEE	ON | OFF	[*comment*]

The IEEE directive enables and disables Applesoft compatible format for floating-point constants created with DC directives. This directive has no effect upon D and E value types of floating-point constants.

ON — In its default setting, DC directives generate numbers compatible with the IEEE floating-point standard. The default is ON.

OFF — If IEEE is turned off, the F value type floating-point constants created with the DC directives will be in Applesoft compatible format.

For example, if you enter 5.5 with IEEE set to ON, the hexadecimal value generated is 0000B040. If you enter the same number with IEEE set to OFF the Applesoft format number generated is 8330000000.

INSTIME — Show Instruction Times

[*label*]	INSTIME	ON | OFF	[*comment*]

The INSTIME directive enables and disables instruction cycle times in the output listing of the APW Assembler.

OFF — With this option no cycle times are shown in the output listing. The default is OFF.

ON — This option causes a column of instruction cycle times to be inserted in the output listing immediately before the text of the source line. This column is two characters wide. It shows the number of machine cycles required to execute the assembly-language instruction appearing on the line. The column is blank for macros, directives, and comments. Otherwise, the first character indicates the number of cycles; the second character may be an asterisk (*) if the cycle time is variable. There are many reasons that an asterisk may occur, for example, a page boundary was crossed, a branch was taken, a move was made, and so forth. Consult one of the commercially available reference sources given in the preface of this manual for more information. .

With INSTIME set to ON cycle times are displayed in lines 9, 10, 12, and 13 of the example.

Cycle Times

```
0004 0000                          LIST ON
0005 0000                          MERR 0
0006 0000                          MSB ON
0007 0000                          GEN OFF
0008 0000                    MAIN  START
0009 0000 4B          3            PHK
0010 0001 AB          4            PLB
```

```
0011 0002                          WRITELN #'Hello World!'
0012 001E A9 00 00     2*          LDA    #0
0013 0021 6B           6           RTL
0014 0022                          END
```

KEEP Keep Object File

KEEP *objname* [*comment*]

The KEEP directive saves assembled code on a disk as an object file. This file can be used as input to the linker to generate an executable load file. The KEEP directive may be used only once per source file, and must appear before any code generating statements.

objname The name of the object file can contain a maximum of 10 letters, digits, and periods.

Important: Remember that if you have a KeepName variable set in your LOGIN file and a KEEP directive in your source file, the KeepName will override the name you used in the source file. If you use a shell command-line KEEP, it will override both the KeepName and any KEEP directives in your source file.

KIND Specify Object Segment Type and Attributes

[*label*] KIND *number* [*comment*]

The KIND directive sets the kind field of the object segment header created when you assemble this source code segment. If you use multiple KIND directives or you mix segments that have KIND directives with segments that don't have KIND directives, be sure that the resulting values match (have the same types and attributes) for all segments that will eventually occupy the same load segment.

number This value can be in the range 0 to FF and has the following meanings

Bits 0-4	Types
$00	code segment
$01	data segment
$02	jump table segment
$04	pathname segment
$08	library dictionary segment
$10	initialization segment
$11	absolute bank segment
$12	direct page/stack segment

	Attributes
Bit 5	position independent (1=YES)
Bit 6	private (1=YES)
Bit 7	static = 0 dynamic = 1

Attributes can be combined with types in a single KIND directive. For example a dynamic initialization segment has a KIND directive value of $90. A private code segment is an object code segment whose name is only available to other code segments within the same object file. A private data segment is an object data segment whose labels are available only to other code segments within the same object file. Absolute bank segments are relocatable within a specified bank. Direct page/stack segments are used to preset direct and stack registers for an application.

To create a dynamic code segment you could use the following code:

```
A       START  LOADSEG1
        KIND   $80
        ...
        ...
        END
```

This causes the linker to create a load segment named LOADSEG1 whose kind field is $80 (dynamic code segment).

LIST — List Output

[label] LIST ON | OFF *[comment]*

The LIST directive enables and disables the APW Assembler output listing.

OFF — If the listing is turned off, error lines may still be produced. See the ERR directive.

ON — A listing of the assembler output is sent to the current output device. The default is ON.

Example using LIST ON:

```
0003 0000                          LIST ON
0004 0000                          GEN OFF
0005 0000                  MAIN    START
0006 0000 4B                       PHK
0007 0001 AB                       PLB
0008 0002                          WRITELN #'Hello World!'
0009 001E A9 00 00                 LDA   #0
0010 0021 6B                       RTL
0011 0022                          END
```

```
11 source lines
1 macros expanded
14 lines generated
```

This example using `LIST OFF` shows that only summaries of the processing are contained in the output listing.

```
Pass 1: MAIN
Pass 2: MAIN

11 source lines
1 macros expanded
14 lines generated
```

LONGA Select Accumulator Size

[*label*]	LONGA	ON | OFF	[*comment*]

The 65C816 processor can perform both 16-bit and 8-bit operations involving the accumulator. The size of the accumulator and amount of memory affected by instructions like `LDA`, `STA`, and `INC` are controlled by a bit in the processor status register. At assembly time, the APW Assembler has no idea how that bit will be set at run time, so it is the responsibility of the programmer to tell the assembler using this directive.

`ON`	This option indicates 16-bit operations. The default is `ON`.
`OFF`	This option indicates 8-bit operations.

The only difference between `LONGA ON` and `LONGA OFF` in the assembled program is the number of bytes placed in the code stream when an immediate load is performed. For example:

```
LONGA ON
LDA #2          2 byte operand
LONGA OFF
LDA #2          1 byte operand
```

Important: The status bit that the processor uses at run time must be set separately.

LONGI Select Index Register Size

[*label*]	LONGI	ON | OFF	[*comment*]

The 65C816 processor can perform both 16-bit and 8-bit operations involving the X and Y registers as well as the accumulator. The size of the X and Y registers is controlled by a bit in the processor status register. At assembly time, the APW Assembler has no idea how

that bit will be set at run time, so it is the responsibility of the programmer to tell the assembler, using this directive. Specifically, the LONGI directive controls the number of bytes generated by immediate loads to the X and Y registers when using the 65C816 processor.

ON This option indicates 16-bit operations. The default is ON.

OFF This option indicates 8-bit operations.

LONGI controls the number of bytes placed in the code stream when an immediate load is performed. For example:

```
LONGI ON
LDX #2          2 byte operand
LONGI OFF
LDY #2          1 byte operand
```

MERR Maximum Error Level

[*label*] MERR *expression* [*comment*]

The MERR directive determines whether or not output code assembled with the ASML or AMSLG commands will be immediately linked and executed.

[*label*] The label is optional.

expression The *expression* operand contains the maximum level severity code that can be detected in an assembly before the link is aborted. The default value is zero. The error levels are described in Appendix C.

MSB Most Significant Character Bit

[*label*] MSB ON | OFF [*comment*]

The MSB directive causes character constants and characters generated by DC directives to have their sign bits set or cleared.

ON Character constants and characters generated by DC directives have bit seven turned on, and appear normal on the Apple IIGS text display.

OFF The default is OFF. This conforms to the ASCII character convention of using only the least significant seven bits in a byte.

NUMSEX Set Floating Point Byte Order

[*label*]	NUMSEX	ON | OFF	[*comment*]

The NUMSEX directive causes floating-point numbers in DC directives to reverse the order of their bytes.

ON	Floating-point numbers generated by DC directives have the most significant byte first.
OFF	The default is OFF. This conforms to the SANE convention of least significant byte first.

For example, if you have a program that contains the DC directive

```
L       DC    F'1.1'
```

and you set the NUMSEX directive to ON, the hexadecimal value that is generated is

```
CONST   ($04) | 3F8CCCCD
```

The same DC directive, with the NUMSEX directive set to OFF generates this hexadecimal value:

```
CONST   ($04) | CDCC8C3F
```

OBJ Designate Destination

[*label*]	OBJ	*val*	[*comment*]

The OBJ directive sets the program counter so that code that follows this directive is assembled as if it were located at the machine code address given in the operand. The OBJ directive has no effect on the actual physical location where the code is assembled; it is used when part of a program must be moved (to *val*) before execution.

val	The absolute address where the code is to be executed.

The effect of OBJ can be canceled by either an END directive, an OBJEND directive, or another OBJ directive.

Note: Code produced in this way does not need to be relocated by the System Loader because it contains references to absolute addresses. It may, however, be included within a segment that is relocatable.

Important: References to local labels must use the > operator to force them to absolute long addressing mode. Otherwise, they will cause addressing errors.

This command is provided for those programs that have their own routines to move segments to specific absolute addresses. We strongly recommend that you not use this command, but take advantage of the capabilities of the Apple IIGS System Loader and Memory Manager instead. Programs that do their own loading and memory management are very unlikely to work successfully with any other Apple IIGS routines.

OBJCASE Specify Case Sensitivity in Object Files

[*label*]	OBJCASE	ON | OFF	[*comment*]

The OBJCASE directive enables and disables case sensitivity in object file labels, object segment names, and load segment names.

ON Use this parameter to make object file labels case sensitive. These labels may, or may not, be case sensitive in the source file.

OFF Use the OFF parameter to make object file labels insensitive to case, whether or not they are treated as case sensitive in the source file. The default is OBJCASE OFF.

Setting CASE also sets OBJCASE, so that if you expect the behavior of a label to be different in an object file from its behavior in a source file, you must specify the OBJCASE directive last.

OBJEND End Destination Segment

[*label*]	OBJEND	[*comment*]

The OBJEND directive indicates the end of a code segment that began with an OBJ directive. The OBJEND directive has no operand, and usually no label.

ORG Designate Origin

[*label*]	ORG	*memloc*	[*comment*]

The ORG directive sets the program counter so that a program begins execution at a fixed location. ORG directives can be used at the beginning of a source file before the first START, PRIVATE, DATA, or PRIVDATA directives in the source code. The ORG directive can also be positioned before any subsequent START, PRIVATE, DATA, or PRIVDATA directives to force that segment to a particular fixed address. Finally, an ORG directive may be used within a program segment.

memloc The operand field contains a constant, or an expression which evaluates to a constant, that is the absolute address at which execution is to begin.

Important: Because the Apple IIGS system uses a relocating loader, you cannot ever be guaranteed that a specific area in memory will be available to your program. We strongly recommend that you not use this command, but take advantage of the capabilities of the Apple IIGS System Loader and Memory Manager instead. Programs that do their own loading and memory management are very unlikely to work successfully with any other Apple IIGS routines.

When the ORG directive is used to force a segment to a particular fixed address, zeros are inserted until the desired location is reached. This action is performed by the linker as the final load file is created.

When an ORG directive is used before the first segment, the entire load file is loaded at that location. If the ORG directive is used outside of any other segment but the first, the first segment must also be preceded by an ORG directive.

When the ORG directive is used inside a program segment, the operand must be an asterisk (*) indicating the current location counter, followed by a + or –, and an expression that evaluates to a constant. The location counter is moved forward or backward by the indicated amount as in the examples that follow:

```
ORG   *+2
```

moves the location counter forward by 2. This is equivalent to the DS directive

```
DS   2
```

An ORG directive with a negative operand moves the location counter backward, as in

```
ORG   *-1
```

which deletes the last byte generated within the segment. It is not possible to delete more bytes than have been generated by the current segment.

PRINTER — Send Output to Printer

[*label*]	PRINTER	ON | OFF	[*comment*]

The PRINTER directive controls output to the printer.

[*label*] The label is optional, but global if used.

ON The ON option causes output to be sent to the printer. A printer capable of printing at least 80 columns is expected. The slot number and printer characteristics may be changed. Refer to the *Apple IIGS Programmer's Workshop Reference* manual.

OFF The OFF option causes output to be sent to the video display. The default is OFF.

PRIVATE Define a Private Code Segment

[*label*]	PRIVATE	[*loadseg*]	[*comment*]

The PRIVATE directive works like the START directive, except that the object segment name cannot be accessed from outside of the object file in which it is created. Segments in different files can have the same names as long as all but one is private. This mimics the capabilities of the C language static variable. Any labels declared inside of the segment that would normally be global, are now private.

label Each PRIVATE directive requires a label, which becomes the name of the object segment produced by the assembler. The label is global in scope.

[*loadseg*] The optional load segment name may be up to 10 characters long. It may begin with an alphabetic character, an underscore (_), or a tilde (~). The name can contain letters, digits, underscores (_), and tildes (~), but no spaces. Several object segments may have the same load segment name. The linker places all object segments with the same load segment name in the same load segment. If you use a LinkEd file to control the APW Linker, the load segment name you specify here will be ignored unless you specifically request it.

Note: If you use a comment with the PRIVATE directive, the comment must start at the SETCOM column (usually the forty-first). A comment that begins before the SETCOM column will be interpreted as a *loadseg* parameter.

PRIVDATA Define a Private Data Segment

label	PRIVDATA	[*loadseg*]	[*comment*]

The PRIVDATA directive operates like the DATA directive, except that the object segment name is not accessible from outside the object file in which it appears. Segments in different files can have the same names as long as all but one is private. This mimics the capabilities of the C language static variable. Any labels declared inside of the segment that would normally be global are now private.

label Each PRIVDATA directive requires a label, which becomes the name of the object data segment produced by the assembler. Each data segment within a file must have a unique object segment name.

[*loadseg*] The optional load segment name may be up to 10 characters long. It may begin with an alphabetic character, an underscore (_), or a tilde (~). The name can contain letters, digits, underscores (_), and tildes (~), but no spaces. Several object segments may have the same load segment name. The linker places all object segments with the same load segment name in the same load segment. If you use a LinkEd file to control the APW Linker, the load segment name you specify here will be ignored unless you specifically request it.

Note: If you use a comment with the PRIVDATA directive, the comment must start at the SETCOM column (usually the forty-first). A comment that begins before the SETCOM column will be interpreted as a *loadseg* parameter.

RENAME Rename Operation Codes

The RENAME directive allows you to give a new name to an existing operation code. Use this directive to prevent conflicts if you are using the APW Assembler with a macro library that implements a cross assembler for another microprocessor that has operation codes that conflict with the 65816 instruction set or with APW directives. The operand is the old operation code followed by the new one.

RENAME *oldop,newop* [*comment*]

oldop, The old operation name can be eight characters or less and may not contain either spaces or the & character.

newop The new operation name is eight characters or less, and contains no spaces or the & character.

The RENAME directive cannot be used within a segment.

To change an LDA instruction to an instruction called LOAD use the RENAME directive

```
RENAME    LDA,LOAD
```

SETCOM Set Comment Column

[*label*] SETCOM *expression* [*comment*]

The SETCOM directive indicates the start of the comment column. The assembler will not search beyond this column for an operand (although it will search for an operation code). Comments may, of course, begin before this column if an operation code also occurs earlier. If the end of an operation code occurs past this column the operand must begin one space after the end of the operation code or it will not be found by the APW Assembler. If you use both SETCOM and a space after an operand, the space has priority.

expression The column number is a constant in the range of 1 to 255. The default is 40.

Note: A line consisting of a comment only must begin with one of the comment characters (; * !) even if the comment text occurs after the comment column number.

65C02 Enable 65C02 Code

[*label*] 65C02 ON | OFF [*comment*]

This directive enables and disables the instructions and addressing modes of the 65C02 processor.

ON Use this parameter to enable the 65C02 instructions and addressing modes.

OFF Use this parameter to disable the 65C02 instructions and addressing modes. The default is OFF.

65816 Enable 65816 Code

[*label*] 65816 ON | OFF [*comment*]

This directive enables and disables the instructions and addressing modes of the 65C816 processor.

ON Use this parameter to enable the 65816 instructions and addressing modes. The default is ON.

OFF When off, only 6502 instructions are valid unless you have previously set 65C02 ON.

START Start Segment

label START [*loadseg*] [*comment*]

The START directive marks the beginning of a named code segment. The code segment extends to the next END directive.

label Each START directive requires a label, which becomes the name of the object code segment produced by the assembler. The label is global in scope. Each code segment must have a unique object segment name.

[*loadseg*] The optional load segment name may be up to 10 characters long. It may begin with an alphabetic character, an underscore (_), or a tilde (~). The name can contain letters, digits, underscores (_), and tildes (~), but no spaces. Several object segments may have the same load segment name. The linker places all object segments with the same load segment name in the same load segment. If you use a LinkEd file to control the APW Linker, the load segment name you specify here will be ignored unless you specifically request it.

Note: If you use a comment with the START directive, the comment must start at the SETCOM column (usually the forty-first). A comment that begins before the SETCOM column will be interpreted as a *loadseg* parameter.

At least one START, PRIVATE, PRIVDATA, or DATA directive is required. If all of these are omitted, the APW Assembler generates an error message.

SYMBOL Print Symbol Tables

[*label*]	SYMBOL	ON|OFF	[*comment*]

The SYMBOL directive controls the printing of the symbol table. An alphabetized listing of all local symbols is printed following each END directive. After all processing is complete, global symbols are printed.

[label] The label is optional, but global if used.

OFF No symbol table is printed. The default is OFF.

ON The symbol table is printed.

For example, the program MYPROG

```
0003  0000                          LIST ON
0004  0000                          SYMBOL ON
0005  0000                          GEN ON
0006  0000                MAIN      START
0007  0000 4B                       PHK
0008  0001 AB                       PLB
0009  0002                          WRITELN #'Hello World!'
      0002                +         ANOP
      0002 80 0D          +         BRA ~B2
      0004 0C 48 65 6C    +~A2      DC I1'1:&STR',C'Hello World!'
      0011 F4 00 00       +~B2      PEA ~A2|-16
      0014 F4 04 00       +         PEA ~A2
      0017 A2 0C 1A       +         LDX #$1A0C
      001A 22 00 00 E1    +~C2      JSL $E10000
0010  001E A9 00 00                 LDA #0
0011  0021 6B                       RTL
0012  0022                          END
```

produces the following symbol table with SYMBOL set to ON:

```
Local  Symbols

   000004 ~A2          000011 ~B2          00001A ~C2
```

TITLE Print Header

[*label*] TITLE [*string*] [*comment*]

The TITLE directive is used to place page numbers at the top of each page sent to the printer.

string If you use this optional string, it is printed at the top of each page, immediately after the page number. If it is coded, it must be a legal string and must be enclosed in single quotation marks if it contains spaces or starts with a single quotation mark. If the string is longer than 60 characters, it is truncated.

TRACE Trace Macros

TRACE ON | OFF

Most conditional assembly directives are not printed by the assembler. This is to avoid printing lines of output that have no real effect on the finished program. It is, however, sometimes desirable to see all of the lines the assembler processes especially when debugging macros. To do this, use TRACE ON. The default is TRACE OFF.

The assembler output with TRACE set to ON looks like this:

```
0006 0000                          TRACE   ON
0007 0000                          CASE    ON
0008 0000                          LIST    ON
0009 0000                          TITLE 'SAMPLE PROGRAM'
0010 0000                TEST      START
0011 0000                          USING TESTDAT
0012 0000                          ADD     I,J,K
     0000               +          ANOP
     0000               +          AIF   C:&A1=0,.A
     0000 AD 00 80      +          LDA   I
     0003               +.A
     0003 18            +          CLC
     0004 6D 00 80      +          ADC   J
     0007               +          AIF   C:&A3=0,.B
     0007 8D 00 80      +          STA   K
     000A               +.B
0012 000A                          ADD     I,J,K
0013 000A                          ADD     K,J,I
     000A               +          ANOP
     000A               +          AIF   C:&A1=0,.A
     000A AD 00 80      +          LDA   K
     000D               +.A
     000D 18            +          CLC
     000E 6D 00 80      +          ADC   J
```

```
     0011                +          AIF   C:&A3=0,.B
     0011 8D 00 80       +          STA   I
     0014                +.B
0013 0014                           ADD     K,J,I
0014 0014 6B                        RTL
0015 0015                           END

Page 2   SAMPLE PROGRAM

0016 0000
0017 0000                TESTDAT    DATA
0018 0000 00 00          I          DS      2
0019 0002 00 00          J          DS      2
0020 0004 00 00          K          DS      2
0021 0006                           END

Page 3   SAMPLE PROGRAM


21 source lines
2 macros expanded
18 lines generated
```

With TRACE OFF the output from the same program looks like this:

```
0008 0000                           LIST    ON
0009 0000                           TITLE 'SAMPLE PROGRAM'
0010 0000                TEST       START
0011 0000                           USING TESTDAT
0012 0000                           ADD     I,J,K
     0000                +          ANOP
     0000 AD 00 80       +          LDA   I
     0003 18             +          CLC
     0004 6D 00 80       +          ADC   J
     0007 8D 00 80       +          STA   K
0013 000A                           ADD     K,J,I
     000A                +          ANOP
     000A AD 00 80       +          LDA   K
     000D 18             +          CLC
     000E 6D 00 80       +          ADC   J
     0011 8D 00 80       +          STA   I
0014 0014 6B                        RTL
0015 0015                           END

Page 2   SAMPLE PROGRAM

0016 0000
0017 0000                TESTDAT    DATA
0018 0000 00 00          I          DS      2
0019 0002 00 00          J          DS      2
0020 0004 00 00          K          DS      2
0021 0006                           END
```

```
Page 3    SAMPLE PROGRAM


21 source lines
2 macros expanded
18 lines generated
```

USING Using Data Segment

[label] USING *dataseg* *[comment]*

The USING directive makes local data segment labels visible to the code segment in which the USING directive appears.

dataseg The operand field contains the data segment label.

[label] Labels defined within the code segment take precedence over labels by the same name in data segments.

For example, the line USING TESTDAT makes the data segment TESTDAT available to the code segment in the program SAMPLE PROGRAM which follows.

```
            GEN      ON
            TITLE 'SAMPLE PROGRAM'
TEST        START
            USING TESTDAT
            ADD      I,J,K
            ADD      K,J,I
            RTL
            END

TESTDAT     DATA
I           DS       2
J           DS       2
K           DS       2
            END
```

Chapter 5

Using Macros in Assembly-Language Programs

This chapter describes how to use macros in your assembly-language programs. It shows you an assembler listing with macros ;tells you what is in the APW macro and equates directory, and tells you how to build your own subset of this directory. Finally, this chapter describes the macro directives you need to use the macros in the directory.

Macro Definition

A macro is a predefined sequence of instructions and directives that acts as a template. You code a single macro call in your source program which is expanded by the APW Assembler at assembly time. When the APW Assembler expands a macro call, it replaces the macro call statement with the contents of the macro definition, substituting actual values for its variables and parameters. You can create shorthand instructions for lengthy source text sequences by using the macros provided with your Apple IIGS or by creating your own macro definitions. Macros are stored in macro files, which are text files, created using the text editor in the same way that a source file is created. Groups of macro files are often referred to as macro libraries simply because they are a collection of macros stored in one place.

Note: APW provides a number of macro definitions contained in separate files in the `AINCLUDE` directory.

Note: Macro definition files are not legal in source files.

Macro Formats

APW macros are coded the same as any other assembly statement. Like instructions and directives, macros can consist of an optional label field, an operation code (which is always the macro name), an optional operand field, and an optional comment field.

[label] *OPERATION [operand1[,operand2[,operand3]]]* *[comment]*

The operation field is usually the only one required. Multiple operands, if they are coded, must be separated by commas. Any time two or more fields appear in one macro statement (an operation code, an operand field, and a comment field for example), they must be separated by at least one space.

Macro Expansions

Normally the text resulting from a macro expansion is not listed in the assembler output. However, inclusion of the GEN ON directive causes the expansion lines to be listed as well as the rest of the code. If you wrote a macro called ATIMES10 and stored it in a macro library called MYMACS the source code in your program might look like what follows.

```
            MCOPY       MYMACS
            GEN         ON

MAIN        START
            LDA         #5
            ATIMES10
            STA         FIFTY
            RTL

FIFTY       DS          2
            END
```

When your program is assembled and the ATIMES10 macro is expanded, your output file would look like this:

```
0001   0000
0002   0000
0003   0000                                    MCOPY   MYMACS
0004   0000                                    GEN     ON
0005   0000
0006   0000                              MAIN  START
0007   0000   A9 05 00                         LDA     #5
0008   0003                              ATIMES10
       0003   48                       +      PHA
       0004   0A                       +      ASL     A
       0005   0A                       +      ASL     A
       0006   0A                       +      ASL     A
       0007   18                       +      CLC
       0008   63 01                    +      ADC     1,S
       000A   63 01                    +      ADC     1,S
       000C   83 01                    +      STA     1,S
       000E   68                       +      PLA
0009   000F   8D 13 00                         STA     FIFTY
0010   0012   6B                              RTL
0011   0013
0012   0013   00 00                    FIFTY  DS      2
0013   0015                                    END
```

Each line of text generated by the macro expansion process is preceded by a plus (+) sign. Also notice that the macro-generated lines are not preceded by line numbers in the assembler listing; only the original source code lines (including the macro call itself) have

line numbers. Because code is actually generated for the macro lines, the program counter is incremented.

Assembler Directives Used With Macros

The MCOPY, MDROP, and MLOAD assembler directives described in this chapter are used to manipulate macro files.

MCOPY Copy Macro Library

[label] MCOPY *maclib* *[comment]*

maclib The name of the library to search for the macro operation code.

The name of the file is placed in a list of available macro libraries. If an operation code cannot be identified, the macro files in the list are loaded into the macro buffer, one at a time, and checked until the correct macro is found. The search begins with the macro file in memory, proceeds to the first file in the list of macro files, and continues through to the last file in the list, in the order that the respective MCOPY directives were encountered (skipping the one that was originally in memory). If no macro with a corresponding name is found, an error is generated.

No more than four macro libraries can be active at any one time. Macro libraries cannot contain COPY or APPEND directives.

MDROP Drop Macro Library

[label] MDROP *maclib* *[comment]*

maclib The name of the library to remove from the list of macro libraries.

This directive removes the name of the library in the operand from the list of macro libraries. If the macro library is active at the time the MDROP directive is encountered, it remains active and is searched for macros until a search is made which loads a different library, or until an MLOAD directive is used.

MLOAD Load Macro Library

[label] MLOAD *maclib* *[comment]*

maclib The name of the library to load into the list of macro libraries.

The list of active macro libraries is checked. If the specified file is not in the list, it is placed in the list and made active, then loaded. If the filename is already in the list, it is loaded into the macro library buffer.

Macro and Equate Directory

The APW directory, AINCLUDE is one of the files on the APW disk. AINCLUDE contains predefined macros and equates. Macros have already been described; equates define error codes and offsets into structures for the Apple IIGS Toolbox.

Predefined Macro Files

The AINCLUDE directory contains the macro files listed in Table 5-1. These macro files enable you to access the Apple IIGS Toolbox routines and ProDOS. They are all described in the *Apple IIGS Toolbox Reference,* Volume 1 and Volume 2, except for M16.UTIL. The file M16.UTIL contains macros that let you move bytes on to and off the stack, do mathematical operations, and write strings to output devices. These macros are described later in this chapter. To use any of the macro files from the directory in your program, you need to include an MCOPY directive in your source code with the name of the the file. The assembler searches the directory for the file that you have named. When the macro file is found, its text is included in your program source code without your having to do anything except provide the name.

Table 5-1. Predefined Macro Files in the AINCLUDE Directory

```
M16.ADB
M16.CONTROL
M16.DESK
M16.DIALOG
M16.EVENT
M16.FONT
M16.INTMATH
M16.LINEEDIT
M16.LIST
M16.LOADER
M16.LOCATOR
M16.MEMORY
M16.MENU
M16.MISCTOOL
M16.NOTESYN
M16.PRINT
M16.PRODOS
M16.QDAUX
M16.QUICKDRAW
M16.SANE
M16.SCHEDULER
M16.SCRAP
M16.SHELL
M16.SOUND
M16.STDFILE
M16.TEXTTOOL
```

```
M16.WINDOW
M16.UTIL
```

Predefined Equates

The AINCLUDE directory contains the equate files listed in Table 5-2. The ASM65816 Equates define error returns and offsets into structures for the Apple IIGS Toolbox. These files are documented in the *Apple IIGS Toolbox Reference*, Volume 1 and Volume 2.

Table 5-2. Equate Files in the AINCLUDE Directory

```
E16.ADB
E16.CONTROL
E16.DIALOG
E16.DESK
E16.EVENT
E16.FONT
E16.INTMATH
E16.LINEEDIT
E16.LIST
E16.LOADER
E16.LOCATOR
E16.MEMORY
E16.MENU
E16.MISCTOOL
E16.NOTESYN
E16.PRINT
E16.PRODOS
E16.QUICKDRAW
E16.SANE
E16.SCRAP
E16.SHELL
E16.SOUND
E16.STDFILE
E16.TEXTTOOL
E16.WINDOW
```

M16.UTIL Macros

The macros described here are all stored in the M16.UTIL file of the AINCLUDE directory. The macros in M16.UTIL can be divided into the following functional groups: macros that remove bytes from the stack, those that add bytes to the stack, macros that perform mathematical operations, those that define storage or formats, environmental macros, and text tool macros.

Macros that remove bytes from the stack include

PULLLONG	pull the top four bytes from the stack
PULLWORD	pull the top two bytes from the stack
PULL3	pull the top three bytes from the stack

PULL1	pull the top byte from the stack

Macros that add bytes to the stack include

PUSH1	push one byte on the stack.
PUSHLONG	push long (4 bytes) onto stack
PUSH3	push three bytes onto stack
PUSHWORD	push two bytes onto stack

Macros used to perform mathematical operations include

ADD	add two-byte integers
ADD4	add together four-byte integers
ASL4	arithmetic left-shift a four-byte number
DEC4	decrement a four-byte number
INC4	increment a four-byte number by one
SUB	subtract two-byte integers
SUB4	subtract four-byte integers
LSR4	logical right shift four bytes

Definition macros include

STR	define a string in Pascal format
DP	define storage for pointers

Environmental macros include

NATIVE	turn on native mode.
EMULATION	turn on emulation mode
LONG	set memory and registers to 16 bits
LONGM	set memory and A register to 16 bits
LONGX	set X and Y registers to 16 bits
SHORT	set memory and registers to 8 bits
SHORTM	set memory and A register to 8 bits

Text tool macros include

WRITESTR	write string to standard output device
WRITELN	write string to standard output device with line terminator
WRITECH	write character to standard output device
READCH	read a character from standard input device

The macros that follow are listed alphabetically. If the same macro can be used with either zero, one, two, or three parameters, each form is shown. These macros are designed to work in full native

mode except where otherwise noted. Also in many of the macros, a parameter may include addressing prefixes such as:

```
">", "<", "|"
```

The parameters in some macros may be expressions.

ADD

This macro adds numbers. ADD can be used with two or three parameters.

```
[label] ADD   address1,address2

[label] ADD   address1,address2,address3
```

If the first form of the macro is used, the two bytes in *address1* are loaded into the A register and the two bytes in *address2* are added to the A register using the instructions:

```
LDA   address1
CLC
ADC   address2
```

If the second form of the macro is used, the result in the A register is stored in *address3*. If the *address1* parameter is not specified, the current contents of the A register is used for the addition.

Note that either *address1* or *address2* or both may be specified as constants. For example, the following macro adds together the numbers 5 and 10 and stores the result (15) in locations LOC and LOC+1:

```
ADD   #5,#10,LOC
```

ADD4

This macro adds two four-byte numbers together and stores the result in the third parameter. The ADD4 macro may be used with either the three parameters shown, or with the first parameter unspecified. If the first parameter is not specified, the current contents of the A register as a two-byte quantity is used for the addition.

```
[label] ADD4    address1,address2,address3
```

The four bytes in *address1* are added to the four bytes in *address2* and the result is stored in *address3* using these instructions:

```
LDA   address1
CLC
```

```
ADC   address2
STA   address3
LDA   address1+2
ADC   address2+2
STA   address3+2
```

Note: After adding the lower two bytes of *address1* and *address2*, the carry is automatically added to the addition calculation of the higher two bytes.

If the *address1* parameter is not specified, the current contents of the A register as a *two-byte quantity* is used for the addition. You should also be aware that when the first parameter is unspecified, the leading comma is still required. For example

```
ADD4  ,LOC2,LOC3
```

Either *address1*, or *address2*, or both, may be specified as constants. For example, the following code adds together the numbers $00010000 and $00000200 and stores the answer ($00010200) in locations LOC through LOC+3:

```
ADD4  #$10000,#$200,LOC
```

ASL4

This macro performs an arithmetic left-shift on a 4 byte number. ASL4 has two different forms:

```
[label] ASL4 address,count
[label] ASL4 address
```

If the first form of the macro is used, the four-byte number starting at location *address* is shifted left *count* number of times using the following instructions.

```
      LDA   address+2
      LDX   count
~A    ASL   A
      ASL   address
      ADC   #0
      DEX
      BNE   ~A
      STA   address+2
```

Both the A and X register are used in the macro. The ADC #0 instruction is needed to add the carry that resulted from shifting the lower two-bytes of the number to the higher two-bytes. Note that *count* could be in the form of *#constant*. If the second form of the macro is used, the count is assumed to already be in the X register.

DEC4

This macro decrements a four-byte number. `DEC4` must be used with the *address* parameter as shown here.

```
[label] DEC4 address
```

The four bytes starting with *address* are decremented by 1 using these instructions:

```
      DEC   address
      BPL   ~A
      DEC   address +2
~A    ANOP
```

DP

This macro defines storage for four-byte pointers. `DP` must be used with its parameter as shown here.

```
[label] DP pointer
```

The macro generates the following instruction.

```
      DC    I4 pointer
```

To generate a table of four-byte pointers between the locations `LOC1` and `LOC2`, you could use the following code sequence:

```
            LOC1    DS 2
            LOC2    DS 2
            ...
            ...
TABLE       DP      LOC1
            DP      LOC2
```

EMULATION

This macro generates code to put the Apple IIGS's 65C816 processor in the mode in which it functions like an 8-bit Apple II 6502 processor in all respects except clock speed. This macro also generates assembler directives to put the APW Assembler into the same mode. `EMULATION` must be specified without parameters:

```
[label]     EMULATION
```

The macro generates the following code:

```
SEC
XCE
LONGA OFF
LONGI OFF
```

INC4

This macro increments a four byte number. `INC4` must be used with the *address* parameter shown here.

[*label*] `INC4` *address*

The four bytes starting with address are incremented by one using these instructions:

```
        INC   address
        BNE   ~A
        INC   address+2
~A      ANOP
```

LONG

This macro generates code to put the processor into both16 bit memory/accumulator and 16-bit index register mode. It also generates assembler directives to put the APW Assembler into the same mode. `LONG` must be specified without parameters:

[*label*] `LONG`

The macro generates the following code:

```
REP     #%00110000
LONGA   ON
LONGI   ON
```

LONGM

This macro generates code to put the processor into 16-bit memory and accumulator mode. It also generates assembler directives to put the APW Assembler into the same mode. `LONGM` must be specified without parameters:

[*label*] `LONGM`

The macro generates the following code:

```
REP     #%00100000
LONGA   ON
```

LONGX

This macro generates code to put the processor into 16-bit index register mode. It also generates assembler directives to put the APW Assembler into the same mode. LONGX must be specified without parameters:

```
[label] LONGX
```

The macro generates the following code:

```
REP     #%00010000
LONGI   ON
```

LSR4

This macro performs a logical right-shift on a four byte number. LSR4 can be used with either one or two parameters:

```
[label] LSR4 address,count
```

```
[label] LSR4 address
```

If the first form of the macro is used, the 4 byte number starting at location *address* is shifted right *count* number of times using the following instructions:

```
        LDA   count
        EOR   $FFFF
        CLC
        ADC   #1
        TAX
        LDA   address
~A      LSR   A
        LSR   address+2
        BCC   ~B
        ORA   #$8000
~B      INX
        BNE   ~A
        STA   address
```

Both the A and X registers are used in the macro. Note that *count* could be in the form of #*const*. If the second form of the macro is used, the *complement* of the count is assumed to already be in the X register.

NATIVE

This macro generates code to put the 65C816 processor into the 16-bit operating state. It also generates assembler directives to put the APW Assembler into the same mode. NATIVE can be used in either of the two forms shown:

[*label*] NATIVE

[*label*] NATIVE *mode*

If the first form of the macro is used, the processor is set to full native mode using the instructions:

```
CLC
XCE
REP     #%00110000
LONGA   ON
LONGI   ON
```

If the second form of the macro is used, *mode* specifies a processor partial native mode defined by how memory, the accumulator, and registers are accessed. Each of these modes is described under the name of the name used in the parameter. The following partial native modes can be specified:

```
NATIVE LONG
NATIVE LONGM
NATIVE LONGX
NATIVE SHORT
NATIVE SHORTM
NATIVE SHORTX
```

PULLLONG

This macro pulls the top four bytes from stack using the A register and optionally stores them in memory. The PULLLONG macro has four forms:

[*label*] PULLLONG

[*label*] PULLLONG *address*

[*label*] PULLLONG *address1, address2*

[*label*] PULLLONG [DirectPage],*offset*

If the first form of the macro is used, the first two bytes are discarded and the last two bytes remain in the A register. If the second form is used, the four bytes are stored using the instructions

```
STA   address
STA   address+2
```

Because the parameter *address* can actually include all direct addressing modes that do not require an index register, any of the following forms can be used:

```
STA   abs       Absolute
STA   dp        Direct Page
STA   long      Absolute Long
```

If the third form of the macro is used, the four bytes of data are stored at two different locations: *address1*, *address2*. The macro call

```
PULLLONG address1,address2
```

generates the code

```
PLA
STA   address1
STA   address2
PLA
STA   address1+2
STA   address2+2
```

The fourth form is a special case for the Direct Page Indirect Indexed Long addressing mode. This form stores the data via indirect Y addressing, using the instruction `STA [dp],Y`. Y is loaded with the value *offset*. If no parameters are coded, the high-order word remains in the accumulator and the low-order word is lost.

For example

```
PULLLONG [100],5
```

generates the code

```
LDY   #5
PLA
STA   [100],Y
LDY   #7
PLA
STA   [100],Y
```

Important: The offset in the macro call should not be preceded by a #.

PULL1

This macro pulls the top byte from the stack using the A register and optionally stores it in memory. `PULL1` has three different forms:

[*label*] PULL1

[*label*] PULL1 *address*

[*label*] PULL1 *address, register*

All forms of the macro first generate the instructions

```
SEP   #%00100000      set memory and A register to 8-bit mode
PLA
```

The last instruction of each macro form restores memory and the A register to 16-bit mode via the instruction

```
REP   #%00100000
```

If the first form of the macro is used, the byte being pulled from the stack remains in the A register. If the second form is used, the byte is stored in memory using the instruction

STA *address*

Note that the parameter *address* can actually include all addressing modes that do not require an index register. Any of the following forms can be used:

STA	*abs*	Absolute
STA	*dp*	Direct Page
STA	*long*	Absolute Long
STA	*(dp)*	Direct Page Indirect
STA	[*dp*]	Direct Page Indirect Long
STA	*(dp,X)*	Direct Page Indexed Indirect

If the third form of the macro is used, the *register* parameter specifies the index register used for the STA instruction, and therefore most of the remaining address modes can be used.

STA	*abs*,X	Absolute Indexed by X
STA	*abs*,Y	Absolute Indexed by Y
STA	*dp*,X	Direct Page Indexed by X
STA	*dp*,Y	Direct Page Indexed by Y
STA	*(dp)*,Y	Direct Page Indirect Indexed
STA	[*dp*],Y	Direct Page Indirect Indexed Long
STA	*dp*,S	Stack Relative
STA	*(dp,S)*,Y	Stack Relative Indirect Indexed

Note: Because the parameters in the Direct Page Indexed Indirect and Stack Relative Indirect Indexed addressing modes are within parentheses, the assembler treats them as though they were a single parameter.

PULL3

This macro pulls the top three bytes from the stack using the A register and optionally stores them in memory. The PULL3 macro has only one form:

[*label*] PULL3 *address*

The macro generates the instructions

```
SEP   #%00100000      set memory and A register to 8-bit mode
PLA
STA   address
REP   #%00100000      restore memory and A register to 16-bit mode
STA   address+1
```

Because *address* can actually include all addressing modes that do not require an index register, this parameter can take any of the following forms:

```
STA   abs             Absolute
STA   dp              Direct Page
STA   long            Absolute Long
```

PULLWORD

This macro pulls the top two bytes from the stack using the A register and optionally stores them in memory. PULLWORD has three different forms

[*label*] PULLWORD

[*label*] PULLWORD *address*

[*label*] PULLWORD *address, register*

If the first form is used, the two bytes remain in the A register. If the second form is used, the two bytes are stored using the instruction

STA *address*

Because the *address* parameter can actually include all addressing modes that do not require an index register (you may need to prefix the address with a <, >, or |), it can take any of the following forms:

```
STA   abs       Absolute
STA   dp        Direct Page
STA   long      Absolute Long
STA   (dp)      Direct Page Indirect
STA   [dp]      Direct Page Indirect Long
STA   (dp,X)    Direct Page Indexed Indirect
```

If the third form of the macro is used, the *register* parameter specifies the index register used for the STA instruction, and therefore most of the remaining address modes can be used:

STA	*abs*,X	Absolute Indexed by X
STA	*abs*,Y	Absolute Indexed by Y
STA	*dp*,X	Direct Page Indexed by X
STA	*dp*,Y	Direct Page Indexed by Y
STA	*(dp)*,Y	Direct Page Indirect Indexed
STA	[*dp*],Y	Direct Page Indirect Indexed
STA	*dp*,S	Stack Relative
STA	*(dp,S)*,Y	Stack Relative Indirect Indexed

Note: Because the parameters in the Direct Page Indexed Indirect and Stack Relative Indirect Indexed addressing modes are within parentheses, the assembler treats them as though they were a single parameter.

PUSHLONG

This macro pushes four bytes on the stack. The PUSHLONG macro has four different forms:

[*label*] PUSHLONG #*constant*

[*label*] PUSHLONG *address*

[*label*] PUSHLONG *address*, *register*

[*label*] PUSHLONG [DirectPage],*offset*

If the first form of the macro is used, a four byte constant is pushed on the stack using the instructions

PEA	*constant*/-16	push-high order two bytes
PEA	*constant*	push low-order two bytes

Note that the constant is preceded by a # in the macro call.

If the second form is used, the four bytes are pushed on the stack using the instructions

```
LDA   address+2
PHA
LDA   address
PHA
```

Note that the parameter *address* can actually include all direct addressing modes that do not require an index register:

LDA	*abs*	Absolute

LDA	*dp*	Direct Page
LDA	*long*	Absolute Long

If the third form is used, the *register* parameter specifies the index register used for the LDA instruction and therefore most of the remaining addressing modes can be used:

LDA	*abs*,X	Absolute Indexed by X
LDA	*abs*,Y	Absolute Indexed by Y
LDA	*dp*,X	Direct Page Indexed by X
LDA	*dp*,Y	Direct Page Indexed by Y
LDA	*(dp)*,Y	Direct Page Indirect Indexed
LDA	[*dp*],Y	Direct Page Indirect Indexed
LDA	*dp*,S	Stack Relative
LDA	*(dp,S)*,Y	Stack Relative Indirect Indexed

The fourth form is a special case for the Direct Page Indirect Indexed Long addressing mode. For example the macro call

```
PUSHLONG [100],5
```

generates the code

```
LDY   #5
LDA   [100],Y
PHA
LDY   #7
LDA   [100],Y
PHA
```

Important: The offset in the macro call should *not* be preceded by a #.

Note: Because the parameters in the Stack Relative Indirect Indexed addressing mode are within parentheses, the assembler treats them as though they were a single parameter.

PUSH1

This macro pushes one byte onto the stack. The PUSH1 macro has four different forms:

[*label*] PUSH1

[*label*] PUSH1 #*constant*

[*label*] PUSH1 *address*

[*label*] PUSH1 *address, register*

In all forms, the macro first generates an instruction to set memory and the A register to 8-bit mode.

```
SEP   #%00100000
```

The last instruction of all forms restores memory and the A register to 16-bit mode.

```
REP   #%00100000
```

The first form of `PUSH1` pushes the byte from the A register. The second form pushes the value of a constant onto the stack. If the fourth form of the macro is used, the *register* parameter specifies the X register.

PUSH3

This macro pushes three bytes on the stack. The `PUSH3` macro has three different forms:

[*label*] `PUSH3` *#constant*

[*label*] `PUSH3` *address*

[*label*] `PUSH3` *address, register*

If the first form of the macro is used, a three byte constant is pushed on the stack using the instructions

`LDA`	*#constant*`|-8`	high-order two bytes
`PHA`		push on stack
`PHB`		push an extra byte on stack
`LDA`	*#constant*	low order two bytes
`STA`	`1,S`	store directly to stack

This code stores one of the three bytes on the stack twice, but the end result is correct. This technique takes less code than going into 8-bit mode to push one- byte and then going back into 16-bit mode to push the other two bytes. Also note that the constant is preceded by a # in the macro call.

If the second form is used, the three bytes are pushed on the stack using the instructions

`LDA`	*address*`+1`	high-order two bytes
`PHA`		push on stack
`PHB`		push an extra byte on stack
`LDA`	*address*	low-order two bytes
`STA`	`1,S`	store directly to stack

The parameter *address* can actually include all direct addressing modes that do not require an index register:

`LDA`	*abs*	Absolute

LDA	*dp*	Direct Page
LDA	*long*	Absolute Long

If the third form is used, the *register* parameter specifies the index register used for the LDA instruction and therefore some of the remaining addressing modes can be used:

LDA	*abs*, X	Absolute Indexed by X
LDA	*abs*, Y	Absolute Indexed by Y
LDA	*dp*, X	Direct Page Indexed by X
LDA	*dp*, Y	Direct Page Indexed by Y

PUSHWORD

This macro pushes two bytes on the stack. The PUSHWORD macro has four different forms:

[*label*] PUSHWORD

[*label*] PUSHWORD #*constant*

[*label*] PUSHWORD *address*

[*label*] PUSHWORD *address*, *register*

If the first form of the macro is used, the two bytes in the A register are pushed on the stack using the instruction

```
PHA
```

If the second form is used, a two-byte constant is pushed on the stack using the instruction

```
PEA   constant
```

If you use this form, be sure to preface the constant with a # in the macro call.

The third form of the macro loads two bytes into the A register using the instruction

```
LDA   address
```

and then pushes on the stack with the instruction

```
PHA
```

The parameter *address* can actually include all addressing modes that do not require an index register. For example:

LDA	*abs*	Absolute
LDA	*dp*	Direct Page

LDA	*long*	Absolute Long
LDA	(*dp*)	Direct Page Indirect
LDA	[*dp*]	Direct Page Indirect Long
LDA	(*dp*,X)	Direct Page Indexed Indirect

The fourth form of the macro uses the *register* parameter to specify the index register for the LDA instruction. This means that most of the remaining addressing modes can be used:

LDA	*abs*, X	Absolute Indexed by X
LDA	*abs*,Y	Absolute Indexed by Y
LDA	*dp*,X	Direct Page Indexed by X
LDA	*dp*,Y	Direct Page Indexed by Y
LDA	(*dp*),Y	Direct Page Indirect Indexed
LDA	[*dp*],Y	Direct Page Indirect Indexed Long
LDA	*dp*,S	Stack Relative
LDA	(*dp*,S),Y	Stack Relative Indirect Indexed

READCH

This macro reads a character from the standard input device (usually the keyboard) and either stores it in the A register, or in the memory location specified by *address*. There are three forms that it can take:

[*label*] READCH

[*label*] READCH *address*

[*label*] READCH *address, register*

If the first form of the macro is used, the character will remain in the A register and not be stored. The second form of the macro calls the Text Tool function *ReadChar* and reads the character into the A register. The A register is stored in location *address* .

```
PEA     0
PEA     1
LDX     #$220C
JSL     $E10000
PLA
STA     address
```

The third form of the macro uses the instruction

STA *address,register*

instead of the STA *address* instruction used in the second form. The second and third forms of the macro can get a character using any of the available addressing modes that the instruction STA supports.

SHORT

This macro generates code to put the processor into both 8-bit memory/accumulator and 8 bit index register mode. It also generates assembler directives to put the APW Assembler into the same mode. SHORT can be specified only in the form shown:

```
[label] SHORT
```

The macro generates the following code:

```
REP     #%00110000
LONGA   OFF
LONGI   OFF
```

SHORTM

This macro generates code to put the processor into 8-bit memory and accumulator mode. It also generates assembler directives to put the APW Assembler into the same mode. SHORTM can only be used in the form shown:

```
[label] SHORTM
```

The macro generates the following code:

```
REP     #%00100000
LONGA   OFF
```

SHORTX

This macro generates code to put the processor into 8-bit index register mode. It also generates assembler directives to put the APW Assembler into the same mode. SHORTX cannot be specified with any parameters.

```
[label] SHORTX
```

The macro generates the following code:

```
REP     #%00010000
LONGI   OFF
```

STR

This macro converts a string into a Pascal type of format and stores it. The first byte of the string will contain the number of characters in the string. The second, and succeeding bytes, are the characters

in the string itself (one byte per character). The STR macro uses one parameter. Remember to use quotation marks if you have any spaces in your string.

[*label*] STR *string*

The macro generates the following instruction:

```
DC    I1'L:string',C'string'
```

For example

```
STR "ABCD"
```

generates the following five bytes:

```
4 'A' 'B' 'C' 'D'
```

SUB

This macro subtracts two byte numbers. SUB can be used with two or three parameters:

[*label*] SUB *address1,address2*

[*label*] SUB *address1,address2,address3*

If the first form of the macro is used, the two bytes in *address1* are loaded into the A register and the two bytes in *address2* are subtracted from the A register using the instructions

```
LDA   address1
SEC
DEC   address2
```

The result is stored in the A register.

If the second form of the macro is used, the result in the A register is stored in *address3*. If the *address1* parameter is not specified, the current contents of the A register are used for the subtraction. In this case, as with the ADD macro, a leading comma is required:

```
SUB   ,LOC2,LOC3
```

Either *address1*, or *address2*, or both, may be specified as constants. For example

```
SUB   #15,#10,LOC
```

subtracts the number 10 from the number 15 and stores the result (5) in locations LOC and LOC+1.

SUB4

This macro subtracts one four-byte number from another. SUB4 can only be used with either the three parameters shown, or with the first parameter unspecified. If the first parameter is not specified, the current contents of the A register as a two-byte quantity is used for the subtraction.

```
[label] SUB4      address1,address2,address3
```

The four bytes in *address2* are subtracted from the four bytes in *address1* and the result is stored in *address3* using the instructions

```
LDA   address1
SEC
DEC   address2
STA   address3
LDA   address1+2
DEC   address2+2
STA   address3+2
```

Important: After subtracting the lower two bytes of *address1* and *address2*, the carry is automatically subtracted from the subtraction calculation of the higher two bytes.

Either *address1*, or *address2*, or both, may be specified as constants or expressions. For example

```
SUB4 #$10000,#$200,LOC
```

subtracts the number $00000200 from the number $0010000 and stores the result ($0000FE00) in locations LOC through LOC+3.

WRITECH

This macro writes a character to the standard output device (usually the screen). It has four different forms:

```
[label] WRITECH
[label] WRITECH   address
[label] WRITECH   #'character'
[label] WRITECH   address, register
```

If the second form of the macro is used, the character in location *address* is written using the instructions

```
LDA     address
PHA
LDX     #$180C
JSL     $E10000
```

which call the Text Tool function *WriteChar* with the character. The third form of the macro uses the instruction

```
LDA     #'character'
```

instead of the instruction

```
LDA     address
```

The fourth form uses the instruction

```
LDA     address,register
```

Therefore the second and third forms of the macro can be used to get a character using *any* of the available addressing modes that the instruction LDA supports. The first form of the macro assumes that the character is already in the A register.

WRITELN

This macro writes a string followed by the line terminator (usually a carriage return) to the standard output device (usually the screen). The macro has three different forms:

```
[label] WRITELN  address

[label] WRITELN  #"string"

[label] WRITELN
```

If the first form of the macro is used, the string in location *address* is written using the instructions

```
PEA     address|-16
PEA     address
LDX     #$1A0C
JSL     $E10000
```

which call the Text Tool function *WriteLine* with the pointer to the string.

If the second form of the macro is used, the specified string is built in the macro and that address is used in the Text Tool call.

If the third form of the macro is used, only the line terminator is written.

WRITESTR

This macro writes a string to the standard output device (usually the screen). It can be used in three different forms:

```
[label] WRITESTR address

[label] WRITESTR #"string"

[label] WRITESTR
```

If the first form of the macro is used, the string in location *address* is written using the instructions

```
PEA     address|-16
PEA     address
LDX     #$1C0C
JSL     $E10000
```

which call the Text Tool function *WriteString* with the pointer to the string.

If the second form of the macro is used, the specified string is built in the macro and that address is used in the Text Tool call.

If the third form of the macro is used, the address of the string is assumed to be in the A and Y registers (A contains the lower two bytes).

Building Your Own Macro Library

Normally you use MacGen to build one macro file containing all of the macros that you use. This method not only saves you macro table space when you are assembling, but it will reduce your assembly time noticably. The assembler does not have to search through all of the unreferenced macros to find the ones that you are using. This file can include your own macros as well as APW macros. Call the MacGen utility to create your macro file by typing in the following command:

```
MACGEN [+C|-C] infile outfile macrofile1 [macrofile2...]
```

MacGen creates a custom macro file for your program by searching one or more macro libraries for the macros referenced in your program and placing the referenced macros in a single file.

`+C | -C` If you specify `+C` (the default value), all excess spaces and all comments are removed from the macro file to save space. If you use the `ON` option of the `GEN` directive to include expanded macros in your source listing, use the `-C` parameter of the MacGen utility to improve the readability of the listing. The same is true if you use the `ON` option of the `TRACE` directive to include conditional execution directives in your source file listing.

infile The full or partial pathname (including the filename) of your source file. MacGen scans this file for references to macros.

outfile The full or partial pathname (including the filename) of the macro file to be created by MacGen.

macrofile1 macrofile2 The full pathnames or partial pathnames (including the filenames) of the macro libraries to be searched. These are the libraries that will be searched for the macros referenced in *infile* . At least one library must be specified. Wildcard characters may be used in the filenames. If you use more than one filename, separate the names with one or more spaces.

MacGen scans *infile*, including all files referenced with `COPY` and `APPEND` directives, and builds a list of the macros referenced. If there are still unresolved references to macros, MacGen scans *macrofile1*, *macrofile2*, and so on. MacGen can handle macros that call other macros. If there are still references to macros that are unresolved after all of the macro files you specified in the command line have been scanned, MacGen lists the missing macros and prompts you for the name of another macro library. Press Return without a filename to terminate the process before all macros have been found. After all macros have been found (or you press Return to end the process), *outfile* is created.

Important: Before you assemble your program, make sure that the source code contains the directive `MCOPY` *outfile* so that the assembler will search *outfile* for your macros. If you do not supply *infile*, *outfile*, or *macrofile1* when you enter the `MACGEN` command, you will be prompted for the missing parameter. Remember that MacGen assumes that the first pathname is *infile*, the second pathname is *outfile*, and that the third and subsequent pathnames are *macrofiles*.

The following example scans the program file `/APW/TEST.ASM` for macro file names, searches the macro libraries `/LIB/MACROS` and `/APW/AINCLUDE` for the referenced files, and creates a new macro library called `/APW/MYMACROS`:

```
MACGEN /APW/TEST.ASM /APW/MYMACROS /LIB/MACROS /APW/AINCLUDE
```

Assembling a Program That Contains Library Macros

In order to assemble a program that contains library macros, do the following:

1. Write a source code program with the calls to the macros that you need and any parameters that they require.

2. Run your program through the MacGen utility to create a macro library unique to your program.

3. Add a MCOPY directive at the beginning of your program with the name of the output file created by MacGen.

4. Assemble and execute the program in the normal way.

Listing Options

There are two directives you can use to set specific options for your listing. These are TRACE and GEN. If you want to see the code generated by the macros, use the GEN directive at the beginning of your program. The TRACE ON directive instructs the APW Assembler to print all the lines that the APW Assembler processes. This can be useful when you are debugging macros.

Chapter 6

Writing Macros

The alternative to using the macros in the libraries described in Chapter 5 is to write your own macros. This chapter describes the macro and conditional assembly directives, symbolic parameters, and sequence symbols that you need to be able to write your own macro definitions. Although symbolic parameter substitution and conditional assemblies are valid outside of macros, these two features are so typically used within macro definition files that they are included in this chapter.

Macro Definition

A macro is a predefined sequence of instructions and directives that acts as a template. When the APW Assembler encounters an instruction in the operation code field that it does not recognize as a valid 65816 instruction mnemonic or assembler directive, it assumes that it is a macro call and searches the active macro definition files (as specified by MCOPY or MLOAD directives in your source code) for a macro definition that matches the unrecognized instruction. The assembler then does a macro expansion, substituting the program text found in the macro definition file for the macro call. When you write your own macro, you must save it in a macro file and then use an MCOPY directive in your source code to access it exactly as though it were one of the predefined macros provided with APW.

You go through the same steps to save the macros that you write as you do to create any other assembler source file: edit the macro, name it, and save it.

A simple macro definition looks like this:

```
        MACRO
&LAB    ATIMES10
&LAB    ASL     A
        PHA
        ASL     A
        ASL     A
        ADC     1,S
        PLX
        MEND
```

Each macro definition begins with a MACRO directive and ends with a MEND directive. The line immediately following the MACRO directive is the macro definition statement. It contains the macro name, in this case ATIMES10. The name may be any sequence of keyboard characters except a blank or an ampersand (&). The name may contain any number of characters.

There are two other directives in addition to MACRO and MEND used to define macros, they are MNOTE and MEXIT. MNOTE is used to code an error message into a macro; MEXIT is used to exit from the macro before the end of the definition code is reached. With the

exception of MNOTE, these directives are legal only within macro definition files. If you try to use them in source code, they will create an error.

Conditional Assembly Directives

Conditional assemblies allow source code lines in a program to be included, excluded, or repeated, depending upon a given condition.

The conditional assembly directives AGO and AIF are branch instructions that require destinations. The destinations are provided by sequence symbols, represented in this text as *.symbol*. A sequence symbol is a period (.), followed by a label. Comments may follow the label after at least one space. Sequence symbols are not printed in the assembler listing unless TRACE ON is in effect.

One use of the conditional assembly is to push macro parameters onto the stack before jumping to a routine, if a set of conditions are met. If the conditions are not met, a branch is made around the code that puts the parameters on the stack.

```
            MACRO
&LAB        THEMACROS    &VAL
&LAB        ANOP
            AIF          C:&VAL=0,.SKIP
            LDA          &VAL
            PHA
.SKIP
            JMP          [!0]
            MEND
```

The directive that determines whether or not the branch will be made is the AIF (Assember IF) directive. The expression C:&VAL=0 determines whether or not the symbolic parameter &VAL has been defined. The C: preceding the &VAL is a count attribute; it returns a value of 0 if the parameter is undefined. The equal sign works as a comparison operator, returning 0 if the comparison is not true, or 1 if it is true. If the expression is nonzero, the macro processor skips the input lines up to the target label defined after the comma (.SKIP); otherwise it continues processing on the next line. The following source program and expanded assembler listing illustrates the two ways in which the macro THEMACROS can be invoked.

```
0001  0000
0002  0000                    MCOPY    THEMACROS
0003  0000                    GEN      ON
0004  0000
0005  0000               MAIN START
0006  0000
0007  0000
0008  0000             D0VALUE ENTRY
0009  0000 AD 0E 00           LDA   VALUE Initialize to 5.
0010  0003 48                 PHA         Pass on the stack.
0011  0004                    THEMACROS
      0004               +    ANOP
      0004 DC 00 00      +    JMP       [!0]
```

```
0012  0007
0013  0007                D05     ENTRY
0014  0007                        THEMACROS #5
      0007              +         ANOP
      0003 A9 05 00     +         LDA       #5
      000A 48           +         PHA
      000B DC 00 00     +         JMP       [!0]
0015  000E
0016  000E 00 00        VALUE     DS  2
0017  0010                        END
```

This routine contains two entry points which are implemented via the two possible expansions of the THEMACROS macro. If the entry point DOVALUE is used, whatever is stored in VALUE is pushed on the stack and the macro call is made without an operand. If you use the entry point at D05, the macro call is made with an operand. An immediate 5 is always pushed on the stack before the call is made.

Symbolic Parameters

Symbolic parameters are variables that can be assigned and reassigned values. Symbolic parameters are coded as an ampersand (&), followed by a name. The name has the same syntax conventions as a label. Symbolic parameters are represented in this text as *&spar*. There are three types of symbolic parameters: A(arithmetic), B (Boolean), and C (character). Arithmetic and Boolean symbolic parameters are initialized to 0. Arithmetic symbolic parameters can contain any 32-bit signed value; Boolean symbolic parameters can contain any value between 0 and 255. Zero is treated as false and any other value as true when used in a logical expression.

Symbolic parameters are assigned values by the SETA, SETB, or SETC directives. Use the COUNT attribute described later in this chapter to determine whether or not a symbolic parameter is defined.

Set Symbols

Set symbols are like assignment statements because they are used to assign values to symbolic parameters. Set symbols are different from assignment statements because set symbols come in types and the type of the set symbol used to make an assignment must correspond with the declared type of the symbolic parameter. An arithmetic symbolic parameter can only be assigned a value by the set symbol directive SETA; Boolean symbolic parameters by SETB, and character symbolic parameters by SETC.

For example, if you attempt to assign the string: 'Here''s a quoted string' using SETA instead of SETC, the assembler will assume that the operand is a constant expression. The result that will be assigned to &STRING will be a 32-bit signed number formed by evaluating the first four characters as numbers.

```
&STRING    SETA         'Here''s a quoted string'
```

Using Symbolic Parameters

Macros become really useful when they contain symbolic parameter substitutions. Symbolic parameter substitution allows the macro definition text to include placeholders that are replaced by actual values when the macro expansion takes place. The actual values to be used are specified in the operand field of the macro call statement in the source file. An example of a macro definition which includes symbolic parameters is the macro ADDLONG. This macro adds together two 32-bit numbers and stores the result at the address of the first number. The name of the symbolic parameter following the ampersand has the same syntax as an APW Assembler label.

```
                    MACRO
&LAB                ADDLONG     &NUMB1,&NUMB2
&LAB                CLC
                    LDA     &NUMB1
                    ADC     &NUMB2
                    STA     &NUMB1
                    LDA     &NUMB1+2
                    ADC     &NUMB2+2
                    STA     &NUMB1+2
                    MEND
```

The symbolic parameters of the macro definition act as dummy values. They are replaced by actual values during the expansion process. The following examples show the input source file that invokes the macro and the APW Assembler output listing, which shows the code actually generated.

```
            MCOPY   MYMACS
            GEN     ON
MAIN        START
DOADD       ADDLONG BIGNUM,BIGNUM2
            RTL
BIGNUM      DC      I4'1234567'
BIGNUM2     DC      I4'2345678'
            END
```

The parameters are separated by a comma in the operand field. There should be no intervening spaces between the operand values. The output listing shows only the generated macro code if you use the GEN ON option.

```
0001 0000
0002 0000
0003 0000                               MCOPY   MYMACS
0004 0000                               GEN     ON
0005 0000
0006 0000                       MAIN    START
0007 0000
0008 0000                       DOADD   ADDLONG BIGNUM,BIGNUM2
 0000       18             +    DOADD   CLC
 0001       AD 14 00       +    LDA     BIGNUM
 0004       6D 18 00       +    ADC     BIGNUM2
 0007       8D 14 00       +    STA     BIGNUM
```

```
 000A        AD 15 00    +   LDA     BIGNUM+2
 000D        6D 19 00    +   ADC     BIGNUM2+2
 0010        8D 14 00    +   STA     BIGNUM+2
0009 0013
0010 0013   6B                      RTL
0011 0014
0012 0014   87 D6 12 00             BIGNUM  DC  I4'1234567'
0013 0018   CE CA 23 00             BIGNUM2 DC  I4'2345678'
0014 001C
0015 001C                           END
```

Every occurrence of the symbolic parameters &NUM1 and &NUM2 is replaced in the expansion by the values found in the corresponding positions in the operand field of the macro call statement in the source file. In this case, the values were the labels of the memory locations containing the two numbers to be added. Notice that another symbolic parameter is included in the definition: &LAB. This parameter is coded to allow the macro call statement to receive a label, which is then used to put a label on one of the model statements. In other words, macro calls themselves cannot be labeled, but they can be used to give a value to a symbolic parameter in the label field of the macro definition statement if one is coded. This is necessary if the macro call is to be the target of a jump or call instruction. In the example, the symbol DOADD was coded in the label field of the macro call statement; in the expansion, the first model statement received this value as a label.

Symbolic Parameter Substitution

Symbolic parameter substitution takes place in two steps. In the first step, the string in the label field is skipped, and substitution begins with the operation field. If the resulting instruction in the operation field is not &SETA, &SETB, &SETC, AMID, AINPUT, or ASEARCH when a symbolic parameter is found in the label field, it too is replaced with its value.

Symbolic parameters are automatically concatenated with the text following them. However, it is often necessary to indicate where the symbolic parameter name ends and the surrounding text begins. The period, or dot operator, can be used to indicate that the end of symbolic parameter has been reached. In this case, the dot operator itself is not included in the text expansion and the text that follows is concatenated with the expanded symbolic parameter. Where a symbolic parameter name is followed immediately by an alphanumeric character, or if a subscript is followed by a mathematical symbol or expression, the dot operator is required.

When you use the dot operator in a logical expression, you must code the dot in addition to the period in the expression. This means that the expression

```
&LOGIC.AND.&LOGIC
```

will expand to `valueand.value` where &LOGIC contains `value`. The correct way to logically AND two symbolic parameters is

```
&LOGIC..AND.&LOGIC
```

Symbolic Parameter Scope

Symbolic parameters may be defined either for the current macro expansion or code segment, or for the entire subroutine. Defining symbolic parameters whose scope is the entire subroutine allows macros to communicate with each other. Symbolic parameters valid only within a macro or the code segment in which they are defined are called local symbolic parameters; those valid throughout the subroutine are called global symbolic parameters. The scope of a global symbolic parameter is therefore equivalent to the scope of a local assembler label. Using global symbolic parameters lets you create macros that pass information to one another.

Assigning values to Symbolic Parameters

The APW Assembler assigns values to symbolic parameters in different ways depending upon the kind of symbolic parameter.

Positional Parameters

In the examples so far, symbolic parameters have received their values from the macro call statement in the source program that invokes the macro. These implicitly defined symbolic parameters are called positional parameters because the values on the macro call line are matched with the symbolic parameters in the macro definition line by their respective positions. When assigning actual values to positional parameters, the APW Assembler counts the commas between parameters on the macro call line; if no value is to be given to a parameter, the right number of commas must be included anyway:

```
MYMACRO A,,C
```

The first positional parameter will receive the value A, the second no value, and the third, C.

In addition to the positional parameters found in the operand field of the macro call, if a label is coded in the macro call and macro definition lines, it too is a positional parameter.

Keyword Parameters

A keyword parameter is another way to reference a symbolic parameter defined in the operand field of a macro statement. The name of the macro is entered, followed by a space. The name of the first symbolic parameter is entered, followed by an equal sign and the value to be assigned to this parameter. Additional symbolic parameters are separated by commas.

To invoke the ADDLONG macro by using keyword parameters, you would enter

```
ADDLONG &NUMB1=BIGNUM,&NUMB2=BIGNUM2
```

When only keyword parameter substitution is used, the order of the parameters is not important. Positional parameter rules regarding commas and spaces apply if positional parameters are intermixed with keywords; in other words, keyword parameters take up a

space and are counted for determining positions when they are mixed with positional parameters.

Character-Type Symbolic Parameters

Character-type symbolic parameters are initialized to null character strings. Remember that character strings have both value and length. The length may be up to 255 characters. The null string has a length of 0.

Subscripted Symbolic Parameters

Symbolic parameters may be subscripted with up to 255 elements. In the case of explicitly defined symbolic parameters, the maximum number of subscripts must have been declared when the symbolic parameter was defined. Only a single subscript is allowed.

Symbolic parameters defined in the macro definition statement are subscripted by including the subscripted variables in parentheses on the macro call line.

For example, if a macro call statement contained the following phrase in the operand field,

```
&SUB=(ALPHA,,GAMMA)
```

the keyword parameter `&SUB` for the given expansion would have three subscripts allowed. The initial value of each element would be

```
&SUB(1) 'ALPHA'
&SUB(2) null string
&SUB(3) 'GAMMA'
```

The most effective way to use subscripted parameters is to code the macro to detect the number of subscripts allowed (using the count attribute) and then take an appropriate action via conditional assembly directives.

One symbolic parameter may be used as a subscript for another symbolic parameter; however, it cannot itself be a subscripted symbolic parameter, nor may symbolic parameters or literal strings be concatenated to form the subscript.

These examples illustrate how previously defined subscripted symbolic parameters are set and used.

```
&AR(4)    SETA   16
&AR(&NUM) SETA   &AR(4)
```

`&AR(4)` and `&AR(&NUM)` are both set to 16.

The following example assumes that four symbolic parameters have been defined. The maximum subscripts for each are shown with the symbolic parameter name. Next is the type, followed by the value. Subscripted symbolic parameters have their values listed on successive lines.

```
Name        Type    Value

&ART        A       $FE
&BIN(2)     B       1 (true)
                    0 (false)
&CHAR       C       'LABEL'
&CHAR2(3)   C       'STRING1'
                    ''  (null string)
                    'A'
```

The sample code that follows shows the instructions entered in a macro. Following it are the instructions as expanded by the macro processor.

Macro file:

```
&CHAR           LDA  &CHAR2(1)
                STA  &CHAR.&BIN(2)
                LDA  #&ART
                BEQ  L&BIN(1)
                LDA  LB&CHAR2(2)
L&BIN(1)        STA  EQ&BIN(2)
LD&CHAR2(3)     #1
```

Expanded instructions look like this:

```
LABEL           LDA  STRING1
                STA  LABEL0
                LDA  #254
                BEQ  L1
                LDA  LB
L1              STA  EQ0
LDA             #1
```

Note that the null string is valid; it is replaced by nothing.

Conditional Assembly and Macro Directives

This section describes the functions that can be performed using the APW conditional assembly and macro directives.

ACTR Assembly Counter

ACTR *numb* *[comment]*

The ACTR directive is used to limit the number of loops caused by conditional assembly branches. Each time a branch is made in a macro definition, a counter is decremented. If it reaches 0, processing of the macro stops to protect it from infinite loops.

numb This number must be in the range of 1 to 255. The loop counter is set from the value in *numb*.

The counter value is set to 255 automatically at the beginning of each macro. In loops with more than 255 iterations, it must be reset within the bounds of the loop to prevent the counter from reaching 0. The ACTR directive is only printed if it contains an error or TRACE ON is in effect.

AGO Assembler GO

```
AGO                 .symbol         [comment]
```

The conditional assembly directive AGO causes an unconditional branch. The target of the branch is determined by the operand.

.symbol This operand is required and it must contain a sequence symbol. The macro definition (or subroutine if not used in a macro) is searched for a matching sequence symbol. Processing continues with the instruction immediately following the sequence symbol. The period (.) in the sequence symbol causes the APW Assembler to search forward first, then backward. If you know that the sequence symbol appears before the AGO directive, replace the period with a ^ character and the assembler does only the backward search.

The search range for a source file includes the entire file, not just the subroutine containing the AGO directive. Searching begins in the forward direction and continues until the sequence symbol is found or the end of the file is reached. The search then begins with the instruction before the AGO directive and continues toward the beginning of the file. Searches for sequence symbols will not cross into a copied or appended file; they are limited to the file in memory.

The search process in a macro definition is similar, except that the search will not cross a MEND or MACRO directive.

The AGO directive is not printed in the output listing unless it contains an error. Sequence symbols appear in the output only if TRACE ON is in effect.

In the following example, the assembler encounters the initial AGO directive. Processing continues at the sequence symbol. All lines between the AGO and sequence symbol are ignored by the assembler.

```
            AGO                 .THERE
!           THESE LINES ARE IGNORED.
            .
            .
            .
.THERE
```

AIF Assembler IF

	AIF	*boolexp, .symbol*	*[comment]*

The Assembler If conditional assembly directive is used to do conditional branches. The Boolean phrase in the operand is tested. If true, processing continues with the first statement following the sequence symbol; if false, processing continues with the first statement following the AIF directive.

boolexp The required Boolean expression part of the operand.

.symbol The required sequence symbol part of the operand.

As with the AGO directive, the period (.) in the sequence symbol may be replaced with a ^ character to speed up branches in the case where the destination sequence symbol comes before the AIF directive.

The AIF directive is not printed in the output listing unless it contains an error. Sequence symbols appear in the output only if TRACE ON is in effect.

As an example, consider a file that contains the following statements:

```
            LCLA    &LOOP
&LOOP       SETA    4
.TOP
            ASL     A
&LOOP       SETA    &LOOP-1
            AIF     &LOOP>0,.TOP
or
            AIF     &LOOP,^.TOP
```

The output listing will contain these lines:

```
            ASL     A
            ASL     A
            ASL     A
            ASL     A
```

AINPUT Assembler Input

&spar	AINPUT	*[string]*	*[comment]*

The conditional assembly directive AINPUT is used to set the value of a symbolic parameter from the keyboard during assembly.

&spar This must be a character-type symbolic parameter.

[*string*] This operand is optional, but if coded, consists of a literal string. If the operand is coded, the string contained in the operand is displayed on the screen as an input prompt during pass 1 of the assembly.

When the AINPUT directive is encountered during pass 1, the prompting string is displayed if one has been coded. Then the assembler pauses. Otherwise the APW Assembler just pauses and waits for you to enter a line at the keyboard. The string that you enter in response to the pause is assigned to the character-type symbolic parameter specified in the label field.

The keyboard responses made during pass 1 are saved. When the AINPUT directive is encountered during pass 2, the response from pass 1 is again placed in the symbolic parameter specified in the label field so that you can use it for conditional branching.

AMID Assembler Mid String

&spar AMID *opstring,pos,num [comment]*

This is a special character-type set symbol that allows you to extract a substring. The character string assigned is the substring specified by the three arguments. The arguments are separated by commas, with no intervening spaces allowed. Spaces may be included in quoted strings.

&spar The label is required and must be a character-type symbolic parameter. This is where the resulting string is assigned.

opstring This is the string to be operated upon. It must be a string without any concatenations. If the string contains embedded spaces or commas, it must be enclosed in quotation marks. Quotation marks within quotation marks must be doubled.

pos The position within the target string of the first character to be chosen. It must be greater than 0. Characters from the target string are numbered sequentially, starting with 1.

num The number of characters to be chosen.

If the combination *pos* and *num* results in an attempt to select a character after the last character of the target string, the selection is terminated. Characters already selected are still valid.

Examples:

```
&CHAR     AMID      TARGET,2,3
&CHAR     AMID      TARGET,5,3
&CHAR     AMID      TARGET,7,3
```

The first example results in ARG being assigned to &CHAR; the second, in ET; the third, a null string.

ASEARCH Assembler String Search

&spar	ASEARCH	*search,target,position [comment]*

ASEARCH lets you search a string for substrings. It returns an arithmetic result, and can therefore only be used to assign values to arithmeti- type symbolic parameters. The arguments are separated by commas, with no intervening spaces allowed. Literal strings containing spaces or commas must be enclosed in quotation marks; quotation marks inside quotation marks must be doubled.

&spar The label is required and must be an arithmetic-type symbolic parameter.

search This parameter contains the string to be searched.

target The target substring to be searched for.

position The character position within the search string where the search is to begin.

If the substring is found within the target string, the character position of the first match is assigned to *&spar*. If the search string is not found in the target string, a value of 0 is returned.

Examples:

```
&NUM        ASEARCH     TARGET,GE,1
&NUM        ASEARCH     &NUM,4,1
&NUM        ASEARCH     'GOT A SPACE?',' ',1
```

The resulting value assigned to &NUM is 4 in the first example. In the second example, &NUM (an arithmetic symbolic parameter) is coded. However, since all symbolic parameters expand to characters strings when symbolic parameter substitution takes place, the ASRCH directive considers the resulting string (the character 4 from the previous result) in evaluating the operand. Symbolic parameter substitution always takes place before directive argument evaluation.

In the typical case, at least one of the arguments of ASEARCH will be a symbolic parameter simply because the use of a search string function would usually be superfluous with two literal strings.

GBLA Define Global Arithmetic Parameter

	GBLA	*&spar*	*[comment]*

The scope of the global arithmetic symbolic parameter defined by GBLA is the entire segment in which the current directive resides (both within the macro and outside of it in the source file). GBLA has no label.

&spar The operand field consists of the name of the symbolic parameter to be defined. If the symbolic parameter is to be subscripted, a number in the

range 1-255 must be specified in parentheses immediately following the symbolic parameter name.

Symbolic parameter definition statements are not printed in the output listing unless they contain errors, or if the TRACE ON directive is in effect.

Suppose that you have defined the following macro and stored it in a macro file:

```
            MACRO
            VERSION
            GBLA    &VERSION
&VERSION    SETA    3
            MEND
```

Your source code program can invoke the VERSION macro and then use the global symbolic parameter &VERSION. For example:

```
MAIN        START
            VERSION
            LDA #&VERSION
            .
            .
            .
            END
```

GBLB — Define global Boolean parameter

```
            GBLB            &spar           [comment]
```

The scope of the global Boolean symbolic parameter defined by GBLB is the entire code segment in which the current directive resides (both within the macro and outside of it in the source code). GBLB has no label.

&spar The operand field consists of the name of the symbolic parameter to be defined. If the symbolic parameter is to be subscripted, a number in the range 1-255 must be specified in parentheses immediately following the symbolic parameter name.

Symbolic parameter definition statements are not printed in the output listing unless they contain errors, or if the TRACE ON directive is in effect.

For example, the statement

```
            GBLB            &BOOL
```

defines a global Boolean symbolic parameter and initializes it to zero.

GBLC Define Global Character Parameter

```
GBLC        &spar        [comment]
```

The scope of the global character symbolic parameter defined by GBLC is the entire code segment in which the current directive resides. GBLC has no label.

&spar The operand field consists of the name of the symbolic parameter to be defined. If the symbolic parameter is to be subscripted, a number in the range 1-255 must be specified in parentheses immediately following the symbolic parameter name.

Symbolic parameter definition statements are not printed in the output listing unless they contain errors, or if the TRACE ON directive is in effect.

For example, the statement

```
GBLC        &STR
```

defines a global character symbolic parameter and initializes it to a null string. The statement

```
GBLC        &STRING(10)
```

defines &STRING(1) through &STRING(10) all as global character symbolic parameters, initialized to null strings.

LCLA Define Local Arithmetic Parameter

```
LCLA        &spar        [comment]
```

LCLA defines a local arithmetic symbolic parameter. The parameter is valid only within the macro or source code segment in which it is defined. LCLA does not contain a label.

&spar The operand field consists of the name of the symbolic parameter to be defined. If the symbolic parameter is to be subscripted, the name must be followed immediately by a number in the range 1-255 enclosed in parentheses.

Symbolic parameter definition statements are not printed in the output listing unless they contain errors, or if the TRACE ON directive is in effect.

For example, the statement

```
LCLA        &NUM
```

defines the local arithmetic symbolic parameter &NUM and initializes it to 0.

LCLB Define Local Boolean Parameter

```
LCLB        &spar        [comment]
```

LCLB defines a local Boolean symbolic parameter. The parameter is valid only inside the macro or within the source code segment in which it is defined. The LCLB statement has no label.

&spar The operand field consists of the name of the symbolic parameter to be defined. If the symbolic parameter is to be subscripted, the name must be followed immediately by a number in the range 1-255 enclosed in parentheses.

Symbolic parameter definition statements are not printed in the output listing unless they contain errors, or if the TRACE ON directive is in effect.

For example, the statement

```
LCLB        &BOOL
```

defines a local Boolean symbolic parameter and initializes it to zero.

LCLC Define Local Character Parameter

```
LCLC        &spar        [comment]
```

LCLC defines a local character symbolic parameter. The parameter is valid only within the macro, or source code segment in which it is defined. LCLC has no label.

&spar The operand field consists of the name of the symbolic parameter to be defined. If the symbolic parameter is to be subscripted, the name must be followed immediately by a number in the range 1-255 enclosed in parentheses.

Symbolic parameter definition statements are not printed in the output listing unless they contain errors, or if the TRACE ON directive is in effect.

For example, the statement

```
LCLC        &STR
```

defines a local character symbolic parameter and initializes it to a null string.

MACRO Begin a Macro Definition

```
        MACRO                                   [comment]
```

Each macro definition begins with a MACRO directive. This directive is coded in the operation field. No operand or label is needed, and any present is ignored. The macro definition that begins with this directive extends to a MEND directive.

MEND End a Macro Definition

```
        MEND                                    [comment]
```

The MEND directive terminates a macro definition. MEND is coded in the operation field and has neither operand nor label. If MEND appears outside of a macro definition, it generates an error.

MEXIT Exit Macro

```
        MEXIT                                   [comment]
```

The MEXIT directive indicates that a macro expansion is complete. MEXIT is used to exit from the macro before the end of the definition code is reached. Unlike MEND, it does not indicate the end of a macro definition. The directive is coded in the operation field. It has no operand and uses no label.

You can see in the example that follows how the code is grouped in the PULLLONG library macro by the use of MEXIT.

```
        MACRO
&LAB    PULLLONG  &ADDR1,&ADDR2
&LAB    ANOP
        AIF       C:&ADDR1=0,.a     Branch if no parameter
        AIF       C:&ADDR2=0,.b     Branch if 1 parameter
        LCLC      &C                Define local symbolic
                  parameter
&C      AMID      &ADDR1,1,1        Get 1st character of 1st
                                    parameter.
        AIF       "&C"="[",.directpage    Assume direct page
                                    indirect long addressing.
        PULLWORD  &ADDR1            Call pullword macro for 2
        STA       &ADDR2            parameter case.
        PULLWORD  &ADDR1+2
        STA       &ADDR2+2
        MEXIT
```

```
.A                                      No parameter case.
      PULLWORD                          Throw away 2 words on
      PULLWORD                          stack, then exit
      MEXIT
.B                                      One parameter case.
      PULLWORD   &ADDR1                 Call pullword macro
      PULLWORD   &ADDR1+2               Pass 1st parameter
      MEXIT
.DIRECTPAGE                             Direct page case.
      LDY        #&ADDR2                Load Y using 2nd parameter
      PULLWORD   &ADDR1,y               Save Y at addr1
      LDY        #&ADDR2+2
      PULLWORD   &ADDR1,y
      MEND
```

MNOTE Macro Note

[label] MNOTE*'message'[,numb]* *[comment]*

A macro definition may include a MNOTE directive.

'message' The operand of a MNOTE directive contains a character string with a message. The assembler prints the message on the output device as a separate line.

[,numb] This optional parameter, if included, is preceded by a comma. The parameter itself is a number that represents the severity code for an error.

For example, the following statements might appear in a program:

```
*           MNOTE  FOLLOWS
             MNOTE              'ERROR!',4
```

The listing contains these lines:

```
            0432  10FE          *        MNOTE  FOLLOWS
            ERROR!
```

Assuming that there were no other errors in the assembly, the maximum error level found (printed at the end of the assembly) would be four.

MNOTE is designed for use when conditional assembly directives are used to scan parameters passed via a macro call for syntax that you define. Although MNOTE statements are intended for use inside macros, they are legal inside of a source program.

SETA Set Arithmetic Parameter

&spar	SETA	*cexpress*	*[comment]*

The Set Arithmetic Parameter directive sets the value of an arithmetic parameter. SETA functions like an assignment statement.

&spar The label is required and must be an arithmetic-type symbolic parameter.

cexpress The operand field consists of a constant expression which resolves to a 32-bit signed number. The result is assigned directly to *&spar*.

Examples:

```
&NUM        SETA    4
&NUM2       SETA    &NUM+LABEL*4
```

The same expression evaluation rules as for absolute addresses are used in resolving the operand of a SETA directive. This means that Boolean expressions are allowed as well.

SETB Set Boolean Parameter

&spar	SETB	*boolexp*	*[comment]*

The SETB directive functions like an assignment statement to set the value of a Boolean symbolic parameter.

&spar The label is required and must be a Boolean-type symbolic parameter.

boolexp The operand field consists of a constant expression in the range from 0 to 255. The result is stored as an 8-bit unsigned value.

Normally, only Boolean operators are used in the SETB operand, resulting in boolean results (0 for false and 1 for true). However, if arithmetic results are produced, any nonzero value will be treated as true if the symbolic parameter is used in a Boolean expression.

Examples:

```
&FLAG       SETB        A<&NUM
&LOGIC      SETB        &NUM>0
&LOGIC      SETB        1
```

SETC Set Character Parameter

&spar	SETC	*cstring*	*[comment]*

The SETC directive functions like an assignment statement to set a character-type symbolic parameter.

&spar The label is required and must be a character-type symbolic parameter.

cstring The operand of SETC is evaluated as a character string and assigned to the symbolic parameter in the label field.

Examples:

```
&STRING(4) SETC        &STRING
&STR       SETC        &FKENAME'.OBJ'
&STRING    SETC        'Here''s a quoted string'
```

Whenever a character-type symbolic parameter is given a value, either via a macro call parameter or by a set symbol statement, special considerations apply. The key point to remember is that the string will be evaluated by the APW Assembler at least twice: first in the source line or macro expansion line where the character-type symbolic parameter is set, and a second time, after symbolic parameter substitution occurs and the newly formed line is evaluated by the assembler.

In general, whenever strings are used in the assembler (for example, within a character-type DC statement), embedded spaces are not allowed. When character-type symbolic parameters are given values, literal strings containing spaces, commas, single or double quotation marks, or plus signs must be enclosed in quotation marks. Either single or double quotation marks are allowed, provided that the same enclosing characters are used to start and stop the string. If the literal string must contain quotation marks, then double quotation marks must be used to contain single quotation marks, and single quotation marks to contain double quotation marks, or the quotation mark must be repeated.

An example of repeated single and double quotation marks:

```
DC         C'don''t forget!'
DC         C"""What?"""
```

The triple quotation marks in the second line are a single double quotation followed by a repeated double quotation. The resultant string is `"What?"`. When quotation marks are repeated, a problem results if the literal string will be evaluated by the assembler more than once. This is because each time the string is evaluated, double quotation marks are reduced to single quotation marks. This occurs when literal, quoted strings are passed to macros.

The following convention is recommended to deal with the problem of what kind of quotation marks to use where: always use single quotation marks in source files, repeating them as necessary, and always use double quotation marks within macro definitions.

For example, a macro PUTSTR, invoked as follows:

```
PUTSTR    'a string with ""some"" quotation marks'
```

might include the statement

```
DC        C"&PARM1"
```

The value assigned to &PARM1 will have no leading or trailing single quotation marks, and, because the double quotation character was not used as the string delimiter, the repeated double quotation characters are passed through to the model statement expansion. When the DC statement is finally evaluated by the assembler, it encounters the repeated double quotation marks and reduces them to single quotation marks in the generated object code because within the macro definition, double quotation marks are used as the string start/stop characters.

Whenever character symbolic parameters are being given values, the plus character (+) may be used to concatenate two literal strings.

For example:

```
&STR      SETC          '&'+'STR'
```

results in a symbolic parameter with the value &STR. Normally this value would be impossible to enter literally because the ampersand (&) involves symbolic parameter substitution.

&SYSCNT Subroutine Counter

```
&SYSCNT
```

&SYSCNT is a predefined symbolic parameter. The value of &SYSCNT is set to 1 at the beginning of each subroutine and is incremented at the beginning of each macro expansion. It is used to prevent labels defined inside of a macro from being duplicated if the same macro is used more than once in the same subroutine. This is done by concatenating &SYSCNT to the end of any labels used within the macro definition itself.

```
&LAB    ADDLONG    &NUMB1,&NUMB2
```

The macro file is a delay loop that does successive decrements of the accumulator to use up time.

```
            MACRO
&lab        WAIT &delay      Setup symbolic parameter for
&lab        ANOP             delays.
            LDA  #&delay     Load A with the value in
                             &delay.
top&syscnt  DEC a            Decrement the value in A by 1.
            BNE  top&syscnt  Branch back to decrement
                             instruction.
            MEND
```

The source file calls the delay macro and sets a different delay value for each pass through the program.

```
              MCOPY WAIT
              GEN  ON
MAIN          START
              WAIT 5              Set delay loop to 5.
              WAIT 4              Set delay loop to 4.
              WAIT 3              Set delay loop to 3.
              END
```

The assembled program shows how the macro file is expanded.

```
0001 0000           MCOPY WAIT
0002 0000
0003 0000           GEN   ON
0004 0000
0005 0000      MAIN START
0006 0000
0007 0000           WAIT  5
0000               +
0000 A9 05 00      +       LDA   #5
0003 3A            +TOP2   DEC   A     TOP1 set before entry.
0004 D0 FD         +       BNE   TOP2  Fall through at 0.
0008 0006
0009 0006           WAIT  4
0006               +
0006 A9 04 00      +       LDA   #4
0009 3A            +TOP3   DEC   A
000A D0 FD         +       BNE   TOP3
0010 000C
0011 000C           WAIT  3
000C               +
000C A9 03 00      +       LDA    #3
000F 3A            +TOP4   DEC    A
0010 D0 FD         +       BNE    TOP4
0012 0012
0013 0012           END

13 source lines
3 macros expanded
12 lines generated
```

&SYSDATE System Date

```
&SYSDATE
```

&SYSDATE is a predefined symbolic parameter. It returns the date in the day/month/year format:

DD MMM YY

For example

04 JUL 87

&SYSNAME Segment Name

```
&SYSNAME
```

`&SYSNAME` is a predefined symbolic parameter. It returns the name of this current load segment.

&SYSTIME System Time

```
&SYSTIME
```

`&SYSTIME` is a predefined symbolic parameter. It returns the current time in the format:

HH:MM

Attributes and Symbolic Parameters

In order to use the conditional assembly directives effectively, you need a way of getting information about a symbolic parameter in addition to its value. This information is provided by means of attributes, which may be thought of as functions that return information about the symbolic parameter. Attributes are resolved during expression evaluation rather than expanded when symbolic parameter substitution takes place. This means that the following line is not expanded:

```
LDA #C:&FOO
```

it is not expanded. Instead `&FOO` is examined and the hexadecimal value of the code that is stored reflects the correct number of subscripts.

The form of an attribute is

X: *&spar*

where the attribute `X` can be represented by the characters `C`, `L`, `T`, or `S`, as follows:

`C` Count attribute
`L` Length attribute

`T` Type attribute

Attributes can be used in operands in the same ways that you use constants.

Count Attribute

The count attribute is used to tell whether or not a symbolic parameter has been defined, and if so, how many subscripts are available. It is normally used to find out if a multiple argument has been assigned to a symbolic parameter by a macro call. The count attribute of an undefined symbolic parameter is 0. The count attribute of a defined symbolic parameter that is not subscripted is 1. The count attribute of a subscripted symbolic parameter is the number of subscripts available. The count attribute is used in the following loop to initialize a numeric array for a symbolic parameter that may or may not be defined.

```
            LCLA  &N
&N          SETA  C:&ARRAY
AIF         &N=0,.PAST
.TOP
&ARRAY(&N)  SETA  &N
&N          SETA  &N-1
            AIF   &N,.TOP
.PAST
```

Although it seems like poor programming not to know if a symbolic parameter has been defined, there are two common uses for the count attribute. The first is when a macro will define a global symbolic parameter to communicate with any future versions of itself. In that case, the macro can test to make sure that the parameter has not been defined already. The following macro uses this method to define a sequence of integers. You don't need to count the macros; they count themselves.

```
        MACRO
&LAB    COUNT
AIF     C:&N>0,.PAST        If already defined, proceed.
GBLA    &N                  If no, define &N on 1st pass.
.PAST
&N      SETA &N+1           Increment &N by 1.
&LAB    DC   I'&N'          Generate 2 bytes containing
                            value in  &N.
        MEND
```

The second use is to check to make sure that a parameter was passed. In the following example, if the count of `&NUM3` is 0, processing moves to `.PAST`. Otherwise `&NUM3` is defined as a local parameter and assigned the value in `&NUM1`.

```
        MACRO
&LAB    ADD  &NUM1,&NUM2,&NUM3
        AIF  C:&NUM3,.PAST  Is there a 3rd parameter?
        LCLC &NUM3          If no, define &NUM3 and set
                            its value to the 1st
                            parameter.
&NUM3   SETC &NUM1
.PAST
```

```
&LAB    CLC
        LDA  &NUM1
        ADC  &NUM2
        STA  &NUM3
        LDA  &NUM1+1
        ADC  &NUM2+1
        STA  &NUM3+1
        MEND
```

Length Attribute

The length attribute of an arithmetic symbolic parameter is four. The length attribute of a Boolean symbolic parameter is one. For a character string symbolic parameter, the length attribute is the number of characters of the string. If the symbolic parameter is subscripted, the subscript of the desired element should be specified; otherwise, the first element is assumed.

In the following example, if `LEN` is equal to `'ABC'`, this macro will generate `LDA #3`:

```
        MACRO
        LEN  &STR
        LDA #L:&STR
        MEND
```

Type Attribute

The type attribute is used to distinguish between the following kinds of symbolic parameters:

- `X` Arithmetic symbolic parameter
- `Y` Boolean symbolic parameter
- `Z` Character symbolic parameter

A character can be tested logically as in

```
AIF   T:&label='X', .a
```

where if the character is an arithmetic symbolic parameter, processing continues at `.a`.

Part III

Appendixes

Appendix A
65816 Instruction Mnemonics and Addressing Modes

65861 Mnemonic	Addressing Modes	Number of Bytes
ADC	Absolute	3
	Absolute Indexed, X	3
	Absolute Indexed, Y	3
	Absolute Long	4
	Absolute Long Indexed, X	4
	Direct Page (DP)	2
	DP Indexed Indirect, X	2
	DP Indexed, X	2
	DP Indirect	2
	DP Indirect Indexed, Y	2
	DP Indirect Long	2
	DP Indirect Long Indexed, Y	2
	Immediate	2*
	SR Indirect Indexed, Y	2
	Stack Relative (SR)	2
AND	Absolute	3
	Absolute Indexed, X	3
	Absolute Indexed, Y	3
	Absolute Long	4
	Absolute Long Indexed, X	4
	Direct Page (DP)	2
	DP Indexed Indirect, X	2
	DP Indexed, X	2
	DP Indirect	2
	DP Indirect Indexed, Y	2
	DP Indirect Long	2
	DP Indirect Long Indexed, Y	2
	Immediate	2*
	SR Indirect Indexed, Y	2
	Stack Relative (SR)	2
ASL	Absolute	3
	Absolute Indexed, X	3
	Accumulator	1
	Direct Page (DP)	2
	DP Indexed, X	2
BCC	Program Counter Relative	2
BCS	Program Counter Relative	2
BEQ	Program Counter Relative	2
BIT	Absolute	3
BIT	Absolute Indexed	3
	Direct Page (DP)	2
	DP Indexed, X	2

	Immediate	2*
BMI	Program Counter Relative	2
BNE	Program Counter Relative	2
BPL	Program Counter Relative	2
BRA	Program Counter Relative	2
BRK	Stack/Interrupt	2 **
BRL	Program Counter Relative Long	3
BVC	Program Counter Relative	2
BVS	Program Counter Relative	2
CLC	Implied	1
CLD	Implied	1
CLI	Implied	1
CLV	Implied	1
CMP	Absolute	3
	Absolute Indexed, X	3
	Absolute Indexed, Y	3
	Absolute Long	4
	Absolute Long Indexed, X	4
	Direct Page (also DP)	2
	DP Indexed Indirect, X	2
	DP Indexed, X	2
	DP Indirect	2
	DP Indirect Indexed, Y	2
	DP Indirect Long	2
	DP Indirect Long Indexed, Y	2
	Immediate	2*
	SR Indirect Indexed, Y	2
	Stack Relative (also SR)	2
COP	Stack/Interrupt	2 **
CPX	Absolute	3
	Direct Page (also DP)	2
	Immediate	2†
CPY	Absolute	3
	Direct Page (also DP)	2
	Immediate	2†
DEC	Absolute	3
	Absolute Indexed, X	3
	Accumulator	1
	Direct Page (also DP)	2
	DP Indexed, X	2
DEX	Implied	1
DEY	Implied	1
DLA	SR Indirect Indexed, Y	2
EOR	Absolute	3
	Absolute Indexed, X	3
	Absolute Indexed, Y	3
	Absolute Long	4
	Absolute Long Indexed, X	4
	Direct Page (also DP)	2
	DP Indexed Indirect, X	2
	DP Indexed, X	2
	DP Indirect	2
	DP Indirect Indexed, Y	2
	DP Indirect Long	2

	DP Indirect Long Indexed Y	2
	Immediate	2*
	SR Indirect Indexed, Y	2
	Stack Relative (also SR)	2
INC	Absolute	3
	Absolute Indexed, X	3
	Accumulator	1
	Direct Page (also DP)	2
	DP Indexed, X	2
INX	Implied	1
INY	Implied	1
JMP	Absolute	3
	Absolute Indexed Indirect	3
	Absolute Indirect	3
	Absolute Indirect Long	3
	Absolute Long	4
JSR	Absolute	3
	Absolute Indexed Indirect	3
	Absolute Long	4
LDA	Absolute	3
	Absolute Indexed, X	3
	Absolute Indexed, Y	3
	Absolute Long	4
	Absolute Long Indexed, X	4
	Direct Page (also DP)	2
	DP Indexed Indirect, X	2
	DP Indexed, X	2
	DP Indirect	2
	DP Indirect Indexed, Y	2
	DP Indirect Long	2
	DP Indirect Long Indexed, Y	2
	Immediate	2
	Stack Relative (alsoSR)	2
LDX	Absolute	3
	Absolute Indexed, Y	3
	DP Indexed, Y	2
	Direct Page (also DP)	2
	Immediate	2†
LDY	Absolute	3
	Absolute Indexed, X	3
	Direct Page (also DP)	2
	DP Indexed, X	2
	Immediate	2†
LSR	Absolute	3
	Absolute Indexed, X	3
	Accumulator	1
	Direct Page (also DP)	2
	DP Indexed, X	2
MVN	Block Move	3
MVP	Block Move	3
NOP	Implied	1
ORA	Absolute	3
	Absolute Indexed, X	3

	Absolute Indexed, Y	3
	Absolute Long	4
	Absolute Long Indexed, X	4
	Direct Page (also DP)	2
	DP Indexed Indirect, X	2
	DP Indexed, X	2
	DP Indirect	2
	DP Indirect Indexed, Y	2
	DP Indirect Long	2
	DP Indirect Long, Indexed	2
	Immediate	2 *
	SR Indirect Indexed, Y	2
	Stack Relative (also SR)	2 **
PEA	Stack (Absolute)	3
PEI	Stack (Direct Page Indirect)	2
PER	Stack (PC Relative Long)	3
PHA	Stack (Push)	1
PHB	Stack (Push)	1
PHD	Stack (Push)	1
PHK	Stack (Push)	1
PHP	Stack (Push)	1
PHX	Stack (Push)	1
PHY	Stack (Push)	1
PLA	Stack (Pull)	1
PLB	Stack (Pull)	1
PLD	Stack (Pull)	1
PLP	Stack (Pull)	1
PLX	Stack/Pull	1
PLY	Stack/Pull	1
REP	Immediate	2
ROL	Absolute	3
	Absolute Indexed, X	3
	Accumulator	1
	Direct Page (also DP)	2
	DP Indexed, X	2
ROR	Absolute	3
	Absolute Indexed, X	3
	Accumulator	1
	Direct Page (also DP)	2
	DP Indexed, X	2
RTI	Stack /RTI	1
RTL	Stack (RTL)	1
RTS	Stack (RTS)	1
SBC	Absolute	3
	Absolute Indexed, X	3
	Absolute Indexed, Y	3
	Absolute Long	4
	Absolute Long Indexed, X	4
	Direct Page (also DP)	2
	DP Indexed Indirect, X	2
	DP Indexed, X	2
	DP Indirect	2
	DP Indirect Indexed, Y	2
	DP Indirect Long	2

	DP Indirect Long Indexed, Y	2
	Immediate	2*
	SR Indirect Indexed, Y	2
	Stack Relative (also SR)	2
SEC	Implied	1
SED	Implied	1
SEI	Implied	1
SEP	Immediate	2
STA	Absolute	3
	Absolute Indexed, X	3
	Absolute Indexed, Y	3
	Absolute Long	4
	Absolute Long Indexed, X	4
	Direct Page (also DP)	2
	DP Indexed Indirect, X	2
	DP Indirect	2
	DP Indirect Indexed, Y	2
	DP Indirect Long Indexed, Y	2
	DP Indirect Long	2
	DP Indexed, X	2
	Stack Relative (also SR)	2
	SR Indexed, Y	2
STP	Implied	1
STX	Absolute	3
	Direct Page	2
	Direct Page Indexed, Y	2
STY	Absolute	3
	Direct Page	2
	Direct Page Indexed, X	2
STZ	Absolute	3
	Absolute Indexed, X	3
	Direct Page	2
	Direct Page Indexed, X	2
TAX	Implied	1
TAY	Implied	1
TCD	Implied	1
TCS	Implied	1
TDC	Implied	1
TRB	Absolute	3
TRB	Direct Page	2
TSB	Absolute	3
TSB	Direct Page	2
TSC	Implied	1
TSX	Implied	1
TXA	Implied	1
TXS	Implied	1
TXY	Implied	1
TYA	Implied	1
TYX	Implied	1
WAI	Implied	1
WDM	reserved	2

XBA	Implied	1
XCE	Implied	1

---

* Add one byte if m=0 (16-bit memory/accumulator)
** The operation code is one byte, but the program counter value pushed onto the stack is incremented by 2 allowing for an optional signature byte.
† Add one byte if x=0 (16-bit index registers)

Appendix B

The ASCII Character Set

Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex
nul	0	0	0	sp	32	40	20	@	64	100	40	`	96	140	60
soh	1	1	1	!	33	41	21	A	65	101	41	a	97	141	61
stx	2	2	2	"	34	42	22	B	66	102	42	b	98	142	62
etx	3	3	3	#	35	43	23	C	67	103	43	c	99	143	63
eot	4	4	4	$	36	44	24	D	68	104	44	d	100	144	64
enq	5	5	5	%	37	45	25	E	69	105	45	e	101	145	65
ack	6	6	6	&	38	46	26	F	70	106	46	f	102	146	66
bel	7	7	7	'	39	47	27	G	71	107	47	g	103	147	67
bs	8	10	8	(	40	50	28	H	72	110	48	h	104	150	68
ht	9	11	9	)	41	51	29	I	73	111	49	i	105	151	69
lf	10	12	A	*	42	52	2A	J	74	112	4A	j	106	152	6A
vt	11	13	B	+	43	53	2B	K	75	113	4B	k	107	153	6B
ff	12	14	C	,	44	54	2C	L	76	114	4C	l	108	154	6C
cr	13	15	D	-	45	55	2D	M	77	115	4D	m	109	155	6D
so	14	16	E	.	46	56	2E	N	78	116	4E	n	110	156	6E
si	15	17	F	/	47	57	2F	O	79	117	4F	o	111	157	6F
dle	16	20	10	0	48	60	30	P	80	120	50	p	112	160	70
dc1	17	21	11	1	49	61	31	Q	81	121	51	q	113	161	71
dc2	18	22	12	2	50	62	32	R	82	122	52	r	114	162	72
dc3	19	23	13	3	51	63	33	S	83	123	53	s	115	163	73
dc4	20	24	14	4	52	64	34	T	84	124	54	t	116	164	74
nak	21	25	15	5	53	65	35	U	85	125	55	u	117	165	75
syn	22	26	16	6	54	66	36	V	86	126	56	v	118	166	76
etb	23	27	17	7	55	67	37	W	87	127	57	w	119	167	77
can	24	30	18	8	56	70	38	X	88	130	58	x	120	170	78
em	25	31	19	9	57	71	39	Y	89	131	59	y	121	171	79
sub	26	32	1A	:	58	72	3A	Z	90	132	5A	z	122	172	7A
esc	27	33	1B	;	59	73	3B	[	91	133	5B	{	123	173	7B
fs	28	34	1C	<	60	74	3C	\	92	134	5C	|	124	174	7C
gs	29	35	1D	=	61	75	3D	]	93	135	5D	}	125	175	7D
rs	30	36	1E	>	62	76	3E	^	94	136	5E	~	126	176	7E
us	31	37	1F	?	63	77	3F	_	95	137	5F	del	127	177	7F
Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex	Char	Dec	Oct	Hex

Appendix C

Error Messages

When a program is assembled into object files, the APW Assembler checks for a variety of problems. These problems are flagged with severity codes to let you know how serious the problem is. The APW Assembler can report the four possible error levels described in the table which follows. The APW Assembler prints the highest error level found at the end of the assembly.

Severity	Meaning
2	Warning — things may be ok.
4	Error — an error was made, but the assembler thinks it knows the intent and has corrected the mistake. Check the result carefully!
8	Error — code was processed, but your program probably will not run. Level 8 errors do not take you back to the APW Editor.
16	Error — it was not even possible to tell how much space to leave. You will need to reassemble to fix the problem.

Source program errors fall into two broad categories: those that cause the assembler to terminate inadvertently (fatal errors) and those that do not (non-fatal errors).

Non-Fatal APW Assembler Errors

When the assembler finds an error that it can recover from, it prints the error on the line after the source line that contained the error. Only one error per line is flagged, even if there is more than one error in the line.

The error message is a brief description of the error. In the sections that follow, each of the error messages is listed, in alphabetical order. The error message description is also given with ways to correct the problem.

ACTR Count Exceeded [16]

More than the allowed number of `AIF` or `AGO` directives were encountered during a macro expansion. Unless changed by the `ACTR` directive, only 255 `AIF` or `AGO` branches are allowed in a single macro expansion. This is a safeguard to prevent infinite loops during

macro expansions. If more than 255 branches are needed, use the ACTR directive inside the loop to keep the count sufficiently high.

Address Length Not Valid [2]

An attempt was made to force the assembler to use an operand length that is not valid for the given instruction. For example, indirect indexed addressing requires a one-byte operand, so forcing an absolute address by coding

```
LDA     ( | 2 ),Y
```

would result in this error.

Addressing Errors [16]

The program counter when pass 1 defined a label was different from the program counter when pass 2 encountered the label. There are three likely reasons for this to happen. The first is if, for some reason, the result of a conditional assembly test was different on the two passes; this is actually caused by one of the remaining errors. The second is if a label is defined using an EQU to be a long or direct page address; then the label is used before the EQU directive is encountered. The last reason is if a label has been defined as direct page or long using a GEQU directive, then redefined as a local label. On the first pass in both of these cases, the assembler assumes a length for the instruction which is then overridden before pass 2 starts.

An intra-segment reference was made while an OBJ directive as in effect and the reference was not forced to absolute long addressing mode.

Duplicate Label [4]

1. Two or more local labels were defined using the same name. The first such label gets flagged as a duplicate label; subsequent redefinitions are flagged as addressing errors. Any use of the label will result in the first definition being used.

2. Two or more symbolic parameters were defined using the same name. Subsequent definitions are ignored.

Duplicate Ref In MACRO Operand [2]

A parameter in a macro call was assigned a value two or more times. This usually happens when both a keyword and positional parameter set the same symbolic parameter. For the macro

```
MACRO
EXAMPLE      &P1,&P2
MEND
```

the call

```
EXAMPLE      A,P1=B
```

would produce this error because, `P1` is set to `A` as a positional parameter, then to `B` as a keyword parameter.

Duplicate Segments [8]

Two segments have appeared during the same assembly with the same segment name. The assembler flags the second and all subsequent segments with level 8 errors.

Error In Expression [8]

Either the expression contains an error, such as mismatched parentheses, or the expression had too many terms for the assembler to handle. There is no fixed limit to the number of terms or level of parentheses in an expression, but generally the assembler will handle as many terms as will fit on a line, and about twenty levels of parentheses. Check for any kind of syntax error in the expression itself.

Error Too Complex [8]

1. An expression contained a label whose value was defined with an `EQU` or `GEQU` that contained an expression. The expression also contained a label defined in the same way, and so on, for ten levels. You need to reduce the level of expressions by coding some of the terms in long hand.

2. Too many parentheses were used. The exact number of parentheses allowed depends upon the type of expression that you use, but is generally twenty levels. If this error occurs, reduce the number of parentheses you are using in the expression.

3. A subscripted symbolic parameter was used to specify the index of another symbolic parameter, as in

```
&A(&B(4))
```

Subscripts for symbolic parameters cannot themselves contain subscripted symbolic parameters. Eliminate one of the subscripts by assigning the value to a different symbolic parameter, as in

```
&C    SETA &B(4)
```

Then you can use `&A(&C)`.

4. An `ORG`, `OBJ`, or `DIRECT` directive had an expression in its operand that did not resolve to a constant at assembly time. Replace the expression with a constant.

5. A GEQU directive appeared outside of a segment with an operand that did not resolve to a constant. You need to move the GEQU inside of the segment. Be sure that this segment is reassembled on all partial assemblies.

Invalid Operand [8]

An operand contains an addressing mode not valid for this instruction.

Label Syntax [16]

1. The label field of a statement contained a string that does not conform to the standard label syntax. A label must begin in column 1, and cannot contain imbedded spaces. Each label starts with an alphabetic character (A to Z), tilde character (~), or underscore (_), and can be followed by zero or more alphabetic characters, underscores, or tildes. Labels are significant up to 255 characters in length. There must be at least one space between the label and the operation code.

2. A macro model statement had something other than a symbolic parameter in the label field. If anything occupies the label field of the statement immediately following a MACRO directive, it must be a symbolic parameter.

Length Exceeded [4]

1. An expression was used in an operand that requires a direct page result, and the expression was not in the range 0 to 255. If external labels are used in the expression, and the result will resolve to direct page when the linker resolves the references, force direct page addressing by preceding the expression with a < character. For example

```
LDA (<LABEL),Y
```

If the expression is a constant expression, correct it so that it is in the range 0 to 255.

2. A directive which requires a number in a specific range received a number outside that range in the directive. See specific directive descriptions for allowed parameter ranges.

Macro File Not In Use [2]

An MDROP was found that specified a macro filename that was never opened with an MLOAD or MCOPY, or that has already been closed with an MDROP. Remove the extra MDROP.

MACRO Operand Syntax Error [4]

The operand of the macro model statement contained something other than a sequence of undefined symbolic parameters separated by commas. If this line has an operand at all, it must consist of a list of symbolic parameters separated by commas, with no imbedded spaces.

Missing Label [2]

1. A DATA, ENTRY, EQU, GEQU, PRIVATE, PRIVDATA, or START directive was found that did not have a label. Because the purpose of these directives is to define a label, a label is required.

2. A SETA, SETB, or SETC directive was found that did not have a symbolic parameter in the label field. Since the the purpose of these directives is to set the value of the symbolic parameter in the label field, a symbolic parameter is required.

Missing Operand [16]

The operation code was one that required an operand, but no operand was found. Make sure that the comment column has not been set to too low a value; see the description of the SETCOM directive. Remember that the assembler requires the A as an operand for the accumulator addressing mode.

Missing Operation [16]

There was no operation code on a line that was not a comment. Make sure that the comment column has not been set to too small a value; see the SETCOM directive. Keep in mind that the operation codes cannot start in column 1.

Misplaced Statement [16]

1. A statement that must appear inside of a code segment was used outside of a code segment. Only the following directives can be used outside a code segment:

AGO	ERR	MCOPY	65C02
AIF	EXPAND	MDROP	65816
ALIGN	GEN	MERR	SYMBOL
APPEND	GEQU	MLOAD	TITLE
CODECHK	IEEE	MSB	TRACE
COPY	KEEP	ORG	
DATACHK	LIST	PRINTER	
DYNCHK	LONGA	RENAME	
EJECT	LONGI	SETCOM	

The way to remember this list is that any directive or instruction that generates code or places information in the object module must appear inside a code segment.

2. A KEEP directive was used after the first START or DATA directive, or two KEEP directives were used for a single assembly. Only one KEEP directive is allowed in each source code segment, and it must come before any code is generated.

3. A RENAME directive was used inside a program segment. The RENAME directive must appear outside of the program segment.

4. An ORG directive was used incorrectly. The problem could be any of the following:

   a. An ORG directive with a constant operand was used inside a program segment.
   b. An ORG directive that was not a displacement from the location counter was used outside a program segment.
   c. Two ORG directives were used in front of the same code segment.

See the description of the ORG directive for details on its use.

5. More than one ALIGN directive was used for the same program segment.

Nest Level Exceeded [8]

Macros were nested more than four levels deep. A macro may use another macro, including itself, provided that the macro used resides in the same macro file as the macro that is using it, and provided that the calls are not nested more than four levels deep.

No MEND [4]

The attempt was made to expand a macro which did not have a MEND directive. All macro definitions must have MEND directives as their last lines.

Numeric Error In Operand [8]

1. An overflow or underflow occurred during the conversion of a floating-point or double precision number from the string form in the source file to the IEEE representation for the number. Floating-point numbers are limited to a calculated value of about 1E-38 to 1E38, while double-precision numbers are limited to a calculated value of about 1E-308 to 1E308. If this error occurs, the APW Assembler inserts the IEEE format representation for 0 on an underflow, and infinity for an overflow.

2. A decimal number was found in the operand field that was not in the range minus 2147483648 to plus 2147483647. Because all integers are represented as four-byte signed numbers, decimal numbers must be in this range.

3. A binary, octal, or hexadecimal constant was found that required more than 32 bits to represent. Constants are represented by a maximum of four bytes.

Operand Syntax [16]

This error covers a wide range of possible problems in the way an operand is written. Generally, a quick look at the operand field will reveal the problem. If this does not help, reread the section of the manual that describes the instruction, directive, or macro in error.

Operand Value Not Allowed [8]

1. An ALIGN directive was used with an operand that was not a power of two.

2. An ALIGN directive was used in a program segment that was either not itself aligned, or was not aligned to a byte value greater than or equal to the ALIGN directive used in the program segment. For example,

```
     ALIGN  4
T    START
     ALIGN  4
     END
```

is acceptable, but

```
     ALIGN  4
T    START
     ALIGN  8
     END
```

will cause an error.

Rel Branch Out Of Range [8]

A relative branch has been made to a label that is too far away. For all instructions except BRL, relative branches are limited to a 1-byte signed displacement from the end of the instruction, giving a range of 129 bytes forward and 126 bytes backward from the beginning of the instruction. A 2-byte displacement is used for the BRL instruction, giving a range of 32770 bytes forward and 32765 bytes backward from the beginning of the instruction. The BRL instruction is only available in native mode.

Sequence Symbol Not Found [4]

A branch was attempted using an AIF or AGO directive, but the sequence symbol named in the operand field could not be found. A sequence symbol serves as the destination for a conditional assembly branch. It consists of a period in column 1, followed by the sequence symbol name starting in column 2. The sequence symbol name follows the same conventions as a label, except that symbolic parameters may not be used.

Set Symbol Type Mismatch [4]

The set symbol type does not match the type of the symbolic parameter being set. There are three types of symbolic parameters: A (arithmetic), B (Boolean), and C (character). All symbolic parameters defined in the parameter list of a macro call are C (character type). SETA and ASEARCH directives must have an arithmetic symbolic parameter; SETB directives must have a boolean symbolic parameter; and SETC, AMID, and AINPUT directives must have a character symbolic parameter in the label field.

Subscript Exceeded [8]

A symbolic parameter subscript was larger than the number of subscripts defined for it. For example,

```
            LDA     &NUM(4)
&NUM(5)     SETA    1
```

would cause this error since NUM is defined for 4 subscripts and the attempt is being made to use it with 5. A subscript of 0 will also cause this error.

Too Many MACRO Libs [2]

A MCOPY or MLOAD directive was encountered, and four macro libraries were already in use. The best solution is to use MACGEN and combine all the macros you need during an assembly into a single file. Not only does this get rid of the problem, it makes assemblies much faster. Another remedy is to use MDROP to get rid of macro libraries that you no longer need.

Too Many Positional Parameters [4]

The macro call statement used more parameters in the operand than the macro model statement had definitions for. Keep in mind that keyword parameters take up a position. For example, the following macro calls must all be to a macro definition with at least three parameters defined in the macro model statement operand.

```
CALL    L1,L2,LL3

CALL    ,,

CALL    L1,,L3

CALL    ,L1=A,L3
```

Undefined Directive In Attribute [8]

The settings attribute was requested for an undefined operation code, or for an operation code that does not use ON or OFF as its operand. The settings attribute is only defined for these directives:

ABSADDR	IEEE	MSB	SYMBOL
CASE	INSTIME	NUMSEX	TRACE
ERR	LIST	OBJCASE	65C02
EXPAND	LONGA	PRINTER	65816
GEN	LONGI		

Undefined Symbolic Parameter [8]

An ampersand (&) character followed by an alphabetic character was found in the source line. The assembler tried to find a symbolic parameter by the given name, and none was defined.

Unidetified Operation [16]

1. An operation code was encountered that was not a valid instruction or directive, nor was it a defined macro. If you are using 65C02 or 65816 instructions, make sure that they are enabled using the `65C02` and `65816` directives. Make sure `MCOPY` directives have been used to make all needed macros available at assembly time.

2. The first operation code in a `RENAME` directive's operand could not be found in the current list of instructions and directives.

3. A `MACRO`, `MEND`, or `MEXIT` directive was encountered in a source file.

Unresolved Label Not Allowed [2]

1. A directive operand contains an expression that must be explicitly evaluated to perform the assembly, but a label whose value could not be determined was used in the expression. In most cases, local labels cannot be used in place of a constant. Even though the assembler knows that the local label exists, it does now know where the label will finally be located by the linker.

2. The length or type attribute of an undefined symbolic parameter was requested. Only the count attribute is allowed for an undefined symbolic parameter.

Fatal Errors

Some errors are so severe that the APW Assembler cannot keep going; these are called fatal errors. When the assembler encounters a fatal error, it prints the error message and then waits for you to press a key. When you have pressed a key, the assembler passes control to the APW Editor, which loads the file that the assembler was working on and places the line that caused the fatal error at the top of the display screen.

File Could Not Be Opened

A ProDOS error occurred during an attempt to open a source or macro file.

This is usually caused by a bad file of some type, or a file that is missing entirely. Begin by carefully checking the spelling in the offending statement. Make sure that the file can be loaded with the listed filename using the editor. It is important to specify the pathname the same way as it is listed in the assembler command when doing this check. If the error occurs in a strange place where no files are asked for, keep in mind that a macro file is not loaded into memory until a macro is found and the error is probably in one of the `MCOPY` or `MLOAD` directives.

Keep File Could Not Be Opened

Either there was not enough memory to open the output file or a ProDOS error was encountered during an attempt to open the output file. Check the filename used in the `KEEP` directive for errors. This error will occur if the filename of the keep file exceeds ten

characters, because the assembler must be able to append .ROOT to the keep filename, and ProDOS restricts file names to fifteen characters.

Out of Memory

The assembler ran out of memory. Add more memory, reduce the size of the source files, reduce the size of the macro files, or reduce the number of symbols that you are using.

ProDOS Errors

The APW Assembler aborts if a ProDOS error is encountered while doing a file read or write. The ProDOS error message is reported.

No END

The APW Assembler aborts if it encounters a segment that does not have a closing END directive.

Symbol Table Overflow

The list that follows outlines the uses made of the symbol table. One or more of the uses will have to be reduced to avoid this error.

1. Each macro in the macro file that is currently open requires 12 bytes. Because only one macro file is open at a time, splitting a macro file into shorter files can help. It is not the length of a macro or the macro file that is a problem, but rather the actual number of macros in a file.

2. Each symbol defined using the GEQU directive requires 17 bytes of symbol table space. This space is not released at the end of each subroutine. The GEQU directive is only needed for specifying fixed direct page or long addresses; using the EQU directive in a data area and issuing a USING directive for the data area in the subroutine will do just as well for other purposes, and the used symbol table space is released as soon as the data area has been assembled.

3. Each local label in a segment requires seventeen bytes of space. This space is released as soon as the segment has been assembled. Shorter segments reduce the total number of local symbols in each.

4. Symbolic parameters require a variable amount of symbol table space. Reducing the total number or cutting down on the depth of macro calls can help.

5. The AINPUT directive saves the answers typed from the keyboard in the symbol table. These answers are removed when the segment where the AINPUT directive appears has been assembled. Two ways exist to reduce this kind of use: either split the segment so that fewer AINPUT directives are in any one segment, or answer the questions posed by the directive more briefly.

Unable To Write To Object Module

A ProDOS error was encountered while writing to the object module. This error is usually caused by a full disk, but could also be caused by a disk drive error of some sort.

Glossary

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.

accumulator: The register in the 65C816 microprocessor where most computations are performed. This register is set to 8-bits for 65C02 instructions and 16-bits for 65816 instructions.

addressing mode: A method of determining an effective address. The APW Assembler supports the 26 addressing modes of the 65816 instruction set.

Apple IIGS 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.**

Apple IIGS Debugger: The Apple IIGS Debugger works with 65816 assembly-language code. You can either step through your program one instruction at a time, or execute a program at full speed. You can set break points, which cause program execution to halt. The contents of registers, memory, direct page, and the stack can be examined while the program is halted.

APW Editor: The program that allows you to enter, modify, and save assembly-language source code programs.

APW Linker: The program that processes the object files generated by the APW Assembler to produce load files. The linker searches libraries, resolves all symbolic references, and generates a file that can be loaded into memory and executed.

APW Shell: The program that accesses ProDOS 16 and the toolbox directly, and that can be called or exited via the `QUIT` call.

assembly: The process of translating source code into object code.

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.

binary file format: A file in binary file format has ProDOS file type $06 and is referred to as a *BIN file*. The System Loader cannot load BIN files. Machine-language programs and pictures are stored in binary files.

branch: (v) To pass program control to a line or statement other than the next in sequence. (n) A statement that performs a branch. See **conditional branch, unconditional branch.**

byte: A unit of information consisting of a sequence of 8 **bits.** A byte can take any value between 0 and 255 ($0 and $FF hexadecimal). The value can represent an instruction, number, character, or logical state.

carry flag: A status bit in the microprocessor, used as an additional high-order bit with the accumulator bits in addition, subtraction, rotation, and shift operations.

case sensitivity: The ability to distinguish between uppercase characters and lowercase characters. Programming languages are *case sensitive* if they distinguish between uppercase and lowercase. With the APW Assembler, you can use the CASE directive to turn case sensitivity ON or OFF in the source file and OBJCASE to set case sensitivity in the object file produced by the assembler.

code segment: A segment that contains program code. The APW Assembler recognizes segments that begin with START or PRIVATE directives and terminate with END directives as code segments. See **data segment.**

comment: Source text intended for the user, ignored during assembly.

comment line: A source text line reserved for comments.

compiler: A program that produces object files (containing machine-language code) from source files written in high-level languages such as C.

concatenate: Literally, to chain together. To combine two or more strings into a single, longer string by joining the beginning of one to the end of the other. Also, to combine two or more files.

conditional assembly: A feature of an assembler that allows the programmer to define macros or other pieces of code such that the assembler assembles them differently under different conditions.

conditional branch: A **branch** whose execution depends on the truth of a condition or the value of an expression. Compare **unconditional branch.**

constant: A constant is either a name set equal to a value by a EQU or GEQU directive, or a binary, octal, or hexadecimal number, or characters.

data segment: A source or object segment that consists solely of data. The APW Assembler recognizes segments that begin with DATA or PRIVDATA directives and terminate with END directives as source data segments.

debugger: A utility program that allows you to analyze an application program for errors that cause it to malfunction. See the Apple IIGS Debugger.

default prefix: The pathname **prefix** attached by ProDOS 16 to a **partial pathname** when no **prefix number** is supplied by the application. The default prefix is equivalent to prefix number 0/.

desktop: The visual interface between the computer and the user—the menu bar and the gray area on the screen. You can have a number of documents on the desktop at the same time.

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.

dot operator: Indicated by a period (.), the dot operator is used to concatenate symbolic parameters.

dynamic segment: A segment that can be loaded and unloaded during execution as needed. Compare **static segment**.

e flag: One of three flag bits in the 65C816 processor that programs use to control the processor's operating modes. The setting of the e flag determines whether the processor is in **native mode** or **emulation mode**. See also **m flag** and **x flag**.

effective address: The address of the memory location on which a particular instruction operates. This address may be arrived at by using one of the assemblers **addressing modes**, for example, Absolute Indexed Indirect addressing or some other addressing method.

emulation mode: For 65C816 processor of the Apple IIGS, the state in which it functions like a 6502 processor in all respects except clock speed. For the Apple IIGS computer, the state in which the computer functions like an 8-bit Apple II.

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.

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 ProDOS 16 file type. Compare **pathname**.

file number: A reference number assigned to a specific file. The loader assigns a file number to each load file in a program; the MakeLib utility program assigns a file number to each object file incorporated into a library file.

fixed-point: A method of representing numbers in which the decimal point (more correctly, the binary point) is considered to occur at a fixed position within the number. Typically, the point is considered to lie at the right end of the number so that the number is interpreted as an integer.

floating-point: A method of representing numbers in which the decimal point (more correctly, the binary point) is permitted to "float" to different positions within the number. Some of the bits within the number itself are used to keep track of the point's position.

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 label in a source segment that is either the name of the segment or an entry point to it. Global labels may be referenced by other segments. Compare with **local label**.

high-order: The most significant part of a numerical quantity. In normal representation, the high-order bit of a binary value is in the leftmost position; the high-order byte of a binary **word** or **long word** quantity consists of the leftmost eight bits.

index: The variable component of an indexed address, contained in an index register and added to the base address to form the effective address.

index register: A register that holds an index for use in indexed addressing. The 65C816 microprocessor has two index registers: the **X register** and the **Y register.**

instruction: A unit of a machine-language or an assembly-language program corresponding to a single action for the processor to perform.

library file: An object file containing object segments, each of which can be used in any number of programs. The APW Linker can search through the library file for segments that have been referenced in the program source file.

LinkEd: A command language that can be used to control the APW Advanced 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.

link map: A listing, produced by the linker, that gives the name, length, and starting location of each segment in a load file.

load file: The output of the linker. Load files contain memory images that the system loader can load into memory and execute without further processing.

load segment: A segment in a load file. Any number of **object segments** can go into the same load segment.

local label: A label defined only within an individual segment. Other segments cannot access the label. Compare with **global label.**

long word: A double-length word. For the Apple IIGS, a long word is 32 bits (4 bytes) long.

low-order: The least significant part of a numerical quantity. In normal representation, the low-order bit of a binary number is in the rightmost position; likewise, the low-order byte of a binary **word** consists of the rightmost eight bits.

macro: A sequence of 65816 assembly-language source code that will be inserted into the source code program when the macro is expanded.

macro call: A request to expand a macro.

macro definition: Assembly-language source code that begins with a MACRO directive and ends with a MEND directive.

macro expansion: The insertion of the macro definition into the program source code. The APW Assembler marks each line of the macro expansion with a plus sign (+) if you use the GEN ON directive.

macro header: In a macro definition, the directive MACRO.

macro model statement: The line in an APW macro immediately following a MACRO directive.

main segment: The first static segment (other than initialization segments) in the initial load file of a program. It is loaded at startup and not removed from memory until the program terminates.

MakeLib utility: A program that creates library files from object files.

Memory Manager: A program in the Apple IIGS Toolbox that allocates blocks of memory as needed, and keeps track of which blocks of memory are available. All applications should request blocks of memory from the Memory Manager rather than loading data directly into a preselected memory location.

m flag: One of three flag bits in the 65C816 processor that programs use to control the processor's operating modes. In **native mode**, the setting of the m flag determines whether the accumulator is 8 bits wide or 16 bits wide. See also **e flag** and **x flag**.

microprocessor: A central processing unit that is contained in a single integrated circuit. The Apple IIGS uses a 65C816 microprocessor.

mnemonic: A sequence of characters that designate an instruction or directive.

native mode: The 16-bit operating state of the 65C816 processor.

object file: The output from the APW Assembler and input to the APW Linker. An object file conforms to APW object module format: it contains 65816 instructions plus the information the linker needs to resolve symbolic references. Any number of object files can be combined into a single load file. An object file must be processed by the linker to create a load file: it cannot be executed directly.

object module format (OMF): The general format used in object files, library files, and load files.

object segment: A segment in an object file.

operand: The information that the operation code uses to perform its function. In the case of 65816 instructions, the operand follows the operation code and is separated from it by at least one space. The operand is an expression that resolves to an address.

operation code: The part of a machine-language, or source code instruction that specifies the operation to be performed.

page: A portion of memory 256 bytes long and beginning at an address that is an even multiple of 256. Memory blocks whose starting addresses are an even multiple of 256 are said to be *page-aligned*. The ALIGN directive can be used to start a segment at the beginning of a page.

partial assembly: A procedure that assembles only specific segments of a program. The APW Assembler will assemble only the segments specified by the NAME parameter in the APW Shell Commands: RUN, ASMLG, ASML, and ASSEMBLE.

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.

pass one: The APW Assembler resolves local labels during the first assembly pass.

pass two: The APW Assembler produces object code, assembly listings, and external labels during the second pass of the assembly.

pathname: The complete name by which a file is specified. A pathname is a sequence of filenames separated by slashes, starting with the filename of the volume directory file and including every subdirectory file that the operating system must search to locate the file, in descending sequence of the subdirectory hierarchy. A full pathname always begins with a slash (/) to indicate that the first name is a volume directory. See also **full pathname, partial pathname, prefix.**

pipeline: (v) To automatically execute two or more programs in sequence, where the output of the first file is the input to the next file and so on. (n) The entire sequential set of programs executed in this way. A program or file being processed by this sequence of programs is said to be *in the pipeline* or *in the pipe.*

pop: To remove the top entry from a **stack,** moving the stack pointer to the entry below it. Synonymous with *pull.* Compare **push.**

prefix: 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.

prefix number: A code used to represent a particular prefix. Under ProDOS 16, there are nine prefix numbers, each consisting of a numeral followed by a slash: 0/, 1/,...8/, and */.

ProDOS 8: A disk operating system developed for standard Apple II computers. It runs on 6502-series microprocessors and 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.

program counter: A pointer to the memory location of the instruction to be executed next.

program segment: One or more code segments and optionally data segments assembled together to perform a function.

pull: To remove the top entry from a **stack,** moving the stack pointer to the entry below it. Synonymous with *pop.* Compare **push.**

push: To add an entry to the top of a **stack,** moving the stack pointer to point to it. Compare **pop.**

relational operator: An operator, such as >, that operates on numeric values to produce a logical result.

relocatable code: Code that is location independent.

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.

sequence symbol: The branching destinations for the conditional assembly directives `AGO` and `AIF`. The format consists of a period (.) followed by a label (the label may not

contain any symbolic parameters). They are not printed in the assembler listing unless the TRACE ON option is in effect.

shell call: A request from a program (or the assembler) to the APW Shell to perform a specific function.

Standard Apple Numeric Environment: The standard Apple data types, arithmetic operations, conversions, expression evaluations, and comparisons. What they are and how they are derived are described in the *Apple Numerics Manual.*

65C816: The microprocessor used in the Apple IIGS. The 65C816 is a CMOS device with 16-bit data registers and 24-bit address registers.

65C02: A CMOS version of the 6502; the microprocessor used in the Apple IIc and in the enhanced Apple IIe.

6502: The microprocessor used in the Apple II, in the Apple II Plus, and in early models of the Apple IIe. The 6502 is an MOS device with 8-bit data registers and 16-bit address registers.

source files: A source file for the APW Assembler consists of 65816 or 65C02 instructions, APW directives, macros, or data. The APW assembler converts source files into **object files.**

stack: An area of dedicated memory that is accessed by push and pull instructions. The area is used by these instructions for temporary data storage.

stack register: The stack register points to the next available location on the system stack.

standard Apple II: Any computer in the Apple II family except the Apple IIGS. That includes the Apple II, the Apple II Plus, the Apple IIe, and the Apple IIc.

static segment: A segment that is loaded only at program boot time, and is not unloaded during execution. Compare **dynamic segment.**

string: An item of information consisting of a sequence of text characters.

substring: A string that is part of another string.

symbol: A character or string of characters that represents an address or numeric value; a symbolic reference or a variable.

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.

symbolic parameter: A variable character or character string that represents addresses or values declared in the prototype statement of a macro definition. There are three types of symbolic parameters: A(arithmetic), B (Boolean), and C (character). Symbolic parameters are assigned values by the SETA, SETB, or SETC directives.

symbolic reference: A name or label, such as the name of a subroutine, that is used to refer to a location in a program. 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.

System Loader: The part of the operating system that reads the files generated by the linker, relocates them (if necessary), and loads them into memory. The System Loader works closely with ProDOS 16 and the Memory Manager.

text file format: The Apple IIGS standard format for text files and program source files.

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 **toolsets.**

tool set: A group of related routines (usually in firmware) that perform necessary functions or provide programming convenience. They are available to applications and system software. The Memory Manager, the System Loader, and QuickDraw II are Apple IIGS tool sets.

unconditional branch: A branch that does not depend on the truth of any condition. Compare **conditional branch.**

utility: In general, an application program that performs a relatively simple function or set of functions such as copying or deleting files. An 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 an APW utility.

variable: The symbol used in a program to represent a memory location where a value can be stored. Compare **constant.**

volume: An object that stores data; the source or destination of information. A volume has a name and a volume directory with the same name; information on a volume is stored in files. Volumes typically reside in devices; a device such as a floppy disk drive may contain one of any number of volumes (disks).

wildcard character: A symbol that may be used as shorthand to represent any character (or 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.

x flag: One of three flag bits in the 65C816 processor that programs use to control the processor's operating modes. In **native mode**, the setting of the x flag determines whether the index registers are 8-bits wide or 16-bits wide. See also **e flag** and **m flag.**

X register: One of the two index registers in the 65C816 microprocessor.

Y register: One of the two index registers in the 65C816 microprocessor.

zero page: The first page (256 bytes) of memory in a standard Apple II computer (or in the Apple IIGS 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.**