Cortland Workshop Assembler Reference

Pre-Alpha Draft
June 20, 1986
Engineering Part Number: 030-3131
Marketing Part Number: A2L6001

Writer: Catherine Williamson
Apple User Education Department

Revision History

816 Assembler Delta Guide, Paul Black, January 9, 1986.

Document Design, *Cortland Workshop Assembler* , Catherine Williamson, March 17, 1986.

Alpha Draft, *Cortland Workshop Assembler Reference,* Catherine Williamson, June 20, 1986.

Current Software Version

The interface documented in this manual conforms to Phase III of the CPW software. With one exception, the interface described is the Shell's command interpreter, since the graphics interface will not be implemented until Phase IV. The exception is Chapter 2, where I have written some realistic fiction, using the MAC as an analog.

Contents

This page is left intentionally blank

About This Manual

Intended Audience

This manual is intended for all programmers writing Cortland Assembly Language programs. You should be familiar with assembly language programming and the Cortland Development Environment before reading this manual.

This manual assumes that you are familiar with the architecture of 65816 processor, its instruction set and its addressing modes. An excellent reference source for this information is *Programming the 65816 Including the 6502, 65C02 and 65802,* by David Eyes and Ron Lichty.

Roadmap to the Cortland Technical Manuals

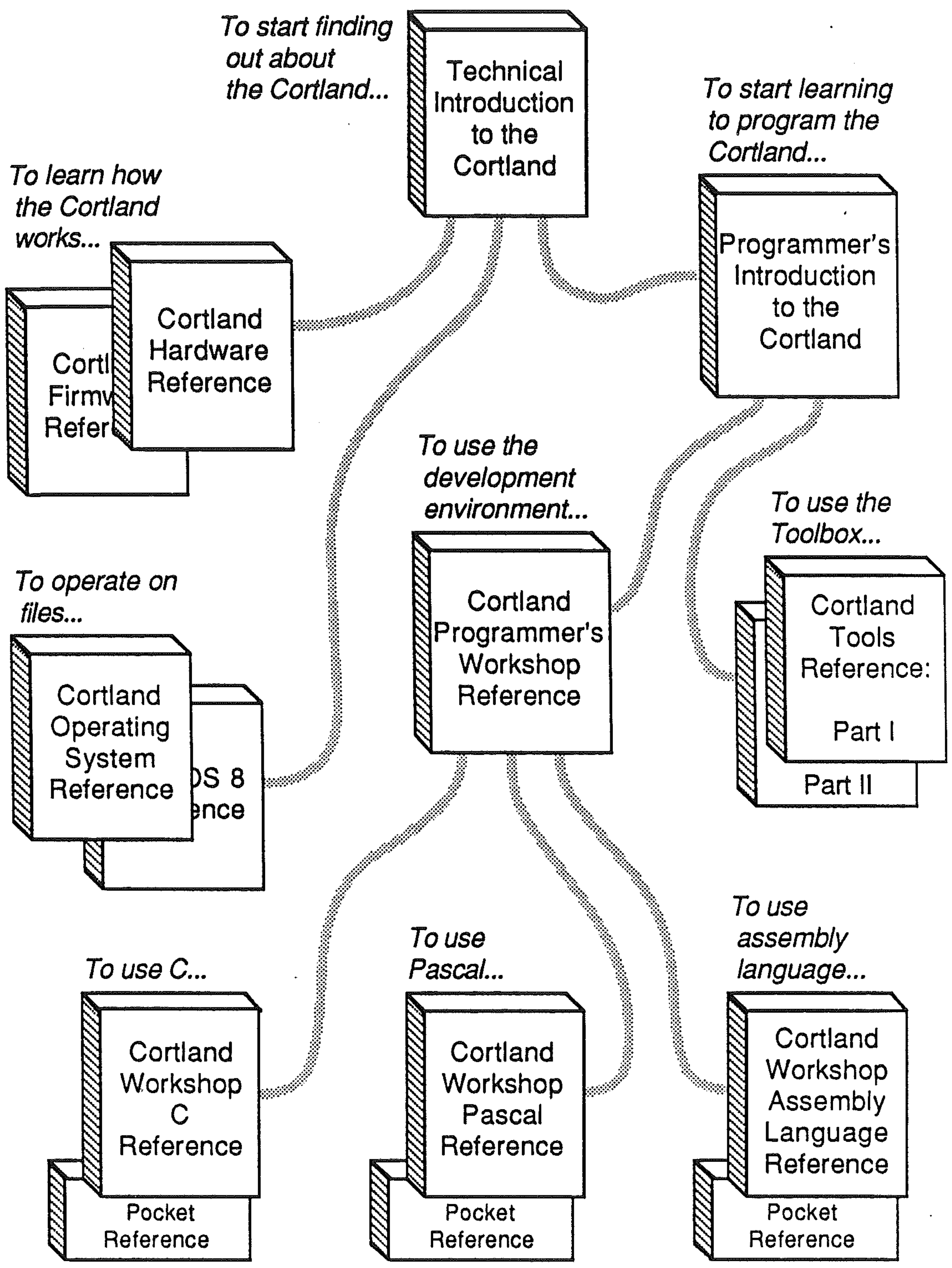
The Cortland has many advanced features, making it more complex than earlier models of the Apple II. To describe it fully, Apple has produced a whole suite of technical manuals. The manuals are listed in the following table. The table is a diagram showing the relationships among the different manuals. Depending on the way you intend to use the Cortland, you may need to refer to a select few of the manuals, or you may need to refer to most of them.

The Cortland Technical Manuals

Title	**Subject**
Technical Introduction to the Cortland	what the Cortland is
Cortland Hardware Reference	machine internals—hardware
Cortland Firmware Reference	machine internals—firmware
Programmer's Introduction to the Cortland	sample program using the toolbox
Cortland Tools Reference: Part I	toolbox specifications
Cortland Tools Reference: Part II	more toolbox specifications
Cortland Function Summary	toolbox pocket guide
Cortland Programmer's Workshop	the development environment
Cortland Workshop Assembler Reference*	using assembly language
Cortland Workshop C Reference*	using C on the Cortland
Cortland Workshop Pascal Reference*	using Pascal on the Cortland
ProDOS/8 Technical Reference	ProDOS for Apple II programs
Cortland Operating System Reference	ProDOS and loader for Cortland
Human Interface Guidelines	for all Apple computers
Apple Numerics Manual	numerics for all Apple computers

*There is a Pocket Reference for each of these.

Figure Preface-1. Roadmap to the technical manuals


To start finding out about the Cortland...
Technical Introduction to the Cortland
To start learning to program the Cortland...
Programmer's Introduction to the Cortland
To learn how the Cortland works...
Cortl Firmw Refer
Cortland Hardware Reference
To use the development environment...
Cortland Programmer's Workshop Reference
To use the Toolbox...
Cortland Tools Reference: Part I
Part II
To operate on files...
Cortland Operating System Reference
OS 8 ence
To use C...
Cortland Workshop C Reference
Pocket Reference
To use Pascal...
Cortland Workshop Pascal Reference
Pocket Reference
To use assembly language...
Cortland Workshop Assembly Language Reference
Pocket Reference

The Technical Introduction

The *Technical Introduction to the Cortland* describes a little about a lot of things, but it doesn't tell everything about anything. To find out all about any one aspect of the Cortland, you should read a specific technical manual. To find out which one, read on.

The Machine Reference Manuals

The *Cortland Hardware Reference* and the *Cortland Firmware Reference* contain information about the machine itself. You don't need to read these manuals to be able to develop applications for the Cortland, but they will give you a better understanding of the machine's features. They will also provide the reasons why some of those features work the way they do.

The Toolbox Manuals

Like the Macintosh, the Cortland has a built-in toolbox that can be called by applications. The toolbox serves two purposes: it makes developing new applications easier, and it supports the desktop user interface.

When you first start using the toolbox, the *Introduction for Programmers* provides the recommendations and guidelines you need. It is not a complete course in programming for the Cortland; rather, it is a starting point. It explains the Cortland tools and describes an event-driven program. It includes a simple example of such a program that uses the Cortland tools, and demonstrates the way you use the Cortland Programmer's Workshop to develop the program.

For detailed specifications of the tool calls, you'll need the two volumes making up the *Cortland Tools Reference*. The *Cortland Function Summary* is a pocket guide to the tools, including the name and parameters for each tool call.

The Cortland Programming Languages

The Cortland does not restrict developers to a single programming language. Apple is currently providing an assembler and compilers for C and Pascal. Other compilers can be used with the workshop, provided that they observe the standards Apple has set up.

There is a separate reference manual for each programming language on the Cortland. The manuals for the languages Apple provides are the *Cortland Assembler Reference*, the *Cortland C Compiler Reference*, and the *Cortland Pascal Compiler Reference*.

The Programmer's Workshop Manual

The core of the development environment on the Cortland is the Cortland Programmer's Workshop, also called CPW. CPW is a set of programs that enable developers to create and debug application programs on the Cortland. The manual that describes CPW is the *Cortland Programmer's Workshop* manual. It includes information about the parts of the workshop that all developers will use, regardless which programming language they use: the shell, the editor, the linker, the debugger, and the utilities.

What About ProDOS?

There are two versions of ProDOS on the Cortland: one for compatibility with the models of Apple II that use 8-bit CPUs, called ProDOS 8, and one that utilizes the full power of the Cortland, ProDOS 16. Those two versions of ProDOS are described in their own manuals, *ProDOS 8 Technical Reference* and *ProDOS 16 Technical Reference*

All-Apple Manuals

In addition to the Cortland manuals mentioned above, there are two manuals that apply to all Apple computers. Those are *Human Interface Guidelines* and *Apple Numerics Manual.*

What's In This Manual

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

Part I, "A Programmer's Reference", introduces you to the Cortland Workshop Assembler and its programming environment.

Chapter 1, "Getting Started", introduces the environment in which you'll use the Assembler. It discusses the Cortland Programmer's Workshop, ProDOS 16, the Cortland Tools, and lists the hardware and software requirements you'll need.

Chapter 2, "Using The Cortland Workshop Assembler", steps through a sample session with the Assembler, describes the assembly process, lists the Shell commands you'll need working with the Assembler, and discusses the Linker, the Debugger and other utilities.

Part II, "Language Reference", is a detailed description of the structure and components of the Cortland Workshop Assembler.

Chapter 3, " Programming the 65816", is intended as a reference chapter to the 65816 registers, instruction set and the addressing modes.

Chapter 4," Coding Conventions", describes the syntax of Cortland Assembly Language.

Chapter 5, "Directives", summarizes the Cortland Assembler directives and provides a summary of the macro directives cross-referenced to the pages in Chapter 6 where they are described. The general Assembler directives are grouped by function, and each directive includes a description, syntax and code example.

Chapter 6, "Macros", consists of two sections: "Using Macros" and "Writing Macros". "Using Macros" takes you through a sample session with macros. It tells you how to build a macro library and describes several assembler directives you will use when working with macro libraries and listing the assembly. "Writing Macros" gives you information on how to include macros in your source text, including macro formats, addressing modes, and data types.

Chapter 7, "The Cortland Libraries", discusses the macro libraries available with the Cortland to access the Cortland Toolbox, make ProDOS 16 calls, make Shell calls, and perform I/O.

Part III includes five appendixes:

Appendix A is a list of the 65816 instruction set, addressing modes, opcodes and execution times.

Appendix B discusses how the Assembler uses memory and includes the Cortland Memory Map.

Appendix C is a comparison of Cortland Assembler and the ORCA/M Assembler.

Appendix D is a chart of the ASCII character set.

Appendix E is a summary of error messages generated by the Cortland Assembler.

Other Materials You Will Need

Anything else?

Visual Cues

Certain conventions in this manual give you information in a visual way, including the introduction of a new term and importand messages such as notes and warnings. These are described in this section. Conventions specific to the way Cortland Workshop Assembly Language is presented in this manual are described under Language Notation.

New terms

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

Notes and Warnings

Special messages of note are marked as such:

- *Note*: Text set off in this manner presents sidelights or interesting points of information.
- **Important**: Text set off in this way, with Important in boldface, presents important information or instructions.

- ***Warning!*** A message such as this lets you know of a potential problem or disaster.

Language Notation

For ease of recognition, instruction, directive and macro mnemonics are presented in upper case in this manual. Note, however, that the Cortland Assembler is case-insensitive.

In common with the Cortland Workshop high-level language manuals, this manual uses different fonts to illustrate certain information:

- English-language text is in Times Roman, as shown:

  Shall I compare thee to a summer's day

- Assembly-language code, both set-off from text and within text is in Courier, as shown:

  ```
  LDA       MSG1,Y
  ```

- Metasymbols used in syntax diagrams and text is in ***TBD***, as shown:

  Example TBD

Additionally, the following conventions are observed:

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

Part I

A Programmer's Reference

Part I, "A Programmer's Reference", introduces you to the Cortland Workshop Assembler and its programming environment.

Chapter 1, "Getting Started", introduces the environment in which you'll use the Assembler. It discusses the Cortland Programmer's Workshop, ProDOS 16, the Cortland Tools, and lists the hardware and software requirements you'll need.

Chapter 2, "Using The Cortland Workshop Assembler", steps through a sample session with the Assembler, describes the assembly process, lists the Shell commands you'll need working with the Assembler, and discusses the Linker, the Debugger and other utilities.

Chapter 1

Getting Started

The Cortland Programmers Workshop

The Cortland Programmer's Workshop (CPW) consists of a suite of software designed to assist developers in writing Cortland applications programs. This development environment includes a **command interpreter**, known as the **Shell**; a text **editor**; a **linker**; a **debugger**; and a set of utilities. Languages supported under CPW in addition to the Assembler include C and Pascal compilers. Further support for developers is provided by a comprehensive set of routines known as the **Cortland Tools**. These Tools are accessed from, but are not considered part of, the Cortland Programmer's Workshop. For a comprehensive description of CPW, refer to the *Cortland Programmer's Workshop* manual. For more information on the Cortland Tools, refer to Chapter 7, "The Cortland Libraries" and *Cortland Tools: Parts I and II.*

About The Cortland Shell

The CPW Shell provides the interface that allows you to work with the Assembler and perform tasks such as file, directory and disk management. The Shell operates in two modalities: a keyboard environment implemented through a command interpreter, and a mouse environment similar to the Macintosh Programmer's Workshop which includes a mouse interface and pull-down menus. With the command interpreter, you type commands in an edit window that allows you to scroll through previous commands and responses. Most tasks can be executed either way, though you can access some of the advanced features, such as programmable **EXEC files**, only through the command interpreter.

The Shell also acts as an extension to ProDOS 16, providing several functions that can be called by programs running under the Shell. The Assembler can call a set of Shell primitives to perform the following functions:

- pass parameters and operations flags between the Shell and Programmer's Workshop programs
- read the current language number
- set the current language number
- set the value of one of the system prefixes
- return the address of the command table
- set the character to be used as the cursor
- get filenames using wildcards
- quit the system and pass control to the next system program

The Cortland Programmer's Workshop provides macros that expand to the Shell calls. Shell macros are discussed in Chapter 7, "The Cortland Libraries".
*** will these macros exist? ***

Commands most often used while working with the Assembler are described in Chapter 2, "Using the Cortland Workshop Assembler". For your convenience, Shell commands you'll use often to work with files, directories and disks are summarized at the end of Chapter 2. For a comprehensive description of the Shell, refer to the *Cortland Programmer's Workshop*.

About the Cortland Workshop Editor

The Cortland Workshop Editor is a full-screen text editor. It can be fully customized and operates under both keyboard control and mouse-based menus.

From the Editor command windows, you can send commands to the Shell to perform tasks such as:

- text manipulation
- text searches
- moving your position in the file
- scrolling the screen
- setting and clearing tab stops
- defining and using keyboard macros

The Cortland Workshop Editor is fully described in the *Cortland Programmer's Workshop*.

About Cortland Workshop Assembler

The Cortland Workshop Assembler is a powerful macro assembler based on the ORCA/M™ Assembler. Features of the Cortland Assembler include:

- A 6502 **emulation mode** that provides software compatibility with existing Apple II programs.
- An extensive group of assembler **directives**.
- A comprehensive set of **macros** to access the Cortland 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 Cortland **object module format**, allowing you to link a **program segment** written in assembly language with segments written in any high-level language.

Modes Of Operation

The Cortland Assembler can operate in either native or emulation mode. In **native mode**, the full instruction set of the 65816 processor is available. The 91 instructions combined with 25 addressing modes make 256 opcodes available to the programmer. The register set can be used for either eight- or sixteen-bit operations. The accumulator can be set to either a sixteen-bit register or two eight-bit registers. The advantage of using a processor with sixteen-bit registers as compared to eight-bit registers is the ability to write programs that are from 25 to 50 percent shorter, and which run 25 to 50 percent faster due to the ease with which sixteen-bit data is manipulated.

In 6502 emulation mode, the Cortland emulates the 6502 processor exactly, including the configuration of the registers, stack location and instruction timing. Thus, the Cortland in emulation mode provides full compatibility with existing Apple II software.

Assembler Directives

The Cortland includes a set of directives that instruct the Assembler to perform a variety of functions. These include:

- Program control
- Data definition
- Symbol definition
- Space allocation
- Assembler options
- ProDos control
- Conditional assembly
- Working with macros

The Assembler directives are described in Chapter 5, "Directives".

Assembler Macros

The Cortland Programmer's Workshop provides a comprehensive set of macros that provide access to the Cortland Toolbox, interface with ProDOS 16 and the Shell, and perform I/O. The Cortland Tools include routines to handle memory management, menu management, **QuickDraw** routines, support for the Standard Apple Numeric Environment (SANE) and more. Cortland Toolbox macros are described in Chapter 7, "The Cortland Libraries". Additionally, the Cortland Programmer's Workshop provides full support for user-defined libraries and macros.

Standard Apple Numeric Environment

The Cortland Assembler provides full support for the Standard Apple Numeric Environment (SANE). SANE is based on the IEEE Standard 754 for Binary Floating-Point Arithmetic [10], 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 libary of high-quality scientific and financial functions. Thus SANE provides a numeric environment sufficient for a wide range of applications.

Object Module Format

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

Cortland System Software

System tasks are handled by ProDOS 16, the **System Loader**, the **Finder,** and the **Memory Manager**. ProDOS 16 is the core, or kernel, of the Cortland's operating system; it provides file management, input/output capability, and control certain other aspects of Cortland's operating system.

The System Loader works closely with ProDOS 16. It is responsible for loading all code and data into the Cortland memory. It is capable of static and dynamic loading and relocating of program segments, data segments, subroutines and libraries.

The Finder has two functions. First, it is a program selector: it lets you launch applications and switch from one application to another without restarting the computer. Second, it is a disk utility program, allowing you to copy, delete, move and rename files and disks.

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

ProDOS 16, the Loader and the Finder are documented in the *Cortland Operating System Reference*. The Memory Manager is documented in both the *Cortland Operating System Reference* and the *Cortland Tools: Parts I and II.*

System Requirements

Hardware Requirements

Using the Cortland Assembler requires at least one megabyte of RAM, two 800K disk drives or one 800K and a hard disk.

*** true? ***

Software Requirements

To use the Cortland Workshop Assembler, you need a System Disk and the Cortland Programmer's Workshop.

The Cortland Workshop Assembler disk includes:

- *** to be supplied ***

The System Disk includes:

- ProDOS 16
- The Finder
- *** other? ***

The Cortland Programmer's Workshop disk includes:

- *** to be supplied ***

This page is left intentionally blank

Chapter 2

Using The Cortland Workshop Assembler

This chapter consists of three sections describing how to use the Cortland Workshop Assembler. The first section, "Writing and running a sample session", steps through a sample session, giving you a fast way to become acquainted with assembling, linking and executing a program. The second section, "The Cortland Workshop Assembler", discusses the assembly process. The third section, "Assembler Shell commands", describes the Shell commands you'll use when working with the Assembler, including building a macro library, making a cross reference file, making a dictionary segment, and command options you can use to assist you in doing partial assemblies.

[writer's note: The original document design called for including this section in Chapter 1. However, it now seems more logical to have Chapter 1 describe the main features of the Assembler and its environment, including CPW, the Shell, the Editor, system software, and Cortland Tools, and have Chapter 2 describe, as its title implies, "Using the Assembler". So, I've put it here. Any comments?]

Writing and Running A Sample Session

This section illustrates how to enter, assemble and link, then run a sample program.

Entering The Program

When you edit an existing program, the editor will already be set to the language in which the file was originally written. When you open a new file, the editor will be set to the last language you used. If necessary, use the command window in the Shell menu to set the editor to the language you wish.

The following session shows you how to edit an existing file, set the current language, create a new file, and save the file.

To edit an existing file:

- Pull down the File menu
- Choose Open from the menu
- Select the file you wish to edit
- Edit the file

To set the current language:

- Pull down the Shell menu

- In the command window, type ASM65816
- Press the Return key

To create a new file:

- Pull down the File menu
- Choose New from the menu
- Enter your program.

The following sample is a small program that prints a simple message.

```
         KEEP      TEST
MAIN     START
         LDX       #MSG2-MSG1
         LDY       #0
LB1      LDA       MSG1,Y
         JSR       COUT
         INY
         DEX
         BNE       LB1
         RTS

COUT     JMP       ($36)

MSG1     DC        C'Hello, world.'
         DC        H'0D'
MSG2     ANOP
         END
```

Finally, to save your program:

- Choose SAVE from the File menu, and give it the filename "MyProg".

Assembling, Linking and Executing the Program

To assemble and link the program you have just entered:

- Pull down the Assemble menu
- Select Assemble, Link and Run from the menu
- Select the file "MyProg"
- Click OK to start the assembly

The Shell checks the language type of your file, and invokes the Assembler. The Assembler then produces an object file, which is in the same object file format as an object file produced by any high-level language. When the Assembler has completed the assembly, it returns control to the Shell, which calls the Linker. The Linker resolves all references, writes the **load file**, then again returns control to the Shell. The Shell then executes the program, and the following message appears on the screen:

"Hello, world."

This concludes the sample session. The sections that follow gives you more detailed information on Assembler options and commands.

The Cortland Workshop Assembler

This section discusses the Assembly process, passes one and two of the Assembler, listing the assembly to the screen and to the printer, stopping the assembly and using the RESET key.

The Assembly Process

The Assembler processes a source program one **program segment**, or subroutine, at a time. Each subroutine goes through two passes. The first pass resolves **local labels.** When the Assembler encounters an END directive or the end of the source file, it passes control to pass two. Lines which appear outside program segments do not contain labels, so they can be completely resolved in pass one.

When pass two is called, it starts at the beginning of the program segment as defined by the START or DATA directives. Pass two then assembles each line for the last time. Local labels have already been resolved in pass one, so pass two can produce both the object code output and the assembly listing. External labels are resolved as $8000, possibly with some offset value. External direct page labels, indicated in the source listing by a < character before the expression, are resolved to $80.

Object code output is in object module format (OMF). Each Cortland Workshop language outputs object code in object module format, allowing you to link subroutines written in different languages together. Object module format is discussed in detail in Chapter 10, "File Formats", of the *Cortland Programmer's Workshop*.

When pass two has finished assembling the subroutine, it prints the local **symbol table**. It then passes control back to pass one to begin the next subroutine. If there are no more subroutines to assemble, the Assembler returns control to the Shell. Depending on the command you used to invoke the Assembler, the Shell passes control either to the Linker or returns with the Shell prompt. If the Linker is called, it uses the object modules produced by the Assembler as input. These are relocated and **global labels** are resolved, giving an executable binary file as output.

The Assembly Listing

The Assembler produces a listing on the screen during pass two unless you instruct it not to list the output. Each segment begins with two messages announcing the subroutine name and pass. The listing continues with the assembled code.

Each output line has four parts: a line number, the current relative address, the code that the Assembler generated, printed in hexadecimal as a sequence of up to four bytes, each byte separated by a space, and lastly the source statement that generated the code.

The first line number is a four-digit decimal number, starting at 0001 on the first line and incrementing for each source line. The line number is incremented even if the output line is not listed. Thus, even if the listing is turned off for part of the assembly, you can still determine how many lines the Assembler has processed. Lines generated by a macro are not considered source lines, so they do not have a line number.

The current relative address is the memory location at which the code would be if the subroutine were placed at location $0000 by the Link Editor. Despite this offset, labels defined relative to the **program counter** within the range zero to $FF are not direct page; the origin of $0000 is simply for convenience in calculation. Internally, the actual origin in the relocatable object module is $1000. The Link Editor outputs indicate where the subroutine is actually located in a given binary file.

Stopping The Assembly

At any time during pass two, you can stop the assembly by pressing any keyboard character. Note that the assembly will stop only if a line or symbol table is being printed, and not for the pass headings which lists the subroutine name. This provides a quick way to scan for errors; by turning off the listing and symbol table, only the output of error lines can stop the listing. Pressing a key at the beginning of the assembly will then stop the listing at the next error. Since the pass headings are still displayed for each subroutine, you can determine which subroutine contains the error.

Assembler Error Messages

If the Assembler detects an error in the source statement, the error is printed on the next line. All error messages are text messages accompanied by a severity level indicator. The error messages are decribed in Appendix E.

If the Assembler encounters a terminal error, such as a symbol table overflow, it returns control to the Shell. The Shell then enters the Editor automatically, and places the line that caused the error at the top of the text edit window. This allows you to identify the line containing the error, even if pass two has not started and no listing has been produced yet.

The RESET key

If the RESET key is pressed during an assembly, control is returned to the Shell. The Shell enters the Editor as if it had encountered a terminal error. The Editor displays the current line, showing where the assembly was interrupted. If this happens during a macro resolution, the line displayed is the macro call statement. If you use the RESET key while a disk write is in progress, you may lost information on the disk.

Printer Listings

Specifying the +L switch with the Shell command you used to assemble your program sends a listing to the printer. Alternatively, you can code the PRINTER directive with the ON option in your source file, in which case subsequent lines are sent to the printer. The Assembler expects an eighty-column printer with an interface card in slot one.

Printed listings are generally the same as listings to the screen, except that the messages announcing the start of various passes are not printed. The assembler assumes sixty-six lines per page, and prints on sixty of those lines. Six lines are skipped after each block of sixty lines to allow for page breaks. After printing the symbol tables for a subroutine, the Assembler skips to the top of the next page.

Assembler Shell Commands

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

- Edit new and existing files
- Assemble, link and execute your program
- Build a macro library file
- Perform a partial assembly
- Make a cross reference symbol table
- Make a dictionary segment, allowing the Linker to scan your library file faster
- Debug your program

Editing Files

There are three Shell commands you will need when you edit a new or existing file. These are:

ASM65816	Change the default language to 65816 assembly language
EDIT	Edit an existing file
NEW	Open a new edit window

When you edit an existing program, the editor will already be set to the language in which the file was originally written. When you open a new file, the editor will be set to the last language you used. If you need to set the language, use the ASM65816 command. To edit an existing file, use the EDIT command. To open a new edit window, use the command NEW.

Assembling A Program

There are three commands you'll need when you are assembling, linking and running your program. These are:

ASSEMBLE	Assembly a program
ASML	Assemble and link a program
ASMLG	Assemble, link and execute a program

ASSEMBLE

ASSEMBLE [+L|-L] [+S|-S] *sourcefile* [KEEP=*outfile*][NAMES=(*seg1*[,*seg2*[,...]])] [*language1*=(*option...*)[*language2*=(*option...*)...]]

This command assembles a source file. The command parameters are described in the section "Assembler Command Options", below.

ASML

ASML [+L|-L] [+S|-S] *sourcefile* [KEEP=*outfile*][NAMES=(*seg1*[,*seg2*[,...]])]
[*language1*=(*option...*)[*language2*=(*option...*)...]]

The command ASML assembles the source file. If the maximum error level found does not exceed the maximum error level allowed, it then links the object module. The command parameters are described below.

ASMLG

ASMLG [+L|-L] [+S|-S] *sourcefile* [KEEP=*outfile*][NAMES=(*seg1*[,*seg2*[,...]])]
[*language1*=(*option...*)[*language2*=(*option...*)...]]

The command ASMLG assembles the source file. If the maximum error level found does not exceed the maximum error level allowed, it links the object module. If the error level is still acceptable, the program is executed.

Assembler Command Options

+L|-L	If you specify +L, the Assembler generates a source listing. If you specify -L, it produces no listing. The default is +L, unless you place a LIST directive with the OFF option in the source file.
+S|-S of	If you specify +S, the Assembler produces a symbol table; the Linker, if it has been invoked, also produces an alphabetical listing all global references in the object module. The Assembler produces an alphabetic listing of all local symbols following each END directive. If you specify -S, these symbol tables are not produced. The default is -S, unless you place a SYMBOL directive with the OFF option in your source file.
sourcefile	The full pathname and filename of the source file.
KEEP=*outfile*	This parameter specifies the name of the output file. For a one-segment program, the output module is named *outfile.root*. If the program contains more than one segment, the first segment is placed in *outfile.root*, then the next segment is placed in *outfile.a*, the following segment in *outfile.b*, and so on, in ascending alphabetical order. If you are performing a **partial assembly**, you can use other filename extensions; refer to the section, "Partial Assemblies", below. If the assembly is followed by a successful link edit, then the load file is named *outfile*.

This parameter has the same effective as placing a KEEP directive in your source file. If you have a KEEP directive in the source file and you also use the KEEP parameter, the KEEP directive takes precedence. In this case, two object modules are produced with the extension *.root;* one corresponding to the parameter and one corresponding to the directive. However, other files with *.a* or other extensions are created with the filename used in the directive only, and the Link Editor uses only the filename given in the KEEP directive.

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

- If you use neither the KEEP parameter nor the KEEP directive, then the object modules are not saved at all. In this case, the link edit cannot be performed, because there is no object module to link.
- The filename you specify as *outfile* must not be over 10 characters long. This is because the extension .root is appended to the name, and ProDOS does not allow filenames of over 15 characters.
- If a file named *outfile* already exists, if is overwritten without a warning when this command is executed.

NAMES=(*seg1*,*seg2*,...)

This parameter instructs the Assembler to perform a partial assembly. The operands *seg1, seg2* and so on specify the names of the segments to be assembled. The Linker automatically selects the latest version of each segment when the program is link edited.

You assign names to segments with START or DATA directives. The object file created when you use the NAMES parameter contains only the specified segments. 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. For example, if you specify the output filename as *outfile* for the original assembly of a program, then the Assembler creates object modules named *outfile.root* and *outfile.a*. In this case, you must also specify the output filename as *outfile* for the partial assembly. The new output file is named *outfile.b*, and contains only the segments listed with the NAMES parameter.

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

Refer to the section "Partial Assemblies or Compiles" in Chapter 4 of the *Cortland Programmer's Workshop* manual for a complete discussion of partial assemblies.

Assembler directives which 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
COPY	Copy a source file
ERR	Print errors
GEN	Generate macro expansions
GEQU	Define a global symbolic constant
IEEE	Enable IEEE format numbers
INSTIME	Show instruction times
KEEP	Keep an output file
LIST	List output
MCOPY	Copy a macro file
MDROP	Drop a macro file
MERR	Maximum error level
MLOAD	Load a macro file
MSB	Set most significant bit
PRINTER	Send output to the printer
RENAME	Rename an op code
SYMBOL	Print symbol tables
65816	Enable 65816 opcodes
65C02	Enable 65C02 opcodes

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 don't follow these rules, an invalid operand error will result.

[*language1=option...*)[*language2=option...*)...]]

This parameter allows you to pass parameters directly to the Cortland compilers. For each compiler, type the name of the language, exactly as defined in the command syntax, an equal sign (=), and the string of parameters enclosed in parentheses. The contents and syntax of the parameter string is specified in the appropriate compiler reference manual. The CPW Shell performs no error checking on this string, but passes it through to the compiler. You can include parameter strings in the command line for as many languages as you wish. If that language compiler is not called, then the string is ignored.

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

Listings and error messages are sent to the command window unless you include a PRINTER directive with the ON option in your source file, or redirect output to another window, disk file, or the printer in the command line.

Important: If you are using a LinkEdit file to take advantage of the advanced link edit capabilities it provides, do not use the ASML command. Instead, use the ASSEMBLE command. You can process the LinkEd file automatically by appending it to the end of your program with an APPEND directive, or you can process it independently with the ALINK command.

Partial Assemblies

If you are writing large programs, you may find that the development process is being slowed down considerably by the amount of time it takes to assemble the program. You can often speed this process up considerably by taking advantage of the CPW's ability to perform partial assemblies using the NAMES option of the Shell commands ASSEMBLE, ASML, or ASMLG. This option is described in the section "Assembler Command Options", above. 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.

The Linker

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

Normally, the Linker is called by the Shell command LINK which provides a limited number of options. Additionally, you can control all functions of the Linker by using a language-like set of commands called **LinkEd.** LinkEd is for advanced programmers who require maximum flexibility from the system; for most purposes, the ordinary Link commands are adequate.

LINK

LINK [+L|-L] [+S|-S] *objectfile* [KEEP=*outfile*]
LINK [+L-L] [+S|-S] (objectfile1,objectfile2,...) [KEEP=outfile]

This command calls the Linker to link edit object modules created by the Assembler to create a load file.

+L|-L

By default, the Linker generates a listing, called a **link map,** of the segments in the object module, including the starting address, the length in bytes (hexadecimal) of each segment, and the segment type. If you specify -L, the link map is not produced.

+S|-S

By default, the Linker produces an alphabetical listing of all global references in the object module, called a **symbol table.** If you specify -S, the symbol table is not produced.

objectfile

The full pathname and filename, minus filename extensions, of the object modules to be linked. All modules to be linked must have the same filename, except for extensions, and must be in the same directory. For example, the program TEST might consist of object modules named test.root, test.a and test.b, all located in my home directory, /CPW/ORSON. In this case, you would use /CPW/ORSON/TEST for objectfile.

(objectfile1,objectfile2...)

You can link several files into one load file with a single LINK command. Enclose in parentheses the full pathnames and filenames, minus filename extensions, of all the object files to be included. Separate the filenames with blanks, tabs or commas. The first file named, *objectfile1*, must have a .root file; for the other object files, the .root file is optional, but the .a file must be present. For example, the program TEST might consist of object modules named TEST1.RROT, TEST1.A, TEST1.B, TEST2.A, TEST2.B, all in directory /CPW/ORSON. In this case, you would use (/CPW/ORSON/TEST1,/CPW/ORSON/TEST2) for *objectfile*.

KEEP=*outfile*

Use this parameter to specify the name of the executable load file. Note that if you do not use the KEEP parameter, then the link edit is performed, but the load file is not saved.

Important: If you do not include any parameters after the LINK command, you are prompted for an input filename, as CPW prompts you for any required parameters. However, since the output pathname and filename are not required parameters, you are not prompted for them. Consequently, the link edit is performed, but the load file is not saved. To save the results of a link edit, you must include the KEEP parameter in the command line.

As an example of the first form of the LINK command, suppose you want to link edit the object file /CPW/TEST1, consisting of files TEST1.ROOT, TEST1.A and TEST1.B. The following command creates the load file /CPW/MYTEST, listing the link map and symbol table:

```
    LINK        /CPW/TEST1  KEEP=/CPW/MYTEST
```

As an example of the second form of the LINK command, suppose you want to link edit the object file /CPW/ORSON/TEST1 consisting of files TEST1.ROOT, TEST1.A and TEST1.B, and the object file /CPW/ORSON/TEST2 consisting of files TEST2.ROOT, TEST2.A and TEST2.B, combining them into a single load file. The following command creates the load file /CPW/MYTEST, printing the link map but suppressing the symbol table:

```
    LINK        -S        (/CPW/TEST1,/CPW/TEST2)          KEEP=MYTEST
```

To automatically link edit a program after assembling it, use one of the commands ASML or AMLG rather than the LINK command. Note that the commands ASML and ASMLG cannot be used to combine object files with different root names into a single load file. To automatically assemble and link two or more object files, use the ASSEMBLE and LINK commands in an EXEC file or with the pipeline operator. For more information on EXEC files and pipelining, refer the Chapter 4, "The Shell", of the *Cortland Programmer's Workshop.*

If you need to take advantage of the advanced link-edit capabilities provided by the Cortland Workshop Linker, create a file of LinkEd commands and process it using the ALINK command or by appending it to the last source file when you assemble your program.

ALINK

The ALINK command calls the Linker to process a file of Advanced Linker (LinkEd) commands. LinkEd commands allow you to do such things as append or insert LinkEd source files in other LinkEd files, place specific object segments in specific load segments, create dynamic or static load segments, set load addresses for nonrelocatable code, search libraries and control the output printed by the Linker. 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. If you do not need to take advantage of the advanced link-edit capabilities provided by LinkEd, do not create a LinkEd file, and do not use the ALINK command. Instead, use either LINK, ASML or ASMLG to link your program. The Linker is described in detail in Chapter 8 of the *Cortland Programmer's Workshop*.

The Debugger

With the Cortland Workshop 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 have specified.

You can load the Debugger at any point in memory: it can execute code in any bank other than the one in which it is resident. The 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 command DUMPOBJ is useful for debugging assembly language code and for whenever you want to see the contents of an object module format (OMF) file. DUMPOBJ dumps an object module to standard output. The default for the file dump is OMF operation codes and records, but you can also list the file as a 65816 machine-language dissassembly ***true?*** or as hexadecimal codes.

Creating A Macro Library

You can search a Cortland **macro library** directly for the macros you reference in your source code. This is not, however, time effective. Using the MACGEN command, you can create a custom **macro library file** containing only those macros needed by your program.

MACGEN

MACGEN [+C] [-C] *infile outfile macrofile1 [macrofile2...]*

The command MACGEN creates a custom macro file by searching one or more macro libraries for the macros referenced in the source code and placing the referenced macros in a single file.

+C|-C	If you specify +C, then all excess blanks are removed from the macro file to save space. If you use the ON option with the GEN directive to include expanded macros in your source file listing, or the ON option with the TRACE directive to include conditional assembly directives in your source

file listing, then use the -C parameter to improve the readability of the listing. +C is the default.

infile	The full pathname and filename of the source file. MACGEN scans infile for references to macros.
outfile	The full pathname and filename of the macro file to be created by MACGEN.
macrofile1 [macrofile2...]	The full pathnames and filenames of the macro libraries to be searched for the macros referenced in *infile*. At least one macro library must be specified. **Wildcards** can be used in the filenames. If you specify more than one filename, separate the names with blanks or commas.

MACGEN scans *infile, i*ncluding all files referenced with COPY and APPEND directives, and builds a list of the macros referenced by the program. It then opens a temporary file called SYSMAC on the work prefix, scans *macrofile1* for referenced macros, and writes the macros it finds to SYSMAC. If there are still unresolved references to macros, MACGEN scans *macrofile2,* and so on. If there are still unresolved references to macros after all the macro files you specfied in the command line have been scanned, then 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, SYSMAC is copied to *outfile.*

The following example scans the program /CPW/MYPROG, searches the macro libraries /CPW/ROM.MACROS, and creates the macro library file /CPW/TEST.MACROS.

```
MACGEN /CPW/MYPROG   /CPW/TEST.MACROS   /CPW/ROM.MACROS
```

Since the macros are written to a temporary file instread of directly to *outfile*, you can specify a previous version of *outfile* as one of the macro libraries to be searched. For example, suppose the program MYPROG already has a custom macro file /CPW/ROM.MACROS. In this case, you could use the following command

```
MACGEN MYPROG   TEST.MACROS TEST.MACROS   /CPW/ROM.MACROS
```

Important: Before you assemble your program, make sure that the source code contains the directive MCOPY *outfile* to instruct the Assembler to search *outfile* for the macros.

Chapter 6, "Macros", of this manual contains a sample session using macros and building a macro library with the MACGEN command.

Making A Cross Reference Table

The XREF utility allows you to make a **cross reference file** of the symbols you have defined and used in your source program. You can also use the XREF utility to count the ocurrences of various operation codes. A language-dependent file must be available on the utility prefix for any language to be scanned. You can redirect output in the normal way.

XREF

XREF [+L|-L] [+X|-X] [+F|-F] [(*subrange1,...,subrange5*)]filename

+L| -L	Using the parameter +L lists the file. Each line of the source file is preceded by a line number. These line numbers are referred to in the cross reference listing. Include a -L to suppress the source listing. The default is +L.
+F|-F	Using the parameter +X lists a frequency count. A frequency count is composed of two lists, each including all the programming language keywords used. The first is in alphabetic order; the second is in decreasing order of the total number of times each keyword is used. The default is -F.
+X|-X	Using the +X parameter generates a cross reference. The cross reference lists each symbol in the source file that is defined within the program; it shows the line or lines on which the symbol is defined, and every line in which the symbol is used. The default is +X

(*subrange1,...subrange5*)

XREF does not recognize symbols as local. That is to say, XREF cross references symbols globally across all segments. To limit the size of a cross reference produced for a large program, you can specify up to five alphabetical subranges to be included in the listing. To specify a subrange, type in a pair of uppercase letters indicating the starting and ending letters with which the symbols can begin. Separate subranges with commas (,), and enclose the set of subranges in parentheses.

The number of symbols that XREF can handle at one time depends on the memory available. If you get a "Symbol Table Overflow" message when you run XREF, try using subranges.

If you omit this parameter, then all symbols are included in the cross reference.

filename The full pathname and filename of the source file to be processed.

The following example show you how to generate a cross reference for the program MYPROG, suppressing the source listing, including a frequency count, and including in the cross reference only symbols that start with the letters A, B, C or S.

Enter the following command line:

```
XREF -L +F    (AC,SS)    MYPROG
```

XREF can process an entire program, including files incorporated with COPY or APPEND directives. It can recognize changes in source code from one file to the next and adjust accordingly.

Important: To process your code with the utility XREF, there must be a file in the /UTILITY subdirectory called XREF.??? ***.??? signifies assembly language***. This file is provided with the Assembler: you must place it in the /Utility subdirectory when you install the Assembler in your copy of CPW.

The following sample cross reference listing illustrates the output from XREF.

include sample

Making a dictionary segment

"Using the MAKELIB command to make a **dictionary segment**" - to be supplied; the implementation and syntax of the MAKELIB command is changing as of the Byte Works review

Chapter 3

Programming the 65816

This chapter discusses the Cortland 65816 processor. It includes a summary of the 65816 registers, the 65816 instruction set, and its **addressing modes**. At the end of the chapter, an instruction list contains the the **opcode** and execution time for each instruction.

This chapter is included as reference material for your convenience. For a comprehensive treatment of the topics summarized here, refer to one of the following publications:

- David Eyes and Ron Lichty, *Programming the 65816*, Simon and Schuster, 1986
- Michael Fisher, *65816/65802 Assembly Language Programming*, Osborne McGraw-Hill, 1896
- William Labiak, *Programming the 65816*, Sybex, 1986

Features of the 65816

The Cortland processor is based on Western Digital's W65SC816 operating together with a custom Fast Processor Interface (FPI) chip. Features of the 65816 include:

- Nine registers, including a 16-bit accumulator, plus sixteen-bit X and Y registers
- Twenty-four-bit internal address bus
- Relocatable **direct page**
- Relocatable stack
- 25 addressing modes and ninety-one instructions, giving 256 opcodes
- Fast block-move instructions
- 6502 emulation, providing software compatibility with existing Apple II programs

65816 Registers

The 65816 processor includes the following registers:

- Accumulators A, B and C
- Index registers X and Y
- Data bank register
- Program bank register
- Direct page register
- Processor status register
- Stack pointer
- Program counter

Accumulators A, B and C

The accumulator is a general-purpose register which stores one of the operands, or the result of most arithmetic or logical operations. In the native mode (e=0), when the accumulator select bit m=0, the accumulator is established as 16-bits wide (A+B=C). When m=1, the accumulator is 8 bits wide (A). In this case, the upper 8 bits (B) may be used for temporary storage in conjunction with the Exchange Accumulator (XBA) instruction.

The Index Registers X and Y

The two index registers X and Y are used as general-purpose registers to provide an index value for calculation of the effective address. When executing an instruction with indexed addressing, the processor fetches the opcode and the base address, and then modifies the address by adding the contents of the index register to the address prior to performing the desired operation. Pre-indexing or post-indexing of indirect addresses may be selected. In native mode (e=0), both index registers are 16-bits wide when the index select bit x=0. When the index select bit x=1, both registers will be 8-bits wide, and the high byte is forced to zero.

Data Bank Register (DBR)

During modes of operation, the 8-bit data bank register holds the default bank address for memory transfers. The 24-bit address is composed of the 16-bit instruction effective address and the 8-bit data bank address. The register value is multiplexed with the data value and is present on the data/address lines during the first half of a data transfer memory cycle. The data bank register is initialized to zero during reset.

Program Bank Register (PBR)

The 8-bit program bank register holds the bank address for all instruction fetches. The 24-bit address consists of the 16-bit instruction effective address and the 8-bit program bank address. The register value is multiplexed with the data value and presented on the data/address lines during the first half of a program memory read cycle. The program bank register is initialized to zero during reset. The PHK instruction pushes the PBR register onto the stack.

Direct Page Register (D)

The 16-bit register provides an address offset for all instructions using direct addressing. The effective bank zero address is formed by adding the 8-bit instruction operand address to the direct page register. The direct page register is initialized to zero during reset.

Processor Status Register (P)

The 8-bit processor status register contains status flags and mode select bits. The carry (C), negative (N), overflow (O), and zero (Z) status flags serve to report the status of most

ALU operations. These status flags are tested by use of conditional branch instructions. The decimal (D), IRQ disable (I), memory/accumulator (M), and index (X) bits are used as mode select flags. These flags are set by the program to change processor operations.

The emulation (E) select and break (B) flags are accessible only through the processor status register. The emulation mode select flag is selected by the exchange carry and emulation bits (XCE) instruction. The M and X flags are always equal to one in the emulation mode. When an interrupt occurs during the emulation mode, the break flag is written to stack memory as bit 4 of the processor status register.

illustration - tbd

Program Counter (C)

The 16-bit program counter provides the addresses which are used to step the processor through sequential program instructions. The register is incremented each time an instruction or operand is fetched from program memory.

Stack Pointer (S)

The stack pointer is a 16-bit register which is used to indicate the next available location in the stack memory area. It serves as the effective address in stack addressing modes are well as subroutine and interrupt processing. The stack pointer allows simple implementation of nested subroutines and multiple-level interrupts. During emulation mode, the stack pointer high-order byte (SH) is always equal to one. The bank address for all stack operations is bank zero.

65816 Instruction Set

This section is divided into two parts. The first, "Instruction Set Summary", groups each instruction by function. The second, "Instruction Descriptions", includes a brief description of each instruction, listed in alphabetical order.

Instruction Set Summary

The 65816 instruction set can be divided functionally into five groups:

- data movement instructions
- flow of control instructions
- arithmetic instructions
- logical and bit manipulation instructions
- system control instructions

Data Movement Instructions

Load/Store Instructions:

LDA	Load accumulator from memory
LDX	Load the X index register
LDY	Load the Y index register
STA	Store the accumulator
STX	Store the X index register
STY	Store the Y index register

Push Instructions:

PHA	Push the accumulator
PHP	Push status register (flags)
PHX	Push X index register
PHY	Push Y index register
PHB	Push data bank register
PHK	Push program bank register
PHD	Push direct page register
PEA	Push effective absolute address
PEI	Push effective indirect address
PER	Push effective relative address

Pull Instructions:

PLA	Pull the accumulator
PLP	Pull status register (flags)
PLX	Pull X index register
PLY	Pull Y index register
PLB	Pull data bank register
PLD	Pull direct page register

Transfer Instructions:

TAX	Transfer A to X
TAY	Transfer A to Y
TSX	Transfer S to X
TXS	Transfer X to S
TXA	Transfer X to A
TYA	Transfer Y to A
TCD	Transfer C accumulator to D
TDC	Transfer D to C accumulator
TCS	Transfer C accumulator to S
TSC	Transfer S to C accumulator
TXY	Transfer X to Y
TYX	Transfer Y to X

Exchange Instructions:

XBA	Exchange B and A accumulators
XCE	Exchange carry and emulation bits

Store Zero to Memory:

STZ	Store zero to memory

Block Moves:

MVN	Move block in negative direction
MVP	Move block in positive direction

Flow Of Control Instructions

BCC	Branch if carry clear
BCS	Branch if carry set
BEQ	Branch if equal
BMI	Branch if minus
BNE	Branch if not equal
BPL	Branch if plus
BRA	Branch always
BRL	Branch always long
BVC	Branch if overflow clear
BVS	Branch if overflow set
JMP	Jump
JSR	Jump to subroutine
JSL	Jump to subroutine long
RTS	Return from subroutine
RTL	Return from subroutine long

Arithmetic Instructions

DEC	Decrement
DEX	Decrement index register X
DEY	Decrement index register Y
INC	Increment
INX	Increment Index register X
INY	Increment index register Y

Logic And Bit Manipulation Instructions

Logic Instructions:

AND	Logical AND
EOR	Logical exclusive-OR
ORA	Logical OR (inclusive OR)

Bit Manipulation Instructions:

BIT	Test bits
TRB	Test and reset bits
TSB	Test and set bits

Shift and Rotate Instructions:

ASL	Shift bits left
LSR	Shift bits left
ROL	Rotate bits left
ROR	Rotate bits right

System Control Instructions

BRK	Break (software interrupt)
RTI	Return from interrupt
NOP	No operation
SEC	Set carry flag
CLC	Clear carry flag
SED	Set decimal mode
CLD	Clear decimal mode
SEI	Set interrupt disable flag
CLI	Clear overflow flag
CLV	Clear overflow flag
SEP	Set status register bits
REP	Clear status register bits
COP	Co-processor of software interrupt
STP	Stop the clock
WAI	Wait for interrupt
WDM	Reserved for expansion

Instruction Descriptions

ADC Add With Carry

Add the data located at the effective address specified by the operand to the contents of the accumulator; add one to the result if the carry flag is set, and store the final result in the accumulator.

AND And Accumulator With Memory

Bitwise logical AND the data located at the effective address specified by the operand with the contents of the accumulator. Each bit in the accumulator is ANDed with the corresponding bit in memory, with the result being stored in the respective accumulator bit.

ASL Shift Memory Or Accumulator Left

Shift the contents of the location specified by the operand left one bit. That is, bit one takes on the value originally found in bit zero, bit two takes the value originally in bit one, and so on; the leftmost bit (bit 7 if m=1 or bit 15 if m=0) is transferred into the carry flag; the righmost bit, bit zero, is cleared. The arithmetic result of the operation is an unsigned multiplication by two.

BCC Branch If Carry Clear

The carry flag in the P status register is tested. If it is clear, a branch is taken; if it is set, the instruction immediately following the two-byte BCC instruction is executed.

BCS Branch If Carry Set

The carry flag in the P status register is tested. If it is set, a branch is taken; if it is clear, the instruction immediately following the two-byte BCS instruction is executed.

BEQ Branch If Equal

The zero flag in the P status register is tested. If it is set, meaning that the last value tested (which affected the zero flag) was zero, a branch is taken; if it is clear, meaning that the value tested was non-zero, the instruction immediately following the two-byte BEQ instruction is executed.

BIT Test Memory Bits Against Accumulator

BIT sets the P status register flags based on the result on two different operations, making it a dual purpose instruction.
First, it sets or clears the n flag to reflect the value of the high bit of the data located at the effective address specified by the operand, and sets or clears the v flag to reflect the contents of the next-to-highest bit of the data addressed.
Second, it logically ANDs the data located at the effective address with the contents of the accumulator; it changes neither value, but sets the z flag if the result is zero, or clears it if the result is non-zero.

BMI Branch If Minus

The negative flag in the P status register is tested. If it is set, the high bit of the value which most recently affected the n flag was set, and a branch is taken.

BNE Branch If Not Equal

The zero flag in the P status register is tested. If it is clear (meaning that the value just test is non-zero), a branch is taken; if it is set (meaning the value tested is zero), the instruction immediately following the two-byte BNE instruction is executed.

BPL Branch If Plus

The negative flag in the P status register is tested. If it is clear, meaning that the last value which affected the zero flag had its high bit clear, a branch is taken. In the two's complement system, values with their high bit clear are interpreted as positive numbers. If the flag is set, meaning that the high bit of the last value was set, the branch is no taken; it is a two's-complement negative number, and the instruction immediately following the two-byte BPL instruction is executed.

BRA Branch Always

A branch is always taken, and no testing is done; in effect, a conditional JMP is executed, but since signed displacements are used, the instruction is only two bytes, rather than the three bytes of a JMP. Additionally, using displacements fron the program counter makes the BRA instruction relocateable. Unlike a JMP instruction, the BRA is limited to targets that lie within the range of the one-byte signed displacement of the conditional branches: minus 128 to plus 128 bytes from the first byte following the BRA instruction.

BRK Software Break

Force a software interrupt. BRK is unaffected by the i interrupt disable flag.

BRL Branch Always Long

A branch is always taken, similar to the BRA instruction. However, BRL is a three-byte instruction; the two bytes immediately following the opcode from a sixteen-bit signed displacement from the program counter. Once the branch address has been calculated, the result is loaded into the program counter, transferring control to that location.

BVC Branch If Overflow Clear

The overflow flag in the P status register is tested. If it is clear, a branch is taken; if it is set, the instruction immediately following the two-byte BVC instruction is executed.

BVS Branch If Overflow Set

The overflow flag in the P status register is tested. IF it is set, a branch is taken; if it is clear, the instruction immediately following the two-byte BVS instruction is executed.

CLC Clear Carry Flag

Clear the carry flag in the status register.

CLD Clear Decimal Mode Flag

Clear the decimal mode flag in the status register.

CLI Clear Interrupt Disable Flag

Clear the interrupt disable flag in the status register.

CLV Clear Overflow Flag

Clear the overflow flag in the status register.

CMP Compare Accumulator With Memory

Subtract the data located at the effective address specified by the operand from the contents of the accumulator, setting the carry, zero and negative flags based on the result, but without altering the contents of either the memory location or the accumulator. That is, the result is not saved. The comparison is of unsigned binary value only.

COP Co-Processor Enable

Execution of COP causes a software interrupt, similarly to BRK, but through the separate COP vector. Alternatively, COP may be trapped by a co-processor, such as a floating-point or graphics processor, to call a co-processor function. COP is unaffected by the i interrupt disable flag.

CPX Compare Index Register X With Memory

Subtract the data located at the effective address specified by the operand from the contents of the X register, setting the carry, aero and negative flags based on the result, but without altering the contents of either the memory location or the register. The result is not saved. The comparison is of unsigned values only (except for signed comparison for equality).

CPY Compare Index Register Y With Memory

Subtract the data located at the effective address specified by the operand from the contents of the Y register, setting the carry, zero, and negative flags based on the result, but without altering the contents of either the memory location or the register. The comparison is of unsigned values only (except for signed comparison for equality).

DEC Decrement

Decrement by one the contents of the location specified by the operand (subtract one from the value).

DEX Decrement Index Register X

Decrement by one the contents of index register X (subtract one from the value). This is a special purpose, implied addressing form of the DEC instruction.

DEY Decrement Index Register Y

Decrement by one the contents of index register Y (subtract one from the value). This is a special purpose, implied addressing form of the DEC instruction.

EOR Exclusive-OR Accumulator With Memory

Bitwise logical Exclusive-OR the data located at the effective address specified by the operand with the contents of the accumulator. Each bit in the accumulator is exclusive-ORed with the corresponding bits in memory, and the result is stored into the same accumulator bit.

INC Increment

Increment by one the contents of the location specified by the operand (add one to the value).

INX Increment Index Register X

Increment by one the contents of index register X (add one to the value). This is a special purpose, implied addressing form of the INC instruction.

INY Increment Index Register Y

Increment by one the contents of index register Y (add one to the value). This is a special purpose, implied addressing form of the INC instruction.

JMP Jump

Transfer control to the address specified by the operand field.

JSL Jump To Subroutine Long (Inter-Bank)

Jump-to-subroutine with long(24-bit) addressing: transfer control to the subroutine at the 24-bit address which 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. It is the return address minus one.

JSR Jump To Subroutine

Transfer control to the subroutine at the location specified by the operand, after first pushing onto the stack, as a return address, the current program counter value, that is, the address of the last instruction byte (the third byte of a three-byte instruction, the fourth byte of a four-byte instruction), not the address of the next instruction.

LDA Load Accumulator From Memory

Load the accumulator with the data located at the effective address specified by the operand.

LDX Load Index Register X From Memory

Load index register X with the data located at the effective address specified by the operand.

LDY Load Index Register Y From Memory

Load index register Y with the data located at the effective address specified by the operand.

LSR Logical Shift Memory Or Accumulator Right

Logical shift the contents of the location specified by the operand right one bit.

MVN Block Move Next

Moves (copies) a block of memory to a new location. The source, destination and length operands of this instruction are taken from the X, Y and C (double accumulator) registers; these should be loaded with the correct values before executing the MVN instruction.

MVP Block Move Previous

Moves (copies) a block of memory to a new location. The source, destination and length operands of this instruction are taken from the X, Y and C (double accumulator) registers; these should be loaded with the correct values before executing the MVP instruction.

NOP No Operation

Executing a NOP takes no action; it has no effect on any registers or memory, except the program counter, which is incremented once to point to the next instruction.

ORA OR Accumulator With Memory

Bitwise logical OR the data located at the effective address specified by the operand with the contents of the accumulator. Each bit in the accumulator is ORed with the corresponding bit in memory. The result is stored into the same accumulator bit.

PEA Push Effective Absolute Address

Push the sixteen-bit operand (typically an absolute address) onto the stack. The stack pointer is decremented twice. This operation always pushes sixteen bits of data, irrespective of the settings of the m and x mode select flags.

PEI Push Effective Indirect Address

Push the sixteen-bit value located at the address formed by adding the direct page offset specified by the operand to the direct page register. The mnemonic implies that the sixteen-bit data pushed is considered an address, although it can be any sixteen-bit data. This operation always pushes sixteen bits of data, irrespective of the settings of the m and x mode select flags.

PER Push Effective PC Relative Indirect Address

Add the current value of the program counter to the sixteen-bit signed displacement in the operand, and push the result onto the stack. This operation always pushes sixteen bits of data, irrespective of the settings of the m and x mode select flags.

PHA Push Accumulator

Push the accumulator onto the stack. The accumulator itself is unchanged.

PHB Push Data Bank Register

Push the contents of the data bank register onto the stack.

PHD Push Direct Page Register

Push the contents of the direct page register onto the stack.

PHK Push Program Bank Register

Push the program bank register onto the stack.

PHP Push Processor Status Register

Push the contents of the processor status register P onto the stack.

PHX Push Index Register X

Push the contents of the X index register onto the stack. The register itself is unchanged.

PHY Push Index Register Y

Push the contents of the Y index register onto the stack. The register itself is unchanged.

PLA Pull Accumulator

Pull the value on the top of the stack into the accumulator. The previous contents of the accumulator are destroyed.

PLB Pull Data Bank Register

Pull the eight-bit value on top of the stack into the data bank register B, switching the data bank to that value. All instructions which reference data that specify only sixteen-bit addresses will get their bank address from the value pulled into the data bank register. This is the only instruction that can modify the data bank register.

PLD Pull Direct Page Register

Pull the sixteen-bit value on top of the stack into the direct page register D, switching the direct page to that value.

PLP Pull Status Flags

Pull the eight-bit value on top of the stack into the processor status register P, switching the status byte to that value.

PLX Pull Index Register X From Stack

Pull the value on the top of the stack into the X index register. The previous contents of the register are destroyed.

PLY Pull Index Register Y From Stack

Pull the value on the top of the stack into the Y index register. The previous contents of the register are destroyed.

REP Reset Status Bits

For each bit set to one in the operand byte, reset the corresponding bit in the status register to zero. For example, if bit three is set in the operand byte, bit three in the status register (the decimal flag) is reset to zero by this instruction. Zeroes in the operand byte cause no change to their corresponding status register bits.

ROL Rotate Memory or Accumulator Left

Rotate the contents of the location specified by the operand left one bit. Bit one takes on the value originally found in bit zero; bit two takes on the value originally in bit one, and so on; the rightmost bit, bit zero, takes the value in the carry flag; the leftmost bit is transferred into the carry flag.

ROR Rotate Memory or Accumulator Right

Rotate the contents of the location specified by the operand right one bit. Bit zero takes on the value originally found in bit one; bit one takes the value originally in bit two, and so on; the leftmost bit takes the value in the carry flag; the rightmost bit, bit zero, is transferred into the carry flag.

RTI Return From Interrupt

Pull the status register and the program counter from the stack. If the processor is set to native mode (e=0), also pull the program bank register from the stack.

RTL Return From Subroutine Long

Pull the program counter (incrementing the stacked, sixteen-bit value by one before loading the program counter with it), then the program bank register from the stack.

RTS Return From Subroutine

Pull the program counter, incrementing the stacked, sixteen-bit value by one before loading the program counter with it.

SBC Subtract With Borrow From Accumulator

Subtract the data located at the effective address specified by the operand from the contents of the accumulator; subtract one more if the carry flag is clear, and store the result in the accumulator.

SEC Set Carry Flag

Set the carry flag in the status register.

SED Set Decimal Mode Flag

Set the decimal mode flag in the status register.

SEI Set Interrupt Disable Flag

Set the interrupt disable flag in the status register.

SEP Set Status Bits

For each one-bit in the operand byte, set the corresponding bit in the status register to one. For example, if bit three is set in the operand byte, bit three in the status register (the decimal flag) is set to one by this instruction. Zeroes in the operand byte cause no change to their corresponding status register bits.

STA Store Accumulator To Memory

Store the value in the accumulator to the effective address specified by the operand.

STP Stop The Processor

During the processor's next phase 2 clock cycle, stop the processor's oscillator input; the processor is effectively shut down until a reset occurs (until the RES pin is pulled low).

STX Store Index Register X To Memory

Store the value in index register X to the effective address specified by the operand.

STY Store Index Register Y To Memory

Store the value in index register Y to the effective address specified by the operand.

STZ Store Zero To Memory

Store zero to the effective address specified by the operand.

TAX Transfer Accumulator To Index Register X

Transfer the value in the accumulator to index register X. If the registers are different in size, the nature of the transfer is determined by the destination register. The value in the accumulator is not changed by the operation.

TAY Transfer Accumulator To Index Register Y

Transfer the value in the accumulator to index register Y. If the registers are different in size, the nature of the transfer is determined by the destination register. The value in the accumulator is not changed by the operation.

TCD Transfer 16-Bit Accumulator To Direct Page Register

Transfer the value in the sixteen-bit accumulator C to the direct page register D, regardless of the setting of the accumulator/memory mode flag.

TCS Transfer Accumulator To Stack Pointer

Transfer the value in the accumulator to the stack pointer S. An alternate mnemonic is TAS (transfer the value in the A accumulator to the stack pointer).

TDC Transfer Direct Page Register To 16-Bit Accumulator

Transfer the value in the sixteen-bit direct page register D to the sixteen-bit accumulator C, regardless of the setting of the accumulator/memory mode flag. An alternate mnemonic is TDA (transfer the value in the direct page register to the A accumulator).

TRB Test And Reset Memory Bits Against Accumulator

Logically AND together the complement of the value in the accumulator with the data at the effective address specified by the operand. Store the result at the memory location.

TSB Test And Set Memory Bits Against Accumulator

Logically OR together the value in the accumulator with the data at the effective address specified by the operand. Store the result at the memory location.

TSC Transfer Stack Pointer To 16-Bit Accumulator

Transfer the value in the sixteen-bit stack pointer S to the sixteen-bit accumulator C, regardless of the setting of the accumulator/memory mode flag.

TSX Transfer Stack Pointer To Index Register X

Transfer the value in the stack pointer S to index register X. The stack pointer's value is not changed by the operation.

TXA Transfer Index Register X To Accumulator

Transfer the value in index register X to the accumulator. If the registers are of different sizes, the nature of the transfer is determined by the destination (the accumulator). The value in the index register is not changed by the operation.

TXS Transfer Index Register X To Stack Pointer

Transfer the value in index register X to the stack pointer S. The index register's value is not changed by the operation.

TXY Transfer Index Registers X To Y

Transfer the value in index register X to index register Y. The value in index register X is not changed by the operation. Note that the two registers are never different sizes.

TYA Transfer Index Register Y To Accumulator

Transfer the value in index register Y to the accumulator. If the registers are different sizes, the nature of the transfer is determined by the destination (the accumulator). The value in the index register is not changed by the operation.

TYX Transfer Index Registers Y To X

Transfer the value in index register Y to index register X. The value in index register Y is not changed by the operation. Note that the two registers are never different sizes.

WAI Wait For Interrupt

Pull 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

This opcode is reserved for future expansion.

XBA Exchange The B And A Accumulators

XBA exchanges the contents of the low-order and high-order bytes of the sixteen-bit accumulator C, where B represents the high-order byte and A represents the low-order byte of the accumulator.

XCE Exchange Carry And Emulation Bits

This instruction allows you to shift between 6502 emulation mode and sixteen-bit native mode. The emulation mode is used to provide software compatibility with programs previously written for the Apple II family of processors.

Addressing Modes

Twenty five addressing modes are available with the 65816 processor. These are summarized below. Not all addressing modes are available for all instructions, but each instruction provides a separate opcode for each of the addressing modes it supports. In Appendix A, each instruction is listed by mnemonic with its opcode and execution time.

Summary of Addressing Modes

Addressing Mode	Syntax Opcode	Example Operand
Implied	DEX	
Accumulator	ASL	A
Immediate	LDA	#55
Program Counter Relative	BEQ	LABEL12
Program Counter Relative Long	BRL	JMPLABEL
Stack	PHA	
Stack Relative	LDA	3,S
Stack Relative Indirect Indexed With Y	LDA	(5,S),Y
Block Move	MVP	0,0
Absolute	LDA	$2000
Absolute Indirect	JMP	($1020)
Absolute Indexed With X	LDA	$2000,X
Absolute Indexed With Y	LDA	$2000,Y
Absolute Indexed Indirect	JMP	($2000,X)
Absolute Long	LDA	$02F000
Absolute Long Indexed With X	LDA	$12D080,X
Absolute Indirect Long	JMP	($2000)
Direct Page	LDA	$81
Direct Page Indirect	LDA	($55)
Direct Page Indexed With X	LDA	$55,X
Direct Page Indexed With Y	LDA	$55,Y
Direct Page Indirect Indexed with Y (Postindexed)	LDA	($55),Y
Direct Page Indexed Indirect With X (Preindexed)	LDA	($55,X)
Direct Page Indirect Long	LDA	($55)
Direct Page Indirect Long Indexed With Y	LDA	($55),Y

Addressing Mode Descriptions

Implied

Implied addressing uses a single-byte instruction. The operand is implicitly defined by the instruction.

Accumulator

This form of addressing always uses a single-byte instruction. The operand is the accumulator.

Immediate Addressing

The operand is the second byte (second and third bytes when in the 16-bit mode) of the instruction.

Program Counter Relative

This addressing mode, referred to as relative addressing, is used only with branch instructions. If the condition being tested is met, the second byte of the instruction is added to the program counter, which has been updated to point to the opcode of the next instruction. The offset is a signed 8-bit quantity in the range from minus 128 to plus 127. The program bank register is not affected.

Program Counter Relative Long

This addressing mode, referred to as relative long addressing, is used only with the unconditional branch long instruction (BRL) and the push effective relative instruction (PER). The second and third bytes of the instruction are added to the program counter, which has been updated to point to the opcode of the next instruction. With the branch instruction, the program counter is loaded with the result. With the push effective relative instruction, the result is stored on the stack. The offset is a signed 16-bit quantity in the range from minus 32768 to plus 32767. The program bank register is not affected.

Stack

Stack addressing refers to all instructions that push or pull data from the stack, such as Push, Pull, Jump to Subroutine, Return from Subroutine, Interrupts, and Return from Interrupt. The bank address is always zero. Interrupt vectors are always fetched from bank 0.

Stack Relative

The low-order 16 bits of the effective address is formed from the sum of the second byte of the instruction and the stack pointer. The high-order 8 bits of the effective address is always zero. The relative offset is an unsigned 8-bit quantity in the range from 0 to 255.

illustration tbs

Stack Relative Indirect Indexed With Y

The second byte of the instruction is added to the stack pointer to form a pointer to the low-order 16-bit base address in bank 0. The data bank register contains the high-order 8 bits of the base address. The effective address is the sum of the 24-bit base address and the Y index register.

illustration tbs

Block Move

This addressing mode is used by the block move instructions. The second byte of the instruction contains the high-order 8 bits of the destination address. The Y index register contains the low-order 16 bits of the destination address. The third byte of the instruction contains the high-order 8 bits of the source address. The X index register contains the low-order 16 bits of the source address. The C accumulator contains one less than the number of bytes to move. The second byte of the block move instructions is also loaded into the data bank register.

illustration tbs

Absolute

With absolute addressing, the second and third bytes of the instruction form the low-order 16 bits of the effective address. The data bank register contains the high-order bits of the operand address.

illustration tbs

Absolute Indirect

The second and third bytes of the instruction form an address to a pointer in bank 0. The program counter is loaded with the first and second bytes at this pointer. With the Jump Long (JML) instruction, the program bank register is loaded with the third byte of the pointer.

illustration tbs

Absolute Indexed With X

The second and third bytes of the instruction are added to the X index register to form the low-order 16 bits of the effective address. The data bank register contains the high-order 8 bits of the effective address.

illustration tbs

Absolute Indexed With Y

The second and third bytes of the instruction are added to the Y index register to form the low-order 16 bits of the effective address. The data bank register contains the high-order 8 bits of the effective address.

illustration tbs

Absolute Indexed Indirect

The second and third bytes of the instruction are added to the X index register to form a 16-bit pointer in bank 0. The contents of this pointer are loaded in the program counter. The program bank register is not changed.

illustration tbs

Absolute Long

The second, third and fourth byte of the instruction form the 24-bit effective address.

illustration tbs

Absolute Long Indexed With X

The second, third and fourth bytes of the instruction form a 24-bit base address. The effective address is the sum of this 24-bit address and the X index register.

Absolute Indirect Long

To be supplied

Direct Page

The second byte of the instruction is added to the direct page register (D) to form the effective address. An additional cycle is required when the direct page register is not page aligned (DL not equal to 0). The bank register is always zero.

illustration tbs

Direct Page Indirect

The second byte of the instruction is added to the direct page register to form a pointer to the low-order 16 bits of the effective address. The data bank register contains the high-order 8 bits of the effective address.

illustration tbs

Direct Page Indexed With X

The second byte of the instruction is added to the sum of the direct page register and the X index register to form the 16-bit effective address. The operand is always in bank 0.

illustration tbs

Direct Page Indexed With Y

The second byte of the instruction is added to the sum of the direct page register and the Y index register to form the 16-bit effective address. The operand is always in bank 0.

illustration tbs

Direct Page Indirect Indexed with Y (Postindexed)

This addressing mode is often referred to as Indirect Y. The second byte of the instruction is added to the direct page register (D). The 16-bit contents of this memory location is then combined with the data bank register to form a 24-bit base address. The Y index register is added to the base address to form the effective address.

illustration tbs

Direct Page Indexed Indirect With X (Preindexed)

This addressing mode is often referred to as Indirect X. The second byte of the instruction is added to the sum of the direct page register (D) and the X index register. The result points to the low-order 16 bits of the effective address. The data bank register contains the high-order 8 bits of the effective address.

illustration tbs

Direct Page Indirect Long

The second byte of this instruction is added to the direct page register to form a pointer to the 24-bit effective address.

illustration tbs

Direct Page Indirect Long Indexed With Y

With this addressing mode, the 24-bit base address is pointed to by the sum of the second byte of the instruction and the direct page register. The effective address is this 24-bit base address plus the Y index register.

illustration tbs

Chapter 4
Coding Conventions

This chapter describes the conventions and syntax for coding Cortland Workshop assembly language programs.

Source Text Structure

An executable assembly language program contains a number of segments containing either code or data. Data segments represent static data, where the data space is defined before the program begins and the lifetime of the data is that of the entire execution of the program. In contrast, code segments represent dynamic data. Dynamic segments can be loaded and unloaded during execution as needed. In your source text, a code segment is delimited by the directives START or PRIVATE and END. A data segment is delimited by the directives DATA or PRIVDATA and END. The segment types PRIVATE and PRIVDATA work just like START and DATA, except that they are not noticed unless the segment is placed in a library or separate compilation is used. In that case, the segment is accessible from the segments in its own object module file, but not visible outside. Segments in different files can have the same names as long as all but one is private. For programmers familiar with the C language, this mimics the C static function.

Used in a code segment, the USING directive designates a specific data segment that it must access. The directives above are described in Chapter 5, "Directives".

Each segment consists of one or more source statements which the Assembler interprets and processes, one at a time, generating object code or performing a specific assembly-time process. Cortland Workshop Assembler source statements may be any of the following:

- 65816 instructions
- Assembler directives
- Macro calls

Instructions

The 65816 instruction set can be divided functionally into five groups:

- data movement instructions
- flow of control instructions
- arithmetic instructions
- logical and bit manipulation instructions
- system control instructions

This manual assumes that you are familiar with the instruction set of the 65816 processor. For your reference, each 65816 instruction is described in Chapter 3, "Programming the

65816". Additionally, Appendix A lists the 65816 instructions and addressing modes with the hex opcode and execution time for each.

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

Directives

Assembler directives guide the assembly process and provide tools for using the instructions. There are two classes of assembler directives: general assembler directives and macro directives.

The format of an Assembler directive is similar to that of an assembly language instruction: functionally, however, they are different. While an instruction corresponds to a machine language instruction, and tells the computer to take some action in the finished program, a directive tells the assembler itself to do something. An example of this is the KEEP directive which instructs the assembler to keep the object module created under the name provided in the operand field.

General Assembler Directives

General assembler directives can be used to perform the following operations:

- Store data or reserve memory for data storage.
- Control the alignment of parts of the program in memory.
- Specify the methods of accessing the sections of memory in which the program will be stored.
- Specify the entry point of the program or a part of the program.
- Specify the way in which symbols will be referenced.
- Specify that a part of the program is to be assembled only under certain conditions.
- Control the format and content of the listing file.
- Display informational messages.
- Control the assembler options that are used to interpret the source program.

The general Assembler directives are described in detail in Chapter 5, "Directives".

Macro Directives

There are a number of directives that are perform certain functions in association with macros:

- Macro language directives, used in macro definitions
- Macro library directives, used when working with macro libraries
- Conditional assembly directives, used to define symbolic parameters, assign new values to them, and control code execution
- Directives used to set specific options in the assembly listing

Macro language directives are used in macro definitions and are valid only within a macro file. Macro library directives are used to include a macro library file in your source code, and load or drop a macro library file. Conditional assembly directives are used to define symbolic parameters, assign new values to them, and to modify the order in which statements are processed by the Assembler. A conditional assembly directive is valid in a source file, but is included under Macro Directives since its main use is within a macro definition. Finally, there are two directives, TRACE and GEN, which set certain options in your listing. They are included under Macro Directives since their main use is with macros. The macro directives are described in detail in Chapter 6, "Macros".

Macros

A macro is expanded by the assembler to produce one or more instructions or directives, allowing you perform complex tasks with a single statement. Chapter 6, "Macros", tells you about using a macro in an assembly language program, introduces a sample session, includes a reference section describing the macro directives and contains a section on how to write your own macros. Chapter 7, "The Cortland Libraries", describes the extensive set of macros that are available on the Cortland, including macros to access the Cortland Tools, macros to interface with the Shell and ProDOS 16, and macros to perform I/O.

Source Statement Format

An assembly language statement can consist of up to four fields:

```
[label]  operator    operand      [;comment]
```

The label field symbolically defines a location in a program. The operator, or opcode field, specifies the action to be performed by the statement. This field can be an instruction, an assembler directive, or a macro call. The operand field contains the instruction operand(s) or the assembler directive argument(s) or the macro argument(s). The comment field contains a comment that explains the meaning of the statement. This field does not affect program exection.

Source Statement Line Length

An assembler source file line can be up to eighty columns long, numbered from one to eighty. Since most printers use eighty columns, it is advisable to restrict the source line to fifty seven columns, as twenty three columns must be allowed for information printed by the Assembler. Otherwise, printed assembler output will wrap around to the next line. This makes the listing difficult to read, and causes the Assembler to miscount the number of lines of printed output, misplacing future page breaks.

Labels

Labels serve the same purpose as line numbers in BASIC, giving you a way of telling the assembler what line you want to branch to or change. The label is optional. Each line may begin with a label, which is required for a few directives. When the label is required, the directive description in Chapter 5 includes that information. The label must begin in column 1, and cannot contain imbedded blanks. Each label starts with an alphabetic character (A to Z) or underscore (_), and is followed by zero or more characters which can be can be alphabetic (A through Z), numeric (0 through 9), the tilde character (~), or the underscore character (_). It is suggested that you reserve the underscore character (~) for use in system labels, so that you can develop libraries whose names will not interfere with names chosen by users of high-level languages.

Labels are significant up to 255 characters in length. If a label is used, there must be at least one space between it and the op code. It is best not to use A as a label, since it can cause confusion between absolute addressing using the label A and accumulator addressing.

Label Scope

A label may be either global or local in scope. 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 glocal label, but not in the same segment. The Assembler will choose the local label in preference to the global label.

Global Labels

There are three directives which define global labels: the START directive, the GEQU directive and the ENTRY directive.

A label defined by the GEQU directive can be seen at assembly time, while all other global labels can be seen only at link time. This implies that you should always use a GEQU directive to define a direct page or long address label that will be used in more than one subroutine. That way, the Assembler can automatically decide which addressing mode is more appropriate.

The ENTRY directive has no operand and its label receives the value of the current location counter. Its most common use is to define an alternate entry point into a subroutine, as shown in the following example:

Example TBS

Local Labels

Inside a code segment, all labels that are not defined using the GEQU or ENTRY directives are local labels. This means that no other code segment can see the label. For example, the following code example would produce an error in SEG1 because LAB1 is not defined in SEG1, but it is legal for both segments to use LAB2.

```
SEG1      START
LAB2      LDA          LAB1
          END

SEG2      START
LAB2      LDA          LAB2
```

```
LAB1     LDA        LAB1
         END
```

The concept of local labels is powerful, and rare in other assemblers. Because a section of code can be developed independently of all other code in the program, you can build a library of subroutines that can be moved from one program to another. And, unlike other assemblers, you don't have to worry about whether you have used a particular label elsewhere in the program. A label can be used in each segment in the program, but only once.

Case-Sensitivity in Labels

You can make a label case-sensitive by specifying the ON option with the CASE directive. CASE OFF reverses the effect. The directive OBJCASE ON causes labels sent to the object module 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 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 behaviour is to be different from the local behaviour, you must specify the OBJCASE directive last.

Label Attributes

The **attribute** of a label or **symbolic parameter** gives you certain information in addition to its value. Attributes can be thought of as functions that return information about a label or the symbolic parameter.

The form of an attribute is

X: Label or Parameter

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
S	Settings attribute

The count attribute gives the number of subscripts defined for a symbolic parameter. The length attribute is the number of bytes generated by the line that defined the label. The type attribute indicates the type of operation in the line that defined the label. The settings attribute returns the current setting of one of the flags set using directives whose operand is ON or OFF.

Using attributes is discussed in detail in Chapter 6, "Macros".

The Operation Code

The operation code, or opcode, is required on each Cortland Assembler source statement. The opcode mnemonic signifies the function the statement performs, such as JSR for the jump-to-subroutine instruction, or KEEP for the keep directive. Leave at least one space between the label and the opcode. If there is no label, the op code can start in any column from two to forty. Normally, the opcode starts in column ten. The editor has a tab stop set to this column for convenient placement.

Opcode mnemonics for machine-language instructions are always three-character alphabetic strings. The opcodes for the 65816 instruction set are listed in both Chapters 3, "Programming the 65816" and Appendix A, "65816 Instruction List".

If you are running in 6502 emulation mode, the Assembler allows substitutions for the following standard 6502 opcodes:

Standard	**Also Allowed**
BCC	BLT
BCS	BGE
CMP	CPA

Asembler directives vary in length from two to seven characters. The opcodes for Assembler directives are listed in Chapter 5, "Directives".

Operands

The operand is the information that the operator uses to perform its function. There must be at least one space between the opcode and the operand. The operand normally starts in column sixteen; the editor provides a tab stop there for convenient placement.

The following table shows all the legal operands of the 65816 processor. The labels DP, ABS and LONG refer to constant expressions that resolve to one, two or three bytes respectively. EXT is a relocatable expression.

Addressing Mode	**Operand Format**
Implied	none needed
Immediate	#DP #>ABS #<ABS #^ABS #ABS #>LONG #<LONG #^LONG #LONG /ABS /LONG
Direct Page	DP <EXT
Absolute	|DP ABS EXT

Absolute Long	>DP >ABS LONG >EXT
Relative	ABS EXT
Direct Page Indexed	DP,X DP,Y <EXT,X <EXT,Y
Absolute Indexed	|DP,X |DP, Y ABS,X ABS,Y EXT,.X EXT,Y
Absolute Long Indexed	>DP,X >ABS,X >EXT,X LONG,X
Absolute Indirect	(ABS)
Direct Page Indirect	(DP) (<EXT)
Direct Page Indirect Long	[DP] [<DP]
Direct Page Indirect Indexed	(DP),Y (<EXT),Y
Direct Page Indirect Indexed Long	[DP],Y [<EXT],Y
Direct Page Indexed Indirect	(DP,X) (<EXT,X]
Absolute Indexed Indirect	(DP,X) (ABS,X) (EXT,X)
Stack Relative	DP,S <EXT,S
Stack Relative Indirect Indexed	(DP,S),Y (<EXT,S),Y
Accumulator	A

Block Move	EXT,EXT DP,DP ABS,ABS LONG,LONG

Expressions

Whenever a number is allowed in an operand field, whether in an instruction or a directive, an expression may be used. In their most general form, expressions can resolve to an integer in the range -2147483648 to 2147483647. The result of a logical operation is always zero or one, corresponding to false and true. If an arithmetic value is used in an Asembler directive which expects a boolean result, zero is treated as false and any other value is treated as true.

If all the terms in an expression are constants, i.e., they are number or labels whose value is set by the EQU or GEQU directives, then the Assembler can determine the final value of the expression without the aid of the Link Editor. In that case, the expression is a constant expression. If any term in the expression is a label that must be relocated, the expression itself must also be relocated. This distinction is important, since the Assembler is able to automatically select between addressing modes that offer one, two and three byte variations only if the expression is a constant expression. In the case of a relocatable expression, the Assembler will always opt for the two byte form of the address, unless it is explicity overridden. The length of addressing used can be forced by using a < before the expression to force direct page addressing, to force absolute addressing, and a > to force long addressing. This is illustrated in the operand format table, above. Note that an exclamation mark (!) can be used instread of the character |, if the keyboard does not support |.

Operands for immediate addressing are resolved to one or two bytes. It is necessary to be able to select which byte or bytes to use from an expression. Three operators are provided to select the appropriate bytes from the value. These operators must appear immediately after the # character, which indicates immediate addressing. If no operator is used, the least significant byte or bytes is used. This also happens if the < operator is used. The > operator has the effect of dividing the expression value by 256, selecting the next most significant byte. FInally, the ^ operator divides the expression by 65536, moving the bank byte into the least significant byte position.

Syntactically, an expression is a simple expression, or two simple expressions separated by a logical comparison operator.

illustration of expression

Thus, logical comparisons have the lowest priority. A simple expression is the customary arithmetic expression. Syntactically, this is expressed as an optional leading sign, a term, and, optionally, a plus (+), minus (-), .OR., or .EOR., followed by another term.

illustration of simple expression

A term is a factor, optionally followed by one of the operators *, /, .AND., or the bit shift operator (|) and another 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, the exclamation mark (!), 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).

illustration of a term

A factor is a constant, label, expression enclosed in parentheses, or a factor preceded by .NOT.. .NOT. is the boolean negation, producing true (one) if the following factor is false, and false (zero) if it is true. Here, a label refers to a named symbol which 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.

illustrations of factor, constant, binary number, octal number, decimal number, hexadecimal number, and character constant

The Comment Field

There must be at least one space between the operand or op code, if there is no operand, and the comment. Some assemblers require a semi-colon before the comment; the Cortland Workshop Assembler does not. Comment usually start in column 41. The editor has a tab stop there for convenient placement. Additionally, the Assembler recognizes certain lines as comment lines, including a blank line, a line beginning with an asterisk, semicolon or exclamation point, or a line beginning with a period.

The Blank Line

Any blank line is treated as a comment line. Blank lines are often used to logically separate sections of code.

*The Characters *, ;, and !*

Any line with an asterisk (*), a semicolon (;), or an exclamation mark (!) in column one is treated as a comment. Any text in the line is ignored by the Assembler, but is printed when it generates the source listing.

The Period

Any line with a period (.) in column one is treated as a comment, and is known as a sequence symbol. Sequence symbols are not printed in the source listing produced by the assembler. They are intended for use as labels for conditional assembly branches. If you decide to use this form of comment to get a line that shows up in the editor, but not later in the listing, place a space after the period.

This page is left intentionally blank

Chapter 5

Directives

A directive is a statement that tells the Cortland Workshop Assembler to take some action. Here, directives are divided functionally into two groups: the general Assembler directives and directives used in working with macros. The general Assembler directives are summarized below and are described in detail in this chapter. The macro directives are summarized below: they are described in detail in Chapter 6, "Macros".

General Assembler Directives

The general Assembler directives perform tasks such as:

- Program control
- Defining data
- Defining symbols
- Allocating memory
- ProDOS control
- Setting Assembler options

These directives are summarized below.

Directive	**Action**
Program Control Directives	
START	Start subroutine
PRIVATE	Define private code segment
DATA	Define data segment
PRIVDATA	Define private data segment
USING	Using data segment
ENTRY	Define entry point
END	End program segment
Data Definition Directives	
DC	Declare constant
DS	Declare storage
Symbol Definition Directives	

EQU	Equate
GEQU	Global equate
RENAME	Rename opcodes

Memory Designation Directives

ALIGN	Align to a boundary
ORG	Designate origin
MEM	Reserve memory

File Control Directives

APPEND	Append a file
COPY	Copy a file
KEEP	Keep object module

Assembler Option Directives

IEEE	Generate IEEE format numbers
LONGA	Select accumulator size
LONGI	Select index register size
MSB	Set the most significant bit of characters
65C502	Enable 65C02 code
65816	Enable 65816 code
MERR	Set the maximum error level
CASE	Specify case-sensitivity
OBJCASE	Specify case-sensitivity in object module

Listing Option Directives

ERR	Print errors
EXPAND	Expand DC statements
LIST	List output
PRINTER	Send output to printer
SYMBOL	Print symbol tables
EJECT	Eject the page
SETCOM	Set comment column
TITLE	Print header
ABSADDR	Allow absolute addresses
INSTIME	Show instruction times

Macro Directives

This section summarizes the directives you'll use when working with macros. You can use macro directives to perform the following tasks:

- Write a macro definition
- Use macro libraries

- Use macros in conditional assemblies
- Set listing options

The macro directives are described in Chapter 6, "Macros", in the sections where they are used. This section is provided as a summary and cross reference to the page where the directive is described.

Directive	Action	Page Cross Reference
Macro Language Directives		
MACRO	Start a macro definition	Page 6-4
MNOTE	Macro note	Page 6-5
MEXIT	Exit macro	Page 6-5
MEND	End a macro definition	Page 6-4
Macro Library Directives		
MCOPY	Copy Macro Library	Page 6-2
MDROP	Drop A Macro Library	Page 6-2
MLOAD	Load A Macro Library	Page 6-3
Listing Directives		
GEN	Generate macro expansions	Page 6-3
TRACE	Trace macros	Page 6-3
Conditional Assembly Directives		
Defining Parameters		
LCLA	Define a local arithmetic symbolic parameter	Page 6-12
LCLB	Define a local boolean symbolic parameter	Page 6-12
LCLC	Define a local string symbolic parameter	Page 6-12
GLBA	Define a global arithmetic symbolic parameter	Page 6-12
GLBB	Define a global boolean symbolic parameter	Page 6-13
GLBC	Define a global string symbolic parameter	Page 6-13
SETA	Assign a value to an arithmetic symbolic parameter	Page 6-13
SETB	Assign a value to a boolean string symbolic parameter	Page 6-13
SETC	Assign a value to a string symbolic parameter	Page 6-14
String Manipulation Directives		
ASEARCH	Assembler String Search	Page 6-14
AMID	Assembler Mid String	Page 6-15

Defining Parameters Using Assembler Input

AINPUT	Assembler Input	Page 6-15

Branching Directives

AGO	Unconditional Branch	Page 6-20
AIF	Conditional Branch	Page 6-21

Miscellaneous Directives

ACTR	Assembly Counter	Page 6-21
ANOP	Assembler No Operation	Page 6-22

Directive Formats

A directive is coded in the same way as an instruction, with four fields as shown:

```
[label]  op code     [operand]   [comment]
```

The label, operand and comment may be required, absent or optional, depending on the directive. The syntax for each directive is shown under the directive's description. Complete information on coding comments is given in Chapter 4, "Coding Conventions".

Directive Descriptions

Program Control Directives

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

START	Start subroutine
PRIVATE	Define private code segment
DATA	Define data segment
PRIVDATA	Define private data segment
USING	Using data segment
ENTRY	Define entry point
END	End program segment

START Start Subroutine

```
label    START                                      [comment]
```

The START directive indicates the start of a named code segment, including both main programs and subroutines. It is not necessary to perform separate assemblies to assemble each of the code segments, nor is it necessary to put them in separate disk files.

Each START directive requires a label, which becomes the subroutine name in the object module produced by the assembler. The label is global in scope, thus the link editor can inform other subroutines of its location at link edit time. This allows subroutines that are assembled separately to be combined later by the link editor.

The START directive is required. If it is omitted, the Assembler generates an error message. The code segment extends until the END directive.

PRIVATE — Define a private code segment

```
label     PRIVATE                                          [comment]
```

The directive PRIVATE operates in a similar way to the directive START, except that it is not accessible from outside the object module in which it was created. The effect is not noticed unless the segment is placed in a library or separate compilation is used. In that case, the segment is accessible from the segments in its own object module file, but not visible outside. Segments in different files can have the same names as long as all but one is private. This mimics the C static function.

DATA — Define Data Segment

```
label     DATA                                             [comment]
```

The DATA directive indicates the start of a data segment. Its purpose is to set up data tables which several routines can access. The data segment continues until an END directive is specified.

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

Labels used within a data segment become local labels for any subroutine issuing a USING directive for the data segment. Labels within data segments should not be duplicated in other data segments.

PRIVDATA — Define a private data segment

```
label     PRIVDATA                                         [comment]
```

The directive PRIVDATA operates in a similar way to the directive DATA, except that it is not accessible from outside the object module in which it was created. The effect is not noticed unless the segment is placed in a library or separate compilation is used. In that case, the segment is accessible from the segments in its own object module file, but not visible outside. Segments in different files can have the same names as long as all but one is private. This mimics the C static function.

USING Using Data Segment

```
[label]  USING       data_segment_name               [comment]
```

The USING directive is used in a code segment to designate a specific data segment that it must access. The operand field contains the name of the data segment. Labels defined within the subroutine take precedence over labels by the same name in data segments.

ENTRY Define Entry Point

```
label    ENTRY                                       [comment]
```

The ENTRY directive is used to enter a subroutine in a place other than at its start. Using the ENTRY directive allows a global label to be defined for that purpose. The label is required.

END End Program Segment

```
         END                                         [comment]
```

The END directive indicates the end of a code or data segment. It directs the assembler to print the local symbol table 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.

Data Definition Directives

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

DC	Declare Constant
DS	Declare Storage

DC Declare Constant

```
[label]  DC          constant_definition             [comment]
```

where:

```
constant_definition =  [repeat_count]identifier,'value',
[constant_definition]
```

The DC directive is used to define a constant within a program. The operand begins with an optional repeat count, which must be in the range 1 to 255 decimal, followed by an identifier describing the value type. The variable being defined will be placed in the object file as many times as specified by the repeat count. The identifier is followed by the value itself, enclosed in quote marks, separated by a comma. Optionally, a second constant definition follows the first, separated by a comma. For example,

```
LABEL DC 2I'2,3',I1'4'
```

places five integers into memory, four sixteen-bit and one eight-bit. The resulting hexadecimal values would be

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

DC Value types

The DC directive value type identifiers are listed below:

Ix	Integer
R	Reference an address
S	Soft reference
H	Hexadecimal constant
B	Binary constant
F	Floating point
D	Double precision floating point
E	Extended floating point

Ix - Integer

The value type I specifies an integer. An integer length from one to eight bytes is indicated by a digit from 1 to 8 following I. If the integer length is omitted, a two-byte integer is generated. All integers are stored least-significant byte first. Integers from one to four bytes in length can be expressed as expressions, including external references. Longer integers can be expressed only as a signed decimal number.

The table below gives the valid range of signed integers that can be expressed with each length of integer. The Cortland Toolbox contains subroutines to perform integer math operations: refer to Chapter 7, "The Cortland Libraries", for more information.

Size	Smallest	Largest
1	-128	127
2	-32768	32767
3	-8388608	8388607
4	-2147483648	2147483647
5	-549755813888	549755813887
6	-140737488355328	140737488355327
7	-36028797018963968	36028797018963967
8	-9223372036854775808	9223372036854775807

A - Address

The value type A is functionally equivalent to I2. It is intended for use in building tables of addresses.

R - Reference an Address

The value type R generates a reference to an address in the object module without saving the address in the final program. This allows a program to note that a subroutine will be needed from the subroutine library without reserving storage for the subroutine address. Using the value type R with the type S below allows for the development of a pseudo system which loads and links only those parts of the pseudo system language needed by a particular program. This option is then used by the pseudo code instructions to insure that any library subroutines that will be needed to execute that instruction are linked. Note that this directive does not take up space in the finished program.

S - Soft Reference

The value type S generates two bytes of storage for each address in the operand, but does not instruct the link editor to link the subroutines into the final program. If the subroutine is not linked, the binary program produced by the link editor will have `$0000` as the two-byte address. This allows a table of addresses to be built, but only those subroutines requested elsewhere in the program, usually by an R type reference, have their addresses placed in the table. See the discussion of the value type R, above.

H - Hexadecimal Constant

The type H has a hexadecimal value. The string between the single quote marks may contain any sequence of hexadecimal digits and blanks. Embedded blanks are removed, and the hexadecimal value is stored unchanged. If there are an odd number of digits, the last byte is padded on the right with a zero, as shown in the following example:

Code **Value**

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

B - Binary Constant

The value type B designates a binary constant. The string between the quote marks can contain any sequence of zeros, ones, and blanks. The blanks are removed, and the resulting bit values are stored. If a byte is left partially filled, it is padded on the right with zeros, as shown in the folowing example:

Code	Value
`DC     B'01 01 01 10'`	`56`
`DC     B'11111111'`	`FF 80`

C - Character String

The value type C designates a character string. The string enclosed in quote marks may contain any sequence of keyboard characters. If a quote mark is desired, enter it twice to distinguish it from the end of the string:

Code	Value
`DC     C'NOW IS THE TIME...'`	`4E 4F 57 20 49 53 20 54 48 45 20 54 49 4D 45 20 2E 2E 2E`
`DC     C'NOW'IS THE TIME'`	`4E 4F 57 27 53 20 54 48 45 20 54 49 4D 45`

Normally, strings are stored with the high-order bit off, corresponding to the ASCII character set. If characters are to be written directly to the Cortland screen, it is preferable to have the high bit set. In that case, use the MSB directive to change the default.

F - Floating Point

The type F designates a value entered as a signed floating-point number, with an optional signed exponent starting with E. Embedded blanks are allowed anywhere except within a sequence of digits. The number is stored as a four-byte floating-point number. Bit one is the sign bit, and is 1 for negative numbers. The next eight bits are the exponent, plus `$7E`. The exponent is a power of two. The remaining 31 bits are the mantissa, with the leading bit removed since it is always 1 in a normalized number. The mantissa is stored most significant byte to least significant byte. This format is compatible with the IEEE floating-point standard and with the Standard Apple Numeric Environment (SANE).

Numbers can range from approximately `1E-38` to `1E+38`. The mantissa is accurate to over seven decimal digits.

Code	Value

`DC     F'3,-3,.35E1,6.25 E-2'`	`40400000`
	`C0400000`
	`40600000`
	`3D800000`

D - Double Precision Floating Point

The type D designates a double-precision floating-point value. This is identical to F, except that an eight-byte number is generated with an eleven-bit exponent and a forty eight-bit mantissa. Numbers can range from about 1E-308 to 1E+308. The mantissa is accurate to slightly more than 15 decimal digits. The exponent is stored most significant byte first.

The type D designates a value entered as a signed floating-point number, with an optional signed exponent starting with E. Embedded blanks are allowed anywhere except within a sequence of digits. The number is stored as a eight-byte floating-point number. Bit one is the sign bit, and is 1 for negative numbers. The next eleven bits are the exponent, plus `$7E`. The exponent is a power of two. The remaining 48 bits are the mantissa, with the leading bit removed since it is always 1 in a normalized number. The mantissa is stored most significant byte to least significant byte. This format is compatible with the IEEE floating-point standard and with the Standard Apple Numeric Environment (SANE).

Code	Value
`DC     D'3,-3,.35E1,6.25 E-2'`	`4008000000000000`
	`C008000000000000`
	`400C000000000000`
	`3FF0000000000000`

DS Declare Storage

```
        DS          operand                       [comment]
```

The DS directive is used to reserve sections of memory for program use. The operand is coded in the same way as an absolute address for an instruction. The operand is resolved into a four-byte unsigned integer; that amount of memory is reserved.

Symbol Definition Directives

The symbol definition directives let you assign values to individual name symbols. With these directives you can name objects such as numeric constants, individual registers, register lists, and opcodes, so that you can use the names instead of the original objects in your source text. The symbol definition directives include:

EQU	Equate
GEQU	Global equate
RENAME	Rename opcodes

EQU Equate

```
  label     EQU         operand                          [comment]
```

The EQU directive is used to assign the value of the operand rather than the location counter to the label. This allows you to assign a numeric value to a name, with the name to be used instead of the number in further operands.

The operand may contain a label that already has a value. If the label does not have a value, an error is generated since the resulting value may be a direct page address. During the first pass, the Assembler has no way of knowing this, since it could not resolve the equate. Instructions are assumed to be absolute addresses on the first pass, and two bytes are reserved. On the second pass, the equate would be resolved as direct page. The addresses would now occupy only one byte, and further addressing would be incorrect. For the same reason, it is important that equates defining direct page addresses be defined before they are used.

Some examples of equates follow:

```
ONE       EQU    1
TWO       EQU    1+1
FOUR      EQU    TWO*TWO
```

As shown, you can use expressions in the operand, and you can use constants defined by earlier equates. Keep in mind that equates are used to define constants: each term in the expression in the operand field must have a specific value when the EQU is encountered. One of the most common problems that results from this fact is when the following construct is used:

```
HERE      EQU    *
```

to set the label HERE to the address of the current location counter. The value of the current location is not known at assembly time, and the operand is not a constant. If you need to define a label without generating code, you can use the ANOP (Assembler no operation) directive.

```
HERE      ANOP
```

You must define constants before they are used. It is customary to put all equates in a segment following the START directive, although this is strict requirement only when the constant is used later as a direct page or long address.

Has the concept that EQU and GEQU no longer need to be constants or constant expressions been implemented?? This paragraph is probably incorrect

GEQU Global Equate

```
label     GEQU          operand                 [comment]
```

The GEQU directive is functionally equivalent to the EQU directive. Additionally, the label is saved in the global symbol table. All program segments are then able to use the label. Labels defined by the GEQU directive are resolved at assembly time, not at link edit time. They are included in the object module, so library routines can use global equates to make constants available to the main program.

RENAME Rename Op Codes

```
          RENAME        old_opcode,new_opcode   [comment]
```

where

new_opcode is eight characters or less, and contains no spaces or the & character.

With the Cortland Workshop Assembler, it would be possible to develop a cross assembler using macros. One problem though, is that other CPUs may have an opcode that conflicts with a 65816 instruction or an existing Cortland Workshop directive. This can be resolved by renaming the existing op code to prevent a conflict. The operand is the old op code followed by the new one. In the following example, the first time LDA is encountered, it is a 65816 instruction. The second time, it is not found in the op code table, so the assembler tries to expand it as a macro.

```
INST            START
                LDA         #1
                END

                RENAME      LDA,NEW
MACRO           START
                LDA         #1
                END
```

The RENAME directive cannot be used inside a segment, that is, it cannot come between the directives START and END.

Memory Designation Directives

The memory designation directives include:

ALIGN	Align to a boundary
ORG	Designate origin
MEM	Reserve memory

ALIGN Align To a Boundary

```
        ALIGN       operand                     [comment]
```

The ALIGN directive is used either prior to the start of a code or data segment, or within a segment. Used before a START, PRIVATE, DATA or PRIVDATA directive, it directs the link editor to align the segment to a byte boundary divisible by the absolute number in the operand of the ALIGN directive. This number must be a power of 2. For example, to align a segment to a page boundary, use the sequence

```
        ALIGN  256
SEG     START
        END
```

Within a segment, ALIGN inserts enough zeroes to force the next byte to fail at the indicated alignment. This is done at assembly time, so the zeros appear in the program listing. If ALIGN is used in a subroutine, it must also have been used before the segment, and the internal align must be to a smaller boundary than the external align.

ORG Designate Origin

```
            ORG         memory_location              [comment]
```

The ORG directive is used to designate the memory location at which a program will begin when it is assembled into machine language. This location is specified as an absolute address in the operand field. The default location is at $010000. Note that the ORG directive must appear prior to the first START, PRIVATE, DATA or PRIVDATA directives.

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. Again, the operand is an absolute address, and must be a constant. In this case, though, the actual method of performing the ORG is to insert zeros until the desired location is realized. This action is performed by the link editor as the final binary module is built.

The ORG directive can also be used inside a program segment, but in that case the operand must be a *, indicating the current location counter, followed by a + or -, and a constant expression. The location counter is moved forward or backward by the indicated amount. Thus,

```
    ORG   *+2
```

is equivalent to

```
    DS   2
```

while

```
    ORG   *-1
```

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

MEM Reserve Memory

```
            MEM         address,address              [comment]
```

The operand for this directive is two absolute addresses, separated by a comma. The absolute addresses specify a range of memory that is to be reserved as a data area. The Link Editor will ensure that subroutines are not placed in this range of memory. This is done by checking the length of each subroutine to see if it will enter a reserved area. If it does, it is started after the end of the reserved area. This directive is intended for use when the high resolution graphics pages are needed.

File Control Directives

File control directives let you save an assembled object module on disk, and create and access files other than the current source text files during assembly. These directives include:

APPEND	Append a file
COPY	Copy a file
KEEP	Keep object module

All three file control directives specify a disk file name in the operand. Any valid CPW pathname may be coded.

APPEND Append A File

```
        APPEND      pathname                        [comment]
```

The APPEND directive is used to tranfer processing to the beginning of the specified file. Any lines following the APPEND directive in the original file are ignored.

The APPEND directive allows you to write large programs and divide them into sections. Then, use the APPEND directive to instruct the Assembler to bring the appended file into the program. The APPEND directive works like a GOTO, in that any line after the APPEND directive is ignored. The directive can be used to append a file on a disk that is not currently in the Cortland. When the Assembler encounters the APPEND directive, it passes control to the Shell. The Shell checks for the disk, and the disk is not found, the Shell returns a message prompting you to place the disk online. In this way, the APPEND directive allows you to assembly any program on a one-drive system that you could assembly on a two- or more-drive system, although you will need to swap disks. If you have made an error, hit the ESC key.

COPY Copy A File

```
        COPY        filename                        [comment]
```

The COPY directive is used to transfer processing to the beginning of a specified file. After the entire file is processed, assembly continues with the first line after the COPY directive in the original file. A copied file can copy another file; the depth is limited by the available memory, an is generally about three or four levels.

KEEP Keep Object Module

```
        KEEP        name                            [comment]
```

The KEEP directive is used to save the assembled code on disk as a relocatable object module, using the specified name as the root name. The link editor may then be used to generate an executable binary file. This directive may only be used one time, and must appear before any code-generating statements.

Assembler Option Directives

The Assembler option directives control the assembly process. For all except MERR, the operand consists of either ON or OFF, choosing whether or not the option is in effect.

A special attribute, S, is provided to let the assembler check the settings of these directives. These attributes are described in Chapter 4, "Coding Conventions" and in Chapter 6, "Macros". The Assembler option directives include:

IEEE	Generate IEEE format numbers
LONGA	Select accumulator size
LONGI	Select index register size
MSB	Set the most significant bit of characters
MERR	Set the maximum error level
CASE	Specify case-sensitivity
OBJCASE	Specify case-sensitivity in object module
65C502	Enable 65C02 code
65816	Enable 65816 code

IEEE Generate IEEE Format Numbers

```
            IEEE        ON|OFF                          [comment]
```

In its default setting, DC directives with F and D operands generate numbers compatible with the IEEE floating point standard. If IEEE is turned off, F type DC directives will generate Applesoft compatible numbers. D type DC directives are not affected; they always generate IEEE double-precision numbers.

LONGA Select Accumulator Size

```
            LONGA       ON|OFF                          [comment]
```

The 65816 processor can perform both sixteen-bit and eight-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 assembler has no idea how that bit will be set at run time, thus it is the responsibility of the programmer to tell the Assembler using this directive. LONGA ON indicates 16-bit operations, while LONGA OFF indicates eight-bit operations. The default is ON. The only difference this will make in the assembled program is to change the number of bits 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
```

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

LONGI Index Register Size Selection

```
            LONGI       ON|OFF                          [comment]
```

This directive controls the number of bytes reserved for immediate loads to the X and Y registers when using the 65816. The default is ON. See LONGA for a complete discussion.

MSB Set the Most Significant Bit of Characters

```
        MSB        ON|OFF                    [comment]
```

Character constants and characters generated by DC statements have bit seven cleared, corresponding to the ASCII character set. If MSB ON is coded, characters generated have bit seven turned on, and appear normal on the Cortland CRT. The default is OFF.

65C502 Enable 65C02 Code

```
        65C502     ON|OFF                    [comment]
```

The 65C02 is used in older models of the Apple II. The instructions and addressing modes available on that processor can be enabled and disabled with this directive. The default is OFF.

65816 Enable 65816 Code

```
        65816      ON|OFF                    [comment]
```

When off, 65816 instructions and operands are identified as errors by the assembler, allowing 65C02 or 6502 code to be generated without fear of accidentally using a feature not available on the smaller processor. The default is OFF.

MERR Set the maximum error level

```
        MERR       operand                   [comment]
```

MERR sets the maximum error level that can be tolerated and still allow the Assembler to link edit immediately after the assembly as would happen with an ASML or AMSLG command from the Shell. The default value is zero. The operand is coded like an absolute address and is evaluated to a one-byte positive integer.

CASE Specify case-sensitivity

```
        CASE       ON|OFF                    [comment]
```

Using the CASE directive with the option ON makes labels case-sensitive.
CASE OFF makes labels case-insensitive.

OBJCASE Specify case-sensitivity in object module

```
        CASE       ON|OFF                    [comment]
```

Using the directive OBJCASE with the option ON makes labels sent to the object module case-sensitive, whether or not they are treated as case sensitive inside the Assembler. Specifying OBJCASE OFF makes exported labels 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 behaviour is to be different from the local behaviour, you must specify the OBJCASE directive last.

Listing Option Directives

The listing option directives allow you to specify certain options when you are listing the assembly to the screen or to the printer. The listing option directives are summarized below.

ABSADDR	Allow absolute addresses
EJECT	Eject the page
ERR	Print errors
EXPAND	Expand DC statements
INSTIME	Show instruction times
LIST	List output
PRINTER	Send output to printer
SETCOM	Set comment column
SYMBOL	Print symbol tables
TITLE	Print header

ABSADDR Allow Absolute Addresses

```
         ABSADDR      ON|OFF                          [comment]
```

The ABSADDR directive allows operands of ON or OFF. The default is OFF. If ABSADDR ON is specified, a new column of six-byte addresses will be shown to the left of the relative offsets that ORCA/M now places in the output listing. The relative offsets will still appear in the output.

The value shown in this new column is a base number plus the number of bytes generated by the assembler since the last change in the base number. The base number defaults to `$010000`. An ORG directive will change the base number to the value specified by the ORG's operand. The net effect is that this column will show the correct absolute memory location of the line, assuming that the listing is from a full assembly, the default ORG has not been overridden by the linker, and the loader loads the file to the default location of `$010000`.

EJECT Eject The Page

```
         EJECT                                        [comment]
```

When a printer is in use, this directive causes the output to skip to the top of the next page. This can be of help in structuring the output of long subroutines. The directive does not affect the code sent to the output file in any way.

ERR Print Errors

```
ERR          ON|OFF                        [comment]
```

If ERR ON has been specified, errors are always printed, regardless of this flag. If ERR OFF has been specified, this flag allows error lines to still be printed. If turned off, errors are no longer printed, but the number of errors found will still be listed at the end of the assembly. The default is ON.

EXPAND Expand DC Statements

```
EXPAND       ON|OFF                        [comment]
```

If turned on, this option causes all bytes generated by DC directives to be shown in the output listing, up to a maximum of sixty-four bytes. Only four bytes of a DC directive can be displayed on a line, so the option defaults to OFF to save paper and patience. When the option is turned off, only the first four bytes of the generated code are shown with the output.

INSTIME Show Instruction Times

```
INSTIME      ON|OFF                        [comment]
```

The INSTIME directive accepts operands of ON or OFF. The default is OFF. If INSTIME OFF is specified, a new column of cycle times is inserted in the output listing immediately before the text of the source line. This column is three characters wide. It shows the number of machine cycles required to execute the assembly language instruction appearing on that line. The characters are blanks for macros, directives and comments. The first character indicates the number of cycles, while the other two characters can be any of the following:

*	Add one cycle if a page boundary is crossed
'	Add one cycle if the branch is taken and one more if a page boundary is crossed
+	Moves are 4 + 7 (number of bytes moved) cycles long

LIST List Output

```
LIST         ON|OFF                        [comment]
```

A listing of the assembler output is sent to the current ouptut device. If the listing is turned off, the assembly process speeds up by about 10%. The default is ON.

PRINTER Send Output to Printer

```
PRINTER      ON|OFF                        [comment]
```

If PRINTER ON is specified, output is sent to the printer. A printer capable of printing at least eighty columns is expected there. If a printer is not connected, the system will hang. The slot number and printer characteristics may be changed by re-configuring the operating system. If the option is turned off, output is sent to the video display. The default is OFF.

SETCOM Set Comment Column

```
        SETCOM      comment_column_#                [comment]
```

The SETCOM directive is used to set the start of the comment column. The assembler will not search beyond this column for an op code, and will not search for an operand unless there is exactly one space between the op code and operand. In this way, a comment is not accidentally used as part of an operand. This column defaults to forty, but can be changed to any number from one to eighty by specifying the number in the operand field.

The SETCOM directive is coded exactly like an absolute address.

SYMBOL Print Symbol Tables

```
        SYMBOL      ON|OFF                          [comment]
```

An alphabetized listing of all local symbols is printed following each END directive. After all processing is complete, global symbols are printed. If this option is turned off, assemblies speed up slightly. The option can also be used to save paper. The default. is ON.

TITLE Print Header

```
        TITLE       [string]                        [comment]
```

The TITLE directive is used to place page numbers at the top of each page sent to the printer. If an operand is coded, the string used is printed at the top of each page, immediately after the page number.

The operand *string* is optional. If it is coded, it must be a legal string, and must be enclosed in single quote marks if it contains blanks or starts with a single quote mark. If the string is longer than sixty characters, it is truncated to sixty characters.

This page is left intentionally blank

Chapter 6

Macros

This chapter consists of two sections: "Using Macros" and "Writing Macros". "Using Macros" takes you through a sample session with macros. It tells you how to build a macro library and describes several assembler directives you will use when working with macro libraries and listing the assembly. "Writing Macros" gives you information on how to include macros in your source text, including macro formats, addressing modes, and data types.

Each of the macro directives are described in detail in the sections appropriate to their use. These macros are summarized in Chapter 5, "Directives", with a cross reference to the page in this chapter on which they are described.

Using Macros

A macro is a pre-defined sequence of instructions and directives that acts as a template, allowing you to code a single macro call in your source program which is then expanded by the Assembler at assembly time. When the Assembler expands a macro call, it replaces the macro call statement with the contents of the macro definition, substituting actual values for certain of its variables and parameters. In this way, you can create lengthy source text sequences, by using the macro libraries available with the Cortland or by creating your own macro definitions.

The following sample session introduces you to using macros. It describes how to how to build a macro library and make it available to the program with the directive MCOPY.

Writing the source program

For the purpose of this sample session, a program that includes a number of macros is illustrated below. To write your own program, refer to the section "Macro Coding Conventions".

sample program to be supplied

Building a macro library

When you assemble a program that uses macros, the Assembler accesses a macro library file that you have created for that purpose. When the Assembler accesses the file, it searches for the macro definitions that you have called, then places the expanded code in your source program. The directive MCOPY in the sample program above instructs the

Assembler to search the macro library file *test.macros*. You can instruct the Assembler to use the Cortland macro libraries directly, but it is not as time-effective as using a custom-built library file. You can build a custom macro file using the Shell command MACGEN.

MACGEN scans your program, opens a file called SYSMAC on the work prefix, and writes the macros to this temporary file. When all the macros that you have referenced have been resolved, MACGEN copies this temporary file to the output file you have specified and deletes the working file. At the Shell prompt, type:

```
# MACGEN   myprog    test.macros    /CPW/ROM.MACROS/TOOLSETX.MACROS
```

where *myprog* is the name of the sample program you just entered, *test.macros* is the name of the macro file you are creating, and /CPW/ROM.MACROS/TOOLSETX.MACROS is the full pathname of the macro library file you wish MACGEN to search. For this example, the library file is one of the Cortland Tools, TOOLSETX.

You can also enter the command MACGEN without specifying any files, in which case the Shell will prompt you for the appropriate filenames.

A complete description of the CPW directories that contain the Cortland macro libraries are described in the section "System Macro Files" in Chapter 7, "The Cortland Libraries".

There are three macro library directives that you can use while running macros. In addition to MCOPY, which you have already used, there are the directives MLOAD and MDROP. MLOAD loads a specified library file into the macro library buffer. MDROP lets you drop a macro library you no longer need. These directives are described below.

MCOPY Copy Macro Library

```
MCOPY        macro_library_file
```

The name of the file is placed in a list of available macro libraries. If an operation code cannot be identified, all macro files in the list are loaded into the macro buffer in sequence, and checked for a macro with the specified name. 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 A Macro Library

```
MDROP        macro_library_file
```

Removes the specified library from the list of macro libraries. This might be necessary if more than four libraries are being used. It can also speed up processing if a library is no longer needed.

If the macro library is active at the time the MDROP directive is encountered, it is left there and searched for macros until a search is made which loads a different library, or until a MLOAD directive is used.

MLOAD Load A Macro Library

```
        MLOAD       macro_library_file
```

The list of macro libraries is checked. If the specified file is not in the list, it is placed there. The file is then loaded into the macro library buffer.

This directive can be used to speed up assemblies by helping the macro processor to find macros.

Executing the Program

Assemble, link and run the program as usual. You may notice that it takes a longer time to assemble the program since the Assembler has to expand the macros. The lines generated message tells you how many lines of code the Assembler created.

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 top of your program. The TRACE directive instructs the Assembler to print all the lines that the Assembler processes. This can be useful when you are debugging macros.

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.

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. Especially when debugging macros, it is desirable to see all of the lines the assembler processes. To do this, use TRACE ON. The default is OFF.

Writing Macro Definitions

This section tells you how to write a macro definition, and discusses the macro language directives MACRO, MEND, MNOTE and MEXIT.

The instruction in the source file that instructs the Assembler to expand a macro is called the macro call statement, or **macro call.** A new macro is created by coding a **macro definition,** which tells the Assembler which instructions to replace the macro call with.

A macro definition consists of four parts:

- The MACRO directive
- The macro definition statement
- Model statements
- The MEND directive

A macro definition starts with the directive MACRO. Immediately following the MACRO directive is the macro definition statement. In the macro definition statement, the name of the macro being defined is placed in the operation code field. If the operation code that the Assembler is trying to identify matches it, the Assembler uses the definition that follows to replace the macro call in the source file within the instructions found in the body of the macro itself. The macro definition opcode may be any sequence of keyboard characters except blanks or the & character. It may contain from one to ten characters.

The following sample code illustrates a macro definition. This macro returns the version number of Cortland Integer Math Tools you are using.

```
 MACRO
&lab  _IntVer
&lab  ldx #$040B
 jsl  $E10000
 MEND
```

The statements between the macro definition statement and the MEND directive are called model statements since the macro processor uses them as models for the new instructions.

Optionally, the macro may contain the directives MNOTE and MEXIT. MNOTE is used to code an error message into the macro. MEXIT is used as a return from a macro definition in conditional assembly branches.

With the exception of MNOTE, the macro language directives are valid only in a macro file; if used inside a regular source file, they will create an error. These directives are described below.

MACRO Begin A Macro Definition

Each macro definition begins with a MACRO directive. This directive is coded like an operation code. No operand of label is needed, and any present is ignored. Its sole purpose is to set the macro definition apart from others in the file.

MEND End A Macro Definition

Each macro definition ends with a MEND directive. This directive is coded like an operation code. No operand of label is needed, and any present is ignored. Its sole purpose is to set the macro definition apart from others in the file.

MNOTE Macro Note

A macro definition may include a MNOTE directive. The operand of an MNOTE directive contains a message, optionally followed by a comma and a number. The assembler prints the message on the output device as a separate line. If the number is present, it is used as a severity code for an error.

Assume that the following statements appear in a program:
Example:

Code

```
*  MNOTE  FOLLOWS
         MNOTE 'ERROR!',4
```

Output

```
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 correct (user defined) syntax. Although MNOTE statements are intended for use inside macros, they are legal inside a source program.

MEXIT Exit Macro

An MEXIT directive indicates that a macro expansion is complete. Unlike MEND, it does not indicate the end of a macro definition. A good way to conceptualize this directive is to think of it as a return from a macro definition. The MEND is the end of the definition, but the MEXIT can return from within the macro definition. The MEXIT directive is designed for use with conditional assembly branches.

Macro Coding Conventions

This section discusses macro formats, macro addressing modes and macro data types.

Macro Formats

The format of a macro is similar to the format of an instruction or directive: it consists of a label field, the macro name, the operand field and an optional comment field.

Macro Addressing Modes

Macros use a variety of addressing modes to increase the power and flexibility of each macro. There are four addressing modes supported by the macros: immediate, absolute, indirect and stack.

Immediate Addressing

Immediate addressing is available on most macros that require an input to perform their function. An immediate operand is coded as a pound sign (#) followed by the value for the operand. All data types are supported.

For example,

```
        PUT8   #4500000000
```

writes the approximate population of the Earth to the screen.

Absolute Addressing

Absolute addresses are coded as a number, label, or expression, using the same rules as absolute addresses on instructions. An absolute address designates the memory location to use as a source or destination by the macro.

For example:

```
        TBS
```

Indirect Addressing

Indirect addresses take the form of an address which points to the address of the data rather than the data itself. Indirect addressing is indicated by enclosing the absolute address where the effective address is stored in braces.

For example:

```
        MUL    {P1},{P2}
```

multiplies the number pointed to by P1 by the one pointed to by P2, placing the result where P1 points.

Stack Addressing

Stack addressing refers to taking a source value from the evaluation stack, or storing a result there. The evaluation stack is the stack used by Cortland languages to pass parameters and evaluate expressions. It is a software stack, distinct from the hardware stack in page 1 for the 65816.

When discussing stack operations, it is customary to refer to values based on the top of the stack (TOS). Thus, the value on the top of the stack is said to be at TOS, while the number below the top one is at TOS minus one.

The ***?** macro can be used to set up this stack.

need example

Macro Data Types

The macros support several data types, including three lengths of integers, characters, strings, and boolean variables. Typing is not enforced; it is possible to read a four-byte integer into an area, then access it as a two-byte integer. The type of data in use is indicated by a single character, as illustrated in the table below.

Type	**Character**
Two-byte integer	2
Four-byte integer	4
Eight-byte integer	8
Character	C
String	S
Boolean variables	B

The type character is used as a part of the macro name. For example, the ***?*** macro can be used to write any of these variable types to an output device. The type is indicated by replacing the x with one of the characters.

need example

Two-byte Integers

A two-byte integer requires two bytes of storage. It is represented in two's complement notation, with the least significant byte stored first, followed by the most significant . Two-byte integers range from minus 32768 to 32767.

Four-byte Integers

A four-byte integer requires four bytes of storage. It is represented in two's complement notation, with the least significant byte stored first, proceeding sequentially to the most significant byte, which is stored last. Four-byte integers range from minus 2147483548 to 2147483647 .

Eight-byte Integers

An eight-byte integer requires eight bytes of storage. It is represented in two's complement notation, with the least significant byte stored first, proceeding sequentially to the most significant byte, which is stored last. Eight-byte integers range from minus 9223372036854775808 to 9223372036854775807.

Characters

A character requires one byte for storage. The ASCII character set is used to represent characters; in general, it is not important whether the high bit is on or off. The system provides all inputs with the high bit off, and converts any outputs as needed; the only conflict arises with comparisons. For that reason, it is recommended that character data always be represented with the high bit off.

Depending on the output device, the effect of a control character may vary. If an output device does not respond to a given control character because that character is not defined, the control character is ignored. Refer to the appropriate reference manual for the hardware device that you are using for more information.

Strings

A string is a sequence of characters of variable length. Each string consists of three parts. The first byte contains the maximum number of characters in the string. This number ranges from 1 to 254. The second byte contains the number of characters currently in the string. This number ranges from zero to the value of the first byte. The third field contains the characters in the string itself. One byte is reserved for each possible character in the string. Unused bytes are not defined and have unreliable values. A string requires two bytes more than the maximum number of characters in the string for storage.

Boolean Variables

A boolean variable requires one byte of storage. It has a value of either TRUE (non-zero) or FALSE (zero).

Using Macros in Conditional Assemblies

Using the Cortland macro directives, you can write structures that determine whether the Assembler will process or ignore sections of source text. These structures can also assign values to variables in your macro definitions. This facility is called **conditional assembly.** It is a powerful tool for creating and controlling variants of your source text.

This section discusses both the conditional assembly directives that define symbolic parameters and assign new values to them and the Assembler control directives which modify the order in which statements are processed by the Assembler.

Defining Symbolic Parameters

A symbolic parameter is a special variable used by the Assembler. Symbolic parameters are true variables in that they can be assigned a value that can later be changed. There are three types of symbolic parameters: arithmetic (A), boolean (B) and character (C).

A symbolic parameter is coded as an & character followed by the symbolic parameter name. The name itself has the same syntax conventions as a label.

When the Assembler encounters a symbolic parameter, it replaces it with a value before assembling the line. Symbolic parameters whose values are set by passing their values during a macro call are said to be implicity defined by appearing on the macro definition line. Symbolic parameters whose values are set during the macro expansion using the conditional assembly directives LCLx, GLBx and SETx are said to be explicity defined.

Defining Parameters Implicitly

Implicitly defined symbolic parameters may be of the character type only. This type of parameter may be positional or keyword. A positional parameter gets its value by being matched with a character string in the source file by position. A keyword parameter gets defined by assigning a value to the symbolic parameter by means of an equals sign.

Positional Parameters

The following code is an example of assigning a value to a positional parameter.

```
          MACRO
&LAB      COUT          &CHAR
&LAB      LDA           &CHAR
          JSR           $FDED
          MEND
```

This macro is called from a source program as follows:

```
          .
          .
          .
          BEQ           L1
          COUT          #'A'
          JMP           L2
L1        COUT          #'B"
L2        RTS
          .
          .
          .
```

At assembly time, the following code is generated. Note again that the Assembler includes the macro call statement only to show what generated the new line: there is no generated code associated with the macro call line itself.

```
          .
          .
          .
          BEQ       L1
          COUT      #'A'
+         LDA       #'A'
+         JSR       $FDED
          JMP       L2
L1        COUT      #'B'
+L1       LDA       #'B'
+         JSR       $FDED
L2        RTS
          .
          .
          .
```

&CHAR is referred to as a positional parameter since it gets its value by being matched with a character string in the source file by position. Note that the symbolic parameter defined in the label field of the macro definition (&LAB) resulted in the label field of the first line of the macro expansion receiving the value of L1 after the second macro call. The symbolic parameter &LAB was also coded in the first line of the macro body, where the value of the macro call label field was substituted for it during the macro expansion.

The following example, which is a macro to print two characters, illustrates again how positional parameters are set during the macro call:

```
          MACRO
&LAB      COUT2     &C1,&C2
          LDA       &C1
          JSR       $FDED
          LDA       &C2
          JSR       $FDED
          MEND
```

Note that the two symbolic parameter declarations on the macro definition line are separated by a comma, with no intervening spaces. The comma delimits the different positional parameters; spaces are not allowed. When the macro is called, as shown below, the actual parameters are coded identically, that is, with commas separating the fields, and no intervening blanks.

```
          .
          .
          .
          COUT2     #'A',#'B'
+         LDA       #'A'
+         JSR       $FDED
+         LDA       #'B'
+         JSR       $FDED
          .
          .
          .
```

The macro processor determines which actual parameters to substitute for the symbolic parameters by matching their relative positions in the macro call statements with those in the macro definition.

Optionally, a positional parameter need not be coded. However, all commas must be included, as if something had been coded. The macro keeps count of position using the commas, so that later positional parameters appear in the right place.

Keyword Parameters

A keyword parameter is coded by typing the name of the symbolic parameter followed by an equal sign (=) and the value assigned to the parameter. For example, a call to the COUT2 macro could be coded as:

```
          .
          .
          .
          COUT2 C2=#'B',C1=#'A'
+         LDA   #'A'
+         JSR   $FDED
+LDA      #'B'
JSR       $FDED
          .
          .
          .
```

When keyword parameter substitution only is used, the order is not important. The rules for commas and blanks apply in the same way as for positional parameters. Keyword and positional parameters can be mixed. If they are mixed, keyword parameters take up a space and are counted for determining positions. The macro processor counts the number of commas encountered when setting values for positional parameters.

Defining Parameter Values Explicitly

All symbolic parameter types, arithmetic, boolean and character, may be declared explicitly. That is, their values may be set and reset during the macro expansion with the conditional assembly directives, resulting in an extremely powerful conditional assembly capability. All the conditional assembly directives described in this section are also valid in source files. They are included here since their main use is in macro definitions. The directives used to define symbolic parameters explicitly include the directives LCLx, GBLx, SETx, and the string search directives AMID, ASEARCH, and AINPUT.

Parameter Scope

Symbolic parameters may be defined either for the current macro expansion or for the entire subroutine. Defining symbolic parameters whose scope is the entire subroutine allows macros to communicate with each other. Symbolic parameters which are only valid inside a macro are called local symbolic parameters; those valid throughout the subroutine are called global symbolic parameters. Using global parameters lets you create macros that pass information to one another.

Symbolic Parameter Definition Statements

A symbolic parameter definition statement does not contain a label. The operand field consists of the name of the symbolic parameter to be defined. If the symbolic parameter is to be subscripted, the maximum allowable subscript 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.

Defining Parameters With LCLx Directives

The LCLx directives are:

LCLA	define a local arithmetic symbolic parameter
LCLB	define a local boolean symbolic parameter
LCLC	define a local character symbolic parameter

For example, the statements

```
LCLA    &NUM
LCLB    &LOGIC
```

define an arithmetic symbolic parameter &NUM and a boolean symbolic parameter &LOGIC. Both are initialized to zero. Arithmetic symbolic parameters can contain any four-byte signed integer value, while boolean symbolic parameters can take on any value from zero to 255. When used in a logical expression, zero is treated as false and any other value as true.

When the Assembler encounters the line

```
LCLC  &STRING
```

the symbolic parameter &STRING is defined. Strings have a length as well as a value: here, &STRING has an initial value of the null string, or a string with no characters, and a length of zero. A string variable can hold up to 255 characters.

Defining Parameters With GBLx directives

The directives GBLx define symbolic parameters with global scope. These directives are:

GBLA	define a global arithmetic symbolic parameter
GBLB	define a global boolean symbolic parameter
GBLC	define global character symbolic parameter

For example, the statements

```
GBLA    &NUM
GBLB    &LOGIC
GBLC    &STRINGS
```

define arithmetic, boolean and character symbolic parameters of global scope.

One predefined, permanent global arithmetic-type symbolic parameter exists called &SYSCNT. The value of &SYSCNT is set to one at the beginning of each subroutine and is incremented at the beginning of each macro expansion. It is used to prevent labels defined inside macros from being duplicated if the same macro is used more than once in the same subroutine. This is done by concatenating &SYSCNT to any labels used within the macro definition itself.

Changing Parameter Values with Set Symbol Directives

You can change and modify the values of symbolic parameters using set symbols. The set symbol is a directive that is logically equivalent to the assignment operator in most languages, but unlike most languages the operator itself is typed. This means that the directive used to assign a value to an arithmetic symbolic parameter is different from the one used to assign a value to a boolean symbolic parameter. The set symbol directives are:

SETA	set an arithmetic symbolic parameter
SETB	set a boolean symbolic parameter
SETC	set a character symbolic parameter

SETA Set Arithmetic

The SETA directive uses a constant expression in the operand field. The operand is resolved as a four-byte signed hexadecimal number. The result is assigned directly to the arithmetic symbolic parameter.

Examples:

```
&NUM      SETA  4
&N(&NUM)  SETA  &NUM2+LABEL*4
```

SETB Set Boolean

The SETB directive uses a boolean expression in the operand field. The expression is evaluated as true or false. If true, the symbolic parameter is assigned a value of one. If false, or if the line contains an error, the symbolic parameter is assigned a value of zero.

The boolean expression in the operand field for a SETB directive is coded using the same rules as an absolute address. It is referred to as a boolean phrase because it most generally takes on a value of true or false (one or zero).

Boolean operators may be used in expressions. If they are used, the resulting expression has a boolean value that appears as a zero or one used to indicate false and true boolean results. Arithmetic results are also valid in a boolean expression, thus a boolean variable can be used in the same way as arithmetic variables. Since only one byte is reserved for each boolean value, the boolean variable selects the least significant byte of an arithmetic result, using it as an unsigned arithmetic value in the range 0 to 255. Use of such a result in a boolean statement will result in the value being evaluated as true if the value is non-zero, and false if the value is zero.

Examples:

```
&FLAG          SETB          A<&NUM
&LOGIC         SETB          &NUM>0
&LOGIC         SETB          1
```

SETC Set Character

The operand is evaluated as a character string and assigned to the symbolic parameter. Several sub-strings may be concatenated to make up the final string; they are separated in the operand field by plus characters (+). Embedded blanks are not allowed. Literal strings containing blanks, commas, or plus signs must be enclosed in quote marks. Quote marks inside quote marks must be doubled.

Examples:

```
&STRING(4)      SETC        &STRING
&STR            SETC        &FKENAME+'.OBJ'
&STRING         SETC        'Here''s a quoted string'
```

Working With Strings

Two directives, ASEARCH and AMID, assist you when manipulating strings. ASEARCH lets you search one string for occurrences of another. AMID allows selection of a small number of characters from a string.

ASEARCH Assembler String Search

This is a special form of arithmetic set symbol. It implements a string search function for character type symbolic parameters.

The ASEARCH directive has three arguments. The first is of character type , and is the target string to be searched. The second is also of character type, and is the string to search for . The last is of arithmetic type, and is the position in the target string to begin the search. The search can be conducted for any sequence of keyboard characters. The result is a number, so the label field must contain an arithmetic symbolic parameter. The arithmetic symbolic parameter is set to the character position in the target string where the search string was first found. If the search string was not found, it receives the value zero.

Examples:

Instructions **Resulting Value**

```
&NUM        ASRCH 'TARGET;, GE,1        4
&NUM        ASRCH 'TARGET', GET,5       0
&NUM        ASRCH 'TARGET', x, 1        0
```

AMID Assembler Mid String

This is a special kind of character type set symbol which provides a mid-string function. It has three arguments in the operand field, separated by commas. Embedded blanks are not allowed.

The first argument is the string to be operated on. It must be a simple string; no concatenation is allowed. If the string contains embedded blanks or commas, it must be enclosed in quote marks. Quote marks inside quote marks must be doubled.

The second and third arguments are of arithmetic type . The second argument specifies the position within the target string of the first character to be chosen. It must be greater than zero. Characters from the target string are numbered sequentially, starting with one. The third argument specifies the number of characters to be chosen.

If the combination of the last two arguments result in an attempt to select characters after the last character of the target string, the selection is terminated. Characters already selected are still valid.

The resulting string is assigned to the character type symbolic parameter specified in the label field.

Examples:

Instructions			Resulting string
`&CHAR`	`AMID`	`'TARGET',2,3`	`RGE`
`&CHAR`	`AMID`	`'TARGET',5,3`	`ET`
`&CHAR`	`AMID`	`'TARGET',7,3`	`null string`

Defining Parameters Using Assembler Input

You can set the value of a symbolic parameter by assigning it a value from the keyboard during the assembly using the directive AINPUT. This directive is described below.

AINPUT Assembler Input

The operand is optional and, if coded, consists of a literal string. If the operand is coded, the string contained in the operand is printed on the screen during pass one as an input prompt. The Assembler then waits for a line to be entered from the keyboard. The string entered is assigned to the character type symbolic parameter specified in the label field.

During pass one, keyboard responses are saved by the Assembler. When an AINPUT directive is encountered on pass two, the reponse given in pass one is again placed in the symbolic parameter specified in the label field. Thus, keyboard response is only needed one time for each input, but the symbolic parameter is set to the response on both pass one and pass two. This means that it is safe to use the response for conditional branching.

Setting Arrays with Symbolic Parameters

You can use all types of symbolic parameter to specify an array subscript. The examples below illustrated how arrays are set and used.

```
&ARR(4)         SETA  16
&ARR(&NUM)      SETA  &ARR(4)
```

Character type 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'
&SUB2           null string
&SUB3           'GAMMA'
```

To use subscripted actual parameters effectively, code the macro itself in such a way as to detect the number of subscripts allowed and to take appropriate action via conditional assembly directives.

Explicity defined symbolic parameters may also be subscripted. The subscript must follow the symbolic parameter name. Only a single subscript is allowed, which must be in the range from 1 to 255. A symbolic parameter used as a subscript for another symbolic parameter cannot be subscripted.

In the following example, assume that four symbolic parameters have been defined, as listed below. The maximum allowable subscripts for the subscripted symbolic parameters 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 below shows instructions as entered in a macro file on the left. 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
          LDA               LB&CHAR2(2)
L&BIN(1)  STA               EQ&BIN(2)
          LD&CHAR2(3)       #1
```

Expanded instructions:

```
LABEL     LDA               STRING1
          STA               LABEL0
```

```
            LDA             #254
            BEQ             L1
            LDA             LB
   L1       STA             EQ0
            LDA             #1
```

Note that a boolean symbolic parameter becomes zero if false and one if true. The null string is valid; it is replaced by nothing.

Concatenating Symbolic Parameters

You can use the period (.), known as the dot operator, to concatenate symbolic parameters as shown below:

```
   STA       &CHAR.&CHAR
```

The period itself does not appear in the final line. It can be used after any symbolic parameter, regardless of how that parameter was defined. It must be used if a symbolic parameter is followed by a character, or if a subscript is followed by a mathematical symbol or expression.

Note that when you use the dot operator in a logical expression, you need to code the dot in addition to the period in the expression. That is to say, the expression

```
      &LOGIC.AND.&LOGIC2
```

returns a syntax error. The correct way to represent the expression is

```
      &LOGIC..AND.&LOGIC2
```

Attributes

Attributes provide a way of giving you information about a symbolic parameter or label in addition to their value. This information is provided by means of attributes which may be thought of as functions that return information about the label or symbolic parameter.

The form of an attribute is

X: Label or Parameter

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
S	Settings attribute

You can use attributes in operands in the same way as a constant.

Count attribute

The count attribute is used to tell whether or not a label 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 label or symbolic parameter is zero. The count attribute of a defined label, or a defined symbolic parameter that is not subscripted, is one. 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
```

While 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 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
         GBLA       &N
.PAST
&N       SETA       &N+1
&LAB     DC         I'&N'
         MEND
```

The second use is to check to make sure that a parameter was passed. The following example illustrates this use.

```
            MACRO
&LAB        ADD           &NUM1,&NUM2,&NUM3
            AIF           C:&NUM3,.PAST
            LCLC          &NUM3
&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 a label is the number of bytes created by the line where the label was defined. This makes counting characters very easy.

need example

The length attribute of an arithmetic symbolic parameter is four. The length attribute of a boolean symbolic parameter is one. For a 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.

Type attribute

The type attribute is used to determine the kind of statement that generated the label. For a symbolic parameter, the type attribute is used to distinguish betweenA, B and C type symbolic parameters. The character that is returned for each type is indicated in the table below.

Character	**Meaning**
A	Address type DC statement
B	Boolean type DC statement
C	Character type DC statement
D	Doube precision floating point type DC statement
F	Floating point type DC statement
G	EQU or GEQU directive
H	Hexadecimal type DC statement
I	Integer type DC statement
K	Reference address type DC statement
L	Soft reference type DC statement
M	Instruction
N	Assembler directive
O	ORG statement
P	ALIGN statement
S	DS statement
X	Arithmetic symbolic parameter
Y	Boolean symbolic parameter
Z	Character symbolic parameter

If a DC statement contains more than one type of variable, the last type in the line determines the type attribute.

Settings attribute

The settings attribute is used to determine the current setting of the assembler flags. These flags are set using directives whose operand is ON or OFF. IF the current seting is ON, the result is one; if the setting is OFF, the result is zero. For example, if you want to write a macro that expands to two different code sequences depending on whether the accumulator is set to 8 or 16 bits, you could use S:LONGA to test the current setting of the

LONGA directive. The directives that accept operands of ON or OFF are summarized below:

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

Modifying The Assembly

The power of the macro language depends primarily on its ability to loop and branch, thereby letting you program the ways in which it expands your original source text. This section discusses the following directives which perform assembly control tasks:

AGO	Unconditional Branch
AIF	Conditional Branch
ACTR	Assembly Counter
ANOP	Assembler No Operation

The branching directives AGO and AIF alter the flow of source statements to be processed by the Assembler. This allows the same macro or source code to be assembled differently based on a given condition. In conditional assembly branches, the destination of the branch may be a **sequence symbol**. A sequence symbol is a line with a period in column one, followed by a label. Comments may follow the label after at least one space. Instructions contained in the line are treated as comments. The line is not printed in the output listing.

AGO Unconditional Branch

The operand contains 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 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.

The search process in a macro definition is similar, except that the search will not cross a MEND or MACRO directive.

Searches for sequence symbols will not cross into a copied or appended file; they are limited to the file in memory.

The AGO directive is not printed in the output listing unless it contains an error.

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.

Example:

```
            AGO         .THERE
   !   THESE LINES ARE IGNORED.
                        .
                        .
                        .
   .THERE
```

AIF Conditional Branch

The operand contains a boolean expression followed by a comma and a sequence symbol. The boolean phrase is evaluated. If true, processing continues with the first statement following the sequence symbol; if false, processing continues with the first statement follwing the AIF directive. 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.

As an example, consider a file which contains the following statements:

```
         LCLA      &LOOP
&LOOP    SETA      4
.TOP     ASL       A
&LOOP    SETA      &LOOP-1
         AIF       &LOOP>0,.TOP
```

The output listing will contain these lines:

```
         ASL       A
         ASL       A
         ASL       A
         ASL       A
```

ACTR Assembly Counter

Each time a branch is made in a macro definition, a counter is decremented. If it reaches zero, processing of the macro stops, to protect it from infinite loops.

The ACTR directive is coded with a number from 1 to 255 in the operand field. The counter is then assigned this value. The ACTR directive is used to limit the number of loops caused by conditional assembly branches. In loops with more than 255 iterations, it must be reset within the bodu of the loop to prevent the counter from reaching zero.

The counter value is set to 255 automatically at the beginning of each macro.

The ACTR directive is not printed unless it contains an error.

ANOP Assembler No Operation

The ANOP directive does nothing. It is used to define labels without an instruction. The label assumes the current value of the program counter.

Chapter 7
The Cortland Libraries

The Cortland includes a comprehensive set of macros to make calls to the Cortland Toolbox, ProDOS 16, and the Shell. Additionally, a group of utility macros are provided as useful aids in writing Assembly code for the Cortland computer. This chapter contains sections on each of these sets of macros. To help you generate you own custom macro library files, a section at the end of the chapter tells you where on the Cortland sytem disk the macros are located.

The Cortland Toolbox

The Cortland Toolbox provides a simple means of constructing application programs that conform to the standard Cortland user interface. By offering a common set of routines that every application program calls to implement the user interface, the Toolbox not only ensures familiarity and consistency for the user but also helps reduce the application's code size and development time. At the same time, it allows a great deal of flexibility: an application can use its own code instead of a Toolbox call wherever appropriate, and can define its own types of windows, menus, controls and desk accessories.

The Cortland Toolbox includes the following groups, organized according to the function they perform:

Desk Manager
Event Manager
Integer Math
Memory Manager
Menu Manager
Miscellaneous Tools
QuickDraw
SANE
Sound Manager
Text Tools
Tool Locator
Scheduler

Note: Some macros on the current CPW disk, V10A3, April 29 1986, have names that differ from the Tools as documented in the *Cortland Tools.* Steve Glass indicated that eventually there would be little discrepancy. For now, I've given the macros the Tool names as documented in the *Cortland Tools.*

Desk Manager

The Desk Manager enables your application to support **desk accessories,** which are mini-applications that can be run at the same time as a Cortland application. There are a number of standard desk accessories, such as the Calculator and the Alarm Clock. You can also write your own desk accessories if you wish.

The Desk Accessories macros include:

Information not yet available

Event Manager

The Event Manager is the part of the Toolbox that allows your application to monitor the user's actions, such as those involving the mouse, keyboard and keypad. The Event Manager macros include:

Event Manager Standard Housekeeping Routines

_EMBootInit	Initializes the Event Managerat boot time
_EMStartUp	Initializes the Event Manager when an application starts up
_EMShut Down	Shuts down the Event Manager and releases any workspace allocated to it
_EMVersion	Returns the version of the Event Manager
_EMReset	Returns an error if the Event Manager is active, but otherwise does nothing
_EMActive	Returns status indicating whether the Event Manager is active
_DOWindows	Returns address of the Event Manager's zero page work area to the Window Manager

Toolbox Event Manager Routines

These routines check events to see if they are of interest to the application. If the events are of interest, and the Desk Manager doesn't want them, the routines return with the event.

_GetNextEvent	Returns the next available event of a specified type or types, and if the event is in the event queue, removes it from the queue.
_EventAvail	Returns the next available event of a specified type or types, but if the event is in the event queue, leaves it in the queue.

Mouse Reading Routines

These routines provide the ability to read the status of the mouse.

_GetMouse	Returns the current location of the mouse.
_Button	Checks the status of a specified mouse button.
_StillDown	Checks a specified mouse button to see if it is still down.
_WaitMouseUp	Checks a specified mouse button to see if it is still down, and, if not, removes preceding mouse-up event.

Posting and Removing Events

_PostEvent	Places an event in the event queue.
_FlushEvents	Removes all events of the type or types specified up to but not including the first event of any type specified by a mask.

Accessing Events Routines

These routines check events to see if they are of interest to the application. If the events are of interest, the routines return with the event.

_GetOSEvent	Returns the next available event of a specified type or types and, if the event is in the event queue, removes it from the queue.
_OSEventAvail	Returns the next available event of a specified type or types, but if the event is in the event queue, leaves it in the queue.

Miscellaneous Event Manager Routines

_TickCount	Returns a count of the number of ticks since the system last started up.
_GetDblTime	Returns suggested maximum difference of ticks which determines a double mouse-click.
_GetCaretTime	Returns the number of ticks between blinks of the caret marking the insertion point.
_SetSwitch	Called by the Control Manager to inform the Event Manager of a pending switch event. Should not be called by an application.
_SetEventMask	Specifies the system event mask. Should not be called by an application.

Integer Math

The Integer Math routines support multiplication and division of several types of numbers, and also convert numbers from one type to another. The types of numbers dealt with are as follows:

- Integers, which are single word signed integers
- Long integers, which are two-word signed integers

- Fixed, which are two-word signed values with 16 bits of fraction
- Frac, which are two-word signed values with 30 bits of fraction

The Integer Math routines are summarized below:

Integer Math Housekeeping Routines

_IMVersion	Returns the version of the Integer Math Tools.
_IMReset	Clears the Heartbeat queue link pointer and sets Mouse flag to "NOT FOUND".
_IMStatus	Returns status indicating whether the Integer Math Tools are active.

Math Routines

The Math routines support multiplication and division of integer, long integer, fixed and frac numbers.

_Multiply	Multiplies two 16-bit inputs and produces a 32-bit result.
_SDivide	Divides two 16-bit signed inputs and produces a signed 16-bit quotient and a signed 16-bit remainder.
_UDivide	Divides two 16-bit unsigned inputs and produces an unsigned 16-bit quotient and an unsigned 16-bit remainder.
_LongMul	Multiplies two 32-bit inputs and produces a 64-bit result.
_LongDivide	Divides two 32-bit unsigned inputs and produces an unsigned 32-bit quotient and an unsigned 32-bit remainder.
_FixRatio	Finds Fixed 32-bit ratio of two 16-bit signed inputs.
_FixMul	Multiplies two 32-bit Fixed inputs and produces a 32-bit Fixed result.
_FracMul	Multiplies two Frac inputs and produces a Frac result.
_FixDiv	Divides two Fixed inputs and produces a Fixed result.
_FracDiv	Divides two Frac inputs and produces a Frac result.
_FixRound	Takes a Fixed input and produces a rounded integer result.
_FracSqrt	Takes a Frac input and produces a Frac square root.
_FracCos	Takes a Frac input and returns its cosine.
_FracSin	Takes a Frac input and returns its sine.
_FixATan2	Takes two inputs and returns a Fixed arc tangent of their ratio.
_HiWord	Returns high word of input.
_LoWord	Returns low word of input.
_Long2Fix	Converts long integer to Fixed.
_Fix2Long	Converts Fixed to long integer.
_Fix2Frac	Converts Fixed to Frac.
_Frac2Fix	Converts Frac to Fixed.
_Fix2X	Converts Fixed to extended.
_Frac2X	Converts Frac to extended.
_X2Fix	Converts extended to Fixed.
_X2Frac	Converts extended to Frac.

Conversion Routines

These routines convert between a binary value and an ASCII character string representing that value. The binary value can be either a two-byte integer or a four-byte integer. The character string can be in either hexedecimal or decimal form.

_Int2Hex	Takes a 2-byte unsigned integer and produces an ASCII string representing the value in hexadecimal format.
_Long2Hex	Takes a 4-byte unsigned integer and produces an ASCII string representing the value in hexadecimal format.
_Hex2Int	Takes an ASCII string representing a hexadecimal value and produces a 2-byte unsigned integer.
_Hex2Long	Takes an ASCII string representing a hexadecimal value and produces a 4-byte unsigned integer.
_Int2Dec	Takes a 2-byte integer and produces an ASCII string representing the value in decimal format.
_Long2Dec	Takes a 4-byte integer and produces an ASCII string representing the value in decimal format.
_Dec2Int	Takes an ASCII string representing a decimal value and produces a 2-byte integer.
_Dec2Long	Takes an ASCII string representing a decimal value and produces a 4-byte integer.
_HexIt	Takes a 2-byte unsigned integer and returns a 4-byte ASCII string representing the value in hexadecimal format.

Memory Manager

The Memory Manager on the Cortland is responsible for allocating blocks of memory to programs. The Manager does the bookkeeping of what memory is being used and keeps track of who owns various blocks of memory. The Memory Manager routines are summarized below:

Memory Manager Housekeeping Routines

_MMBootInit	Initializes Memory Manager at boot time; must never be made by an application.
_MMAppInit	Made by an application when it starts up.
_MMAppQuit	Made by an application when it terminates.
_MMGetVersion	Returns the version of the Memory Manager.
_MMReset	Used by the system upon Reset. Should not be used by an application.
_MMStatus	Returns status indicating whether the Memory Manager is active. The Memory Manager is always active.

Memory Allocation Routines

_NewHandle	Creates a new block and returns the handle to the block.
_ReAllocHandle	Reallocates a block that was purged.
_DisposHandle	Purges a specified unlocked block and deallocates the handle.
_DisposAll	Discards all of the handles belonging to userID.

_PurgeHandle	Purges a specified unlocked block, but does not deallocate the handle.
_PurgeAll	Purges all of the purgeable blocks for a specified owner.

Block Information and Fress Space Routines

_GetHandleSize	Returns the size of a block.
_SetHandleSize	Changes the size of a specified block.
_CompactMem	Compacts memory space.
_FreeMem	Returns the total number of free bytes in memory.
_MaxBlock	Returns the size of the largest free block in memory, not counting memory that can be freed by purging or compacting.
_TotalMem	Returns the size of all memory, including the main 256K.

Locking and PurgeLevel Routines

_HLock	Locks a block specified by a handle.
_HLockAll	Locks all of the blocks owned by an owner.
_HUnLock	Unlocks a block specified by a handle
_HUnLockAll	Unlocks all of the blocks owned by an owner.
_SetPurge	Sets the purge level of a specified block.
_SetPurgeAll	Sets the purge level of all blocks owned by a specified owner.

Miscellaneous Memory Manager Routines

_BlockMove	Copies a specified number of bytes from a source to a destination.
_PtrToHand	Not yet implemented.
_HandToPtr	Not yet implemented.
_HandToHand	Not yet implemented.

Menu Manager

The Menu Manager supports the use of menus which can be part of the Cortland user interface. Menus allow users to examine all choices available to them at any time without being forced to choose one of them, and without having to remember command words or special keys. The Cortland user simply positions the cursor in the menu bar and presses the mouse button over a menu title. The application then calls the Menu Manager, which highlights selected title (by redrawing it with its InvertColor) and "pulls down" the menu below it. As long as the mouse button is held down, the menu is displayed. Dragging through the menu causes each of the menu items (commands) in it to be highlighted in turn. If the mouse button is released over an item, that item is "chosen". The item blinks briefly to confirm the choice, and the menu disappears.

When the user chooses an item, the Menu Manager tells the application which item was chosen, and the application performs the corresponding action. When the application completes the action, it removes the highlighting from the menu title, indicating to the user that the operation is complete.

If the user moves the cursor out of the menu with the mouse button held down, the menu remains visible, though no menu items are highlighted. If the mouse button is released outside the menu, no choice is made: The menu just disappears and the application takes no action. The user can always look at a menu without causing any changes in the document or on the screen.

The Menu Manager routines are summarized below:

_BootMmgr	Initializes Menu Manager at boot time.
_InitMenus	Initalizes the Menu Manager at application start-up
_TermMenus	Closes the Menu Manager's port and frees any allocated menus.
_MMgrVersion	Returns the version of the Menu Manager.
_MMgrStatus	Returns status indicating whether the Menu Manager is active.
_MMgrSpare	Reserved for future use.
_GetMenuPt	Returns a pointer to a menu record.
_SetTitleWidth	Sets the width of a title. The title width defines the area where the user can select the menu, and defines the area that is inverted when the title is highlighted.
_GetTitleWidth	Returns the width of a menu title. The width defines the area where the user can select the menu, and defines the area that is inverted when the title is highlighted.
_SetMenuFlag	Sets the menu to a specified state.
_GetMenuFlag	Returns the menu flag for a specified menu(see MENU RECORDS for definition).
_SetMenuTitle	Specifies the title for a menu.
_SetMenuTitle	Specifies the title for a menu.
_GetMenuTitle	Returns a pointer to the title of a menu
_SetMenuID	Specifies a new menu number.
_GetItemPtr	Returns a pointer to an item record.
_SetItem	Specifies the name for an item.
_GetItem	Returns a pointer to the name of an item.
_EnableItem	Sets a specified item to display normally and allows it to be selected.
_DisableItem	Sets a specified item to display in dimmed characters and does not allow it to be selected.
_CheckItem	Sets a menu item to display or not display a check mark to the left of the item.
_SetItemMark	Sets a specified character to display or not display to the left of the item.
_GetItemMark	Returns the current character which displays to the left of a specified menu item.
_SetItemStyle	Sets the text style for a specified menu item.
_GetItemStyle	Returns the text style for a specified menu item.
_SetItemFlag	Sets a specified item number to be underlined or not underlined, and sets the style of highlighting.
_GetItemFlag	Returns whether or not a specified item is underlined and highlighted.
_SetItemID	Specifies the ID number of a menu item.
_SetItemBlink	Determines how many times all menu items should blink when selected.
_MNewRes	Adjust for a new screen resolution and redraws the current system menu bar in the new resolution.

_BootMmgr	Called at boot time.
_InitMenus	Initializes the Menu Manager at application startup.
_TermMenus	Closes the Menu Manager's port and frees any allocated menus.
_MMgrVersion	Returns the version of the Menu Manager.
_MmgrReset	This call does nothing, but is one of the standard housekeeping calls of the toolbox. If a program makes the call, the call will return without an error.
_InitPalette	Reinitializes the palettes needed for the color Apple logo in the system menu bar.
_NewMenu	Allocates space for a menu list and its item.
_DisposeMenu	Frees the memory allocated by NewMenu. The menu list will no longer be usable.
_FixMenuBar	Computes standard sizes for the menu bar and menus.
_CalcMenuSize	Sets menu dimensions, either manually or automatically.
_MenuSelect	Draws highlighted titles, pulls down menus, and handles user interaction when a mouse button is clicked on a menu bar.
_MenuKey	Maps a character to the associated menu and item for that character.
_CheckFallDown	Checks if the current cursor position is inside the menu bar's fall down area.
_MenuRefresh	Refreshes the screen under the menu
_DrawMenuBar	Draws the current menu bar, along with any menu titles on the bar.
_HiliteMenu	Draws a menu's title using the menu bar's BarColor
_FlashMenuBar	Flashes the entire current menu bar by first redrawing it using user-specified colors then redrawing it again using normal colors.
_InsertMenu	Inserts a specified menu into the menu list after a specified menu item, or at the front of the list if InsertAfter is zero.
_DeleteMenu	Removes a specified menu from the menu list.
_InsertItem	Inserts a menu item into a menu after a specified menu item, or at the front of the list if insert afteritem is zero.
_DeleteItem	Removes a specified item from a specified menu.
_SetSysBar	Sets a new system bar.
_GetSysBar	Returns a pointer to the current system menu bar.
_SetMenuBar	Sets the current menu bar.
_GetMenuBar	Returns a pointer to the current menu bar.
_CountMItems	Returns the number of items, including any dividing lines, in a specified menu.
_SetFallArea	Specifies the height in pixels of the FallDown area.
_GetFallArea	Returns the height of the FallDown area.
_SetBarColors	Sets the normal, inverse, and outline colors of the current menu bar.
_GetBarColors	Returns the colors for the current menu bar.
_SetTitleStart	Sets the starting position for the leftmost title within the current menu bar.
_GetTitleStart	Returns the starting position for the leftmost title within the current menu bar.

Miscellaneous Tools

There are a number of tools that do not fall easily into one logical category. They are grouped here under "Miscellaneous Tools".

Miscellaneous Tools Housekeeping Routines

The Miscellaneous Tools housekeeping routines, summarized below, provide the standard housekeeping routines of the tool sets. The routines allow the Miscellaneous Tools to be initialized and handled as a Cortland Tool Set.

_MTBootInit	Initializes Heartbeat interrupt chain link pointer, clears TickCounter and Heartbeat task link pointer, and sets Mouse flag to"NOT FOUND".
_MTVersion	Returns the version of the Miscellaneous Tools.
_MTReset	Clears the Heartbeat queue link pointer and sets Mouse flag to"NOT FOUND".
_MTStatus	Returns status indicating whether the Miscellaneous Tools are active.

Battery RAM Routines

The Battery Ram routines, summarized below, allow the Battery RAM, which is RAM powered by battery and thus non-volatile, to be read or written.

_WriteBRAM	Writes data from memory to the Battery RAM.
_ReadBRAM	Reads data from the Battery RAM into memory.
_WriteBParam	Writes data to a specified parameter in the Battery RAM.
_ReadBParam	Reads data from a specified parameter in the Battery RAM.

Clock Routines

The clock routines summarized below allow the clock to be set or read. Setting the clock requires that the time be passed as an input parameter in a hex format. Two tools are provided for reading the clock. One returns time in a hex format; the other returns time in an ASCII format.

_ReadTimeHex	Returns current time in Hex format.
_WriteTimeHex	Sets the current time.
_ReadAsciiTime	Returns current time in ASCII format.

Vector Initialization Routines

The vector initialization routines summarized below allow an application to set or get the current vector for the interrupt handlers.

_SetVector	Sets the vector address for a specified interrupt manager or handler.
_GetVector	Returns with the vector address for a specified interrupt manager or handler.

HeartBeat Routines

The heartbeat routines summarized below allow a vector to be installed or removed from the HeartBeat Interrupt service queue. These routines are summarized below:

_SetHeartBeat	Installs a specified task into the HeartBeat Interrupt Service queue.
_DelHeartBeat	Deletes a specified task from the HeartBeat Interrupt Service queue.
_ClrHeartBeat	Removes all tasks from the HeartBeat Interrupt Service queue.

ID Tag Manager Routines

The ID Tag Manager tools summarized below are used to create and delete ID tags, and inquire about the status of an ID Tag. The ID Tag marks memory segments as belonging to a specific application or desk accessory.

_GetNewID	Creates a new ID tag.
_DeleteID	Deletes all references to a specified ID tag.
_StatusID	Removes all tasks from the HeartBeat Interrupt Service queue.

Mouse Routines

The Mouse routines summarized below interface with the mouse firmware. They can be used to set mouse mode, inquire about mouse status, read the clamp and position values, and set the clamp values.

_ReadMouse	Returns mouse position, status, and mode.
_InitMouse	Initializes mouse clamp values and clears mouse mode and status.
_SetMouse	Sets the mode for the mouse.
_HomeMouse	Positions the mouse at the minimum clamp position.
_ClearMouse	Sets the X and Y axis to $0000 if the minimum clamp values are negative, or to the minimum clamp position if the clamps are positive.
_ClampMouse	Sets the clamp values to new values, and then sets the mouse position to the minimum clamp values.
_GetMouseClamp	Returns the current values of the mouse clamps.
_PosMouse	Positions the mouse to specified coordinates.

_ServeMouse	Returns mouse interrupt status.

Absolute Clamp Routines

The absolute clamp routines, Set AbsClamp and GetAbsClamp, provide support for absolute devices such as graphics tablets.

SetAbsClamp	Sets the clamp values to new values.
GetAbsClamp	Returns the current values of the absolute device clamps.

Additional Miscellaneous Tools

_GetAddr	Returns an address of a byte, word, or long parameter referenced by the firmware.
_IntSource	Enables or disables certain interrupts.
_FWentry	Allows certain Cortland emulation mode entry points to be supported from full native mode.
_Tick Counter	Returns the current value of the tick counter.
_PackBytes	Packs bytes into a special format which uses less storage space.
_UnPackBytes	Unpacks data from the packed format used by PackBytes.
_Munger	Manipulates bytes in a string of bytes.
_GetIRQenbl	Returns the hardware interrupt enable states for interrupt sources that can be controlled by the Miscellaneous Tool set.

QuickDraw

QuickDraw is the part of the Toolbox that allows Cortland programmers to perform highly complex graphic operations very easily and very quickly. QuickDraw helps you draw many different things on the Cortland screen, including text, lines, rectangles, ovals, roundrects, wedges, polygons and regions. QuickDraw also has some other abilities you won't find in many other graphics packages. These abilities take care of most of the housekeeping; the trivial but time-consuming overhead that's necessary to keep things in order, including the ability to define manu distinct ports on the screen, full and complete clipping to arbitrary areas, and offscreen drawing. The QuickDraw Tools are described below.

QuickDraw Housekeeping Functions

_QDBootInit	Initializes QuickDraw II at boot time. The function puts the address of the cursor update routine into the bank E1 vectors. An application should never make this call.
_QDStartup	Initializes Quickdraw II, sets the current port to the standard port, and clears the screen.
_QDShutDown	Frees up any buffers that were allocated. This call can fail if QuickDraw is not active when the call was made.

_QDVersion	Returns the version of QuickDraw II.
_QDStatus	Returns whether or not QuickDraw is active.

QuickDraw Global Environment Calls

_GetStandardSCB	Returns a copy of the standard SCB in the low byte of the word.
_SetMasterSCB	Sets the master SCB to the specified value (only the low byte is used).
_GetMasterSCB	Returns a copy of the master SCB (only the low byte is valid).
_InitColorTable	Returns a copy of the standard color table for the current mode.
_SetColorTable	Sets a color table to specified values.
_GetColorTable	Fills a color table with the contents of another color table.
_SetColorEntry	Sets the value of a color in a specified color table.
_GetColorEntry	Returns the value of a color in a specified color table .
_SetSCB	Sets the scan line control byte (SCB) to a specified value.
_GetSCB	Returns the value of a specified scan line control byte (SCB).
_SetAllSCBs	Sets all scan line control bytes (SCBs) to a specified value.
_SetSysFont	Tells QuickDraw to use the font passed as a system font.
_GetSysFont	Returns a handle to the current system font.
_SetMaxWidth	Tells QuickDraw to resize its internal buffers based on this new MaxWidth.
_GetMaxWidth	Returns the current MaxWidth.
_SetTBSize	Tells QuickDraw to resize the text buffer based on the values passed.
_ForceTBSize	Tells QuickDraw to resize the text buffer based on the values passed.
_SaveTBDims	Returns a code for the current dimensions of the text buffer.
_RestoreTBDims	Resizes the text buffer based on the TBDimCode.
_ClearScreen	Sets the words in the screen memory to the value passed.
_GrafOn	Turns on the super hi-res graphics mode.
_GrafOff	Turns off the super hi-res graphics mode.

QuickDraw GrafPort Calls

_OpenPort	Initializes specified memory locations as a standard port and allocates new VisRgn and ClipRgn.
_InitPort	Initializes specified memory locations as a standard port.
_ClosePort	Deallocates the memory associated with a port.
_SetPort	Makes the specified port the current port.
_GetPort	Returns the handle to the current port.
_SetPortLoc	Sets the current port's map information structure to the specified location information.
_GetPortLoc	Gets the current port's map information structure and puts it at the address indicated.
_SetPortRect	Sets the current port's port rectangle to the specified rectangle.
_GetPortRect	Returns the current port's map port rectangle.
_SetPortSize	Changes the size of the current GrafPort's PortRect.

_MovePortTo	Changes the location of the current GrafPort's PortRect.
_SetOrigin	Adjusts the contents of PortRect and BoundsRect so that the upper left corner of PortRect is set to the specified point.
_SetClip	Sets the clip region to the region passed by using CopyRgn.
_GetClip	Copies the Clip Region to the region passed. The region must have been created earlier with a new rgn call.
_ClipRect	Changes the clip region of the current GrafPort to a rectangle equivalent to a given rectangle.
_HidePen	Decrements the pen level.
_ShowPen	Increments the pen level.
_GetPen	Returns the pen location.
_SetPenState	Sets the pen state in the GrafPort to the values passed.
_GetPenState	Returns the pen state from the GrafPort.
_SetPenSize	Sets the current pen size to the specified pen size.
_GetPenSize	Returns the current pen size at the place indicated.
_SetPenMode	Sets the current pen mode to the specified pen mode.
_GetPenMode	Sets the current pen mode to the specified pen mode.
_SetPenPat	Sets the current pen pattern to the specified pen pattern.
_GetPenPat	Returns the current pen pattern at the specified location.
_SetSolidPenPat	Sets the pen pattern to a solid pattern using the specified color.
_SetPenMask	Sets the pen mask to the specified mask.
_GetPenMask	Returns the pen mask at the specified location.
_SetBackPat	Sets the background pattern to the specified pattern.
_GetBackPat	Returns the background pattern at the specified location.
_SetSolidBackPat	Sets the background pattern to a solid pattern using the specified color.
_SolidPattern	Sets the specified pattern to a solid pattern using the specified color.
_PenNormal	Sets the pen state to the standard state (PenSize = 1,1; PenMode = copy; PenPat = Black; PenMask = 1's). The pen location is not changed.
_MoveTo	Moves the current pen location to the specified point.
_Move	Moves the current pen location by the specified horizontal and vertical displacements.
_SetFont	Sets the current font to the specified font.
_GetFont	Returns a handle to the current font.
_SetFontID	Sets the fontID field in the GrafPort.
_GetFontID	Returns the FontID field in the GrafPort.
_GetFontInfo	Returns information about the current font in the specified record.
_GetFGSize	Returns the size of the font globals record.
_GetFontGlobals	Returns information about the current font in the specified record.
_SetFontFlags	Sets the font flags to the specified value.
_GetFontFlags	Returns the current font flags.
_SetTextFace	Sets the text face to the specified value.
_GetTextFace	Returns the current text face.
_SetTextMode	Sets the text mode to the specified value.
_GetTextMode	Returns the current text mode.
_SetSpaceExtra	Sets the space extra field in the grafport to the specified value.
_GetSpaceExtra	Returns the space extra field from the grafport.
_SetCharExtra	Sets the char extra field in the grafport to the specified value.

_GetSpaceExtra	Returns the space extra field from the grafport.
_SetForeColor	Sets the foreground color field in the grafport to the specified value.
_GetForeColor	Returns the current foreground color from the grafport.
_SetBackColor	Sets the background color field in the grafport to the specified value.
_GetBackColor	Returns background color field from the grafport.
_GetFGSize	Returns the size of the font globals record
_GetFontGlobals	Fills the font globals record with the appropriate info.
_SetClipHandle	Sets the clip region handle field in the grafport to the value passed.
_GetClipHandle	Returns a copy of the handle to the ClipRgn.
_SetVisRgn	Sets the vis region to the region passed by using CopyRgn.
_GetVisRgn	Copies the contents of the VisRgn into the region passed.
_SetVisHandle	Sets the clip region handle field in the graf port to the value passed.
_GetVisHandle	Returns a copy of the handle to the VisRgn.
_SetPicSave	Sets the picsave field to the value passed. This is an internal routine that should not be used by application programs.
_GetPicSave	Returns the contents of the PicSave field in the GrafPort.
_SetRgnSave	Sets the RgnSave field to the value passed. This is an internal routine that should not be used by application programs.
_GetRgnSave	Returns the contents of the RgnSave field in the GrafPort.
_SetPolySave	Sets the PolySave field to the value passed. This is an internal routine that should not be used by application programs.
_GetPolySave	Returns the contents of the PicSave field in the GrafPort.
_SetGrafProcs	Sets the GrafProcs field to the value passed.
_GetGrafProcs	Returns the contents of the Pointer to the GrafProcs record associated with the GrafPort.
_SetUserField	Sets the UserField field in the GrafPort to the value passed.
_GetUserField	Returns the contents of the UserField field in the GrafPort.
_SetSysField	Sets the SysField field to the value passed. This is an internal routine that should not be used by application programs.
_GetSysField	Returns the contents of the SysField field in the GrafPort.

Drawing Calls

_LineTo	Draws a line from the current pen location to the specified point.
_Line	Draws a line from the current pen location to a new point specified

Drawing Rectangles

_FrameRect	Draws the boundary of the specified rectangle with the current pattern and pen size.
_PaintRect	Paints (fills) the interior of the specified rectangle with the current pen pattern.

_EraseRect	Erases the interior of the specified rectangle with the background pattern.
_InvertRect	Inverts the pixels in the interior of the specified rectangle.
_FillRect	Paints (fills) the interior of the specified rectangle with the specified pattern.

Drawing Regions

_FrameRgn	Draws the boundary of the specified region with the current pattern and current pen size.
_PaintRgn	Paints (fills) the interior of the specified region with the current pen pattern.
_EraseRgn	Fills the interior of the specified region with the background pattern.
_InvertRgn	Inverts the pixels in the interior of the specified region.
_FillRgn	Fills the interior of the specified region with the specfied pattern.

Drawing Polygons

_FramePoly	Frames the specified polygon.
_PaintPoly	Paints the specified polygon.
_ErasePoly	Erases the specified polygon.
_InvertPoly	Inverts the specified polygon.
_FillPoly	Paints the specified polygon.

Drawing Ovals

_FrameOval	Draws the boundary of the oval enscribed in the specified rectangle with the current pattern and pen size.
_PaintOval	Paints (fills) the interior of the oval enscribed in the specified rectangle with the current pen pattern.
_EraseOval	Erases the interior of the oval enscribed in the specified rectangle with the background pattern.
_InvertOval	Inverts the pixels in the interior of the oval enscribed in the specified rectangle.
_FillOval	Paints (fills) the interior of the oval enscribed in the specified rectangle with the specified pattern.

Drawing RoundRects

_FrameRRects	Draws the boundary of the roundrect enscribed in the specified rectangle with the current pattern and pen size.
_PaintRRect	Paints (fills) the interior of the roundrect enscribed in the specified rectangle with the current pen pattern.
_EraseRRect	Erases the interior of the roundrect enscribed in the specified rectangle with the background pattern.
_InvertRRect	Inverts the pixels in the interior of the roundrect enscribed in the specified rectangle.

_FillRRect	Paints (fills) the interior of the roundrect enscribed in the specified rectangle with the specified pattern.

Drawing Arcs

_FrameArc	Draws the boundary of the arc enscribed in the specified rectangle with the current pattern and pen size.
_PaintArc	Paints (fills) the interior of the arc enscribed in the specified rectangle with the current pen pattern.
_EraseArc	Erases the interior of the arc enscribed in the specified rectangle with the background pattern.
_InvertArc	Inverts the pixels in the interior of the arc enscribed in the specified rectangle.
_FillArc	Paints (fills) the interior of the arc enscribed in the specified rectangle with the specified pattern.

Transferring Pixels

_ScrollRect	Shifts the pixels inside the intersection of the specified rectangle, VisRgn, ClipRgn, PortRect, and BoundsRect.
_PaintPixels	Transfers a region of pixels.

Drawing and Measuring Text

_DrawChar	Draws the specified character.
_DrawText	Draws the spedified text.
_DrawString	Draws the spedified string.
_DrawCString	Draws the spedified C-String.
_CharWidth	Returns the width of the specified character.
_TextWidth	Returns the width of the specified text.
_StringWidth	Returns the width of the specified string.
_CStringWidth	Returns the width of the specified C-String.
_CharBounds	Fills in the specified rectangle with bounds of character.
_TextBounds	Fills in the specified rectangle with bounds of text.
_StringBounds	Fills in the specified rectangle with bounds of string.
_CStringBounds	Fills in the specified rectangle with bounds of CString.

QuickDraw Utility Routines

Calculations With Rectangles
Calculations With Points
Calculations With Regions
Calculations with Polygons
Mapping and Scaling Utilities
Miscellaneous Utilities

Customizing QuickDraw Operations

_SetStdProcs	Sets up the specified record of pointers.
_StdText	Draws standard text.
_StdLine	Draws standard lines.
_StdRect	Draws standard rects.
_StdRRect	Draws standard round rects.
_StdOval	Draws standard ovals.
_StdArc	Draws standard arcs.
_StdPoly	Draws standard polys.
_StdRgn	Draws standard regions.
_StdPixels	Draws standard pixels.
_StdComment	Does standard comments for pictures.
_StdTxMeas	Does standard text measuring.
_StdGetPic	Does standard retieval from picture record.
_StdPutPic	Does standard storage into picture record.
_SetIntUse	Tells QuickDraw's cursor drawing code whether or not it should use scan line interrupts.
_GetAddress	Returns the address of the specified table.

Handling Cursors

_SetCursor	Sets the cursor to the image passed in the cursor record.
_GetCursorAdr	Returns a pointer to the current cursor record.
_HideCursor	Decrements the cursor level. A cursor level of zero indicates the cursor is visible; a cursor level less than zero indicates the cursor is not visible.
_ShowCursor	Increments the cursor level unless it is already zero. A cursor level of zero indicates the cursor is visible; a cursor level less than zero indicates the cursor is not visible.
_ObscureCursor	Hides the cursor until the mouse moves. This tool is used to get the cursor out of the way of typing.
_InitCursor	Reinitializes the cursor.

SANE Tools

To be supplied

Scheduler

Information not yet available

Sound Manager

The Sound Manager gives developers the ability to access the Sound hardware without having to know specific hardware I/O addresses.

Sound Manager calls (other than the standard housekeeping routines) can be broken down into two groups. The first group of calls is made through the normal tool call mechanism, with parameters being passed to and from the called routines on the stack. The second group of routines are low-level routines which, unlike most tool calls, pass their parameters in registers. The Sound Manager routines are summarized below:

Sound Manager Housekeeping Routines

_SoundBootInit	Initializes the Sound Manager. Must not be made by an application.
_SoundStartup	Initializes a work area to be used by the sound routines.
_SoundShutdown	Shuts down the Sound Manager.
_SoundVersion	Returns the version of the Sound Manager.
_SoundReset	Stops sound from all generators. Must not be made by an application.
_SoundToolStatus	Returns status indicating whether the Sound Manager is active.

Sound Manager Tools

_WriteRamBlock	Writes a specified number of bytes from system RAM into DOC RAM.
_ReadRamBlock	Reads any number of locations from the 64K DOC ram area into a user-specified buffer.
_GetTableAddress	Returns the jump table address for the low-level routines
_GetSoundVolume	Reads the volume setting for a generator.
_SetSoundVolume	Changes the volume setting for the volume registers in the DOC, or changes the system volume.
_FFStartSound	Enables the DOC to start generating sound on a particular generator.
_FFStopSound	Stops sound from specified generators.
_FFSoundStatus	Returns the status of all fifteen sound generators.
_FFGeneratorStatus	Reads the first two bytes of the Generator Control Block corresponding to a specified generator.
_SetSoundMIRQV	Sets up the entry point into the sound interrupt handler.
_SetUserSoundIRQV	Sets up the entry point for a user synthesizer interrupt handler.
_FFSoundDoneStatus	Returns the current Free Form synthesizer sound-playing status.

Sound Manager Low-Level Routines

_Read Register	Reads any register within the DOC.
_Write Register	Writes a onebyte parameter to any register in the DOC chip.
_Read RAM	Reads a specified Ensoniq RAM location.
_Write RAM	Writes a one-byte value to any specified Ensoniq RAM location.
_Read Next	Reads the next location pointed to by the Sound GLU address register.

_Write Next	Writes one byte of data to the next DOC register or RAM location, depending on the setting of the Sound GLU control register.

Text Tools

The Text Tool Set provides an interface between Cortland character device drivers, which must be executed in emulation mode, and new applications running in native mode. It also provides a means of redirection of I/O through RAM-based drivers. The Text Tools make it possible to deal with the text screen without switching modes and moving to bank zero. Dispatches to RAM-based drivers will occur in full native mode.

Text Tools Housekeeping Routines

The Text Tools housekeeping routines summarized below provide the standard housekeeping routines of the tool sets. The routines allow the Text Tools to be initialized and dealt with as a Cortland Tool Set.

_MTBootInit	Initializes Heartbeat interrupt chain link pointer, clears Tick Counter and Heartbeat task link pointer, and sets Mouse flag to"NOT FOUND".
_MTVersion	Returns the version of the Miscellaneous Tools.
_MTReset	Clears the Heartbeat queue link pointer and sets Mouse flag to "NOT FOUND".
_MTStatus	Returns status indicating whether the Miscellaneous Tools are active.
_TextBootInit	Called at boot time. This routine sets up the default device parameters as follows:
_TextReset	Resets the device parameters to the defaults
_SetInGlobals	Sets the global parameters for the input device.
_SetOutGlobals	Sets the global parameters for the output device.
_SetErrGlobals	Sets the global parameters for the error output device.
_GetInGlobals	Returns with the current values for the input device global parameters.
_GetOutGlobals	Returns with the current values for the Output device global parameters.
_GetErrGlobals	Returns with the current values for the Error Output device global parameters.
_SetInputDevice	Sets the input device to the specified devicetype.
_SetOutputDevice	Sets the output device to the specified devicetype.
_SetErrorDevice	Sets the error output device to the specified devicetype. The routine returns an
_GetInputDevice	Returns the type of driver installed as the input device.
_GetErrorDevice	Returns the type of driver installed as the error output device.
_InitTextDev	Initializes the specified text device.
_CtrlTextDev	Passes the control code to the specified text device. The control codes passed
_StatusTDev	Executes a status call to the specified text device.
_WriteChar	Combines specified character with the output global AND mask and global OR mask, and writes character to the output text device.

_ErrWriteChar	Combines specified character with the output global AND mask and global OR mask, and writes character to the error output text device.
_WriteLine	Combines a pointed-to character string (first byte of string specifies length) with the output global masks, and then writes the string to the output text device.
_ErrWriteLine	Combines a pointed-to character string (first byte of string specifies length) with the output global masks, and then writes the string to the error output text device.
_WriteString	Combines a pointed-to character string (first byte of string specifies length) with the output global masks, and then writes the string to the output text device.
_ErrWriteString	Combines a pointed-to character string (first byte of string specifies length) with the output global masks, and then writes the string to the error output text device.
_WriteBlock	Combines a character string at textptr +offset (with a length specified by count) with the output global masks, and then writes the string to the output text device.
_ErrWriteBlock	Combines a character string at textptr +offset (with a length specified by count) with the output global masks, and then writes the string to the error output text device.
_WriteCString	Combines a pointed-to character string (string terminates with $00) with the output global masks, and then writes the string to the output text device.
_ReadLine	Reads a character string from the input text device, combines it with the input global masks, and writes the string to the memory location starting at bufferpointer.

Tool Locator

The Tool Locator is the tool that allows tools and applications to communicate. As such, it represents a special case among the tool sets; you need to know about it if you are considering writing your own tool set. You don't need to know about the Tool Locator if you are simply using the other Cortland tools that Apple provides. The Tool Locator routines are summarized below:

Tool Locator Houskeeping Routines

_TLBootInit	Initializes the Tool Locator and all other ROM-based Tool Sets.
_TLVersion	Returns the version of the Tool Locator.
_TLReset	Calls the reset routine of every tool set in the system.

Tool Locator Routines

_GetTsPtr	Returns pointer to the Function Pointer Table of the specified tool set.
_SetTSPtr	Installs the pointer to a Function Pointer Table in the appropriate Tool Pointer Table.
_GetFuncPtr	Returns pointer to the specified function in the specified Tool Set.

_GetWAP	Gets the pointer to the work area for the specified module.
_SetWAP	Sets the pointer to the work area for the specified module.
_BootInit	Initializes the Tool Locator and all other ROM-based Tool Sets.
_AppInit	Does nothing.
_AppEnd	Does nothing

ProDOS Macros

Information not yet available

Cortland Shell Macros

Will these exist?

Utility Macros

The Utility macros are provided in addition to the Cortland Toolbox as useful aids in writing Assembly code for the Cortland computer. The Utility macros include:

pullword	Pull 2 bytes from stack
pulllong	Pull long (4 bytes) from stack
pull1	Pull 1 byte from the stack
pull3	Pull 3 bytes from the stack
pullxy	Pull long (4 bytes) from the stack using X and Y
pullay	Pull long (4 bytes) from the stack into A and Y
pullx	Pull from stack using X
pushword	Push 2 bytes onto the stack
pushlong	Push long (4 bytes) onto the stack
push1	Push 1 byte onto the stack
push3	Push 3 bytes onto the stack
pushxy	Push long (4 bytes) onto the stack from X and Y
pushay	Push long (4 bytes) onto the stack from A and Y
lday	Load A and Y (4 bytes)
stay	Store A and Y (4 bytes)
add	Add 2 byte integers
add4	Add 4 byte integers
sub	Subtract 2 byte integers
sub4	Subtract 4 byte integers
str	Generate a Pascal-type string
dp	Define a pointer
move1	Move 1 byte
moveword	Move 2 bytes
move3	Move 3 bytes
movelong	Move 4 bytes
zero	Zero block
asl4	Left shift 4 bytes
lsr4	Right shift 4 bytes

native	Enable native mode
emulation	Enable emulation mode
long	Set memory and registers for 16 bits
short	Set memory and registers for 8 bits
Check_Error	Error check
writech	Write a character
writestr	Write a string
writeln	Write a line

System Macro Files

The following information is valid as for CPW V1.0A3, April 29, 1986.

The Cortland macros reside in the following directories:

/CPW/ROM.MACROS	Tool calls and ProDOS calls
CPW/UTILITY.MACROS	Utility macros
/CPW/SANE.MACROS	SANE tool calls
/CPW/CPW.MACROS	Byte-works macros

The macros that make tool calls and ProDOS calls are located in the directory /CPW/ROM.MACROS. This directory contains the following files:

Tool locator	TL.MACROS
Memory Manager	MM.MACROS
Miscellaneous Tools	MT.MACROS
QuickDraw II macros	QD.MACROS
Desk Manager	DESK.MACROS
Event Manager	EM.MACROS
Scheduler	***not available yet***
FDB	***not available yet***
Sound Manager	***not available yet***
Integer Math	INT.MACROS
Text Tools	TEXT.MACROS
ProDOS macros	PRODOS.MACROS

Additionally, this directory contains the files HANDY.STUFF and the exec file MAKE.ALL. The file HANDY.STUFF contains a number of routines handy in making tool calls. The exec file MAKE.ALL joins all the files in this directory into a single file called MAKE.ALL. MAKE.ALL is the file you use as the source for the MACGEN utility.

The directory /CPW/SANE.MACROS contains the current release note and information on where you can obtain the SANE.MACROS.

The directory /CPW/CPW.MACROS contains the Byte Works macros. ***I'm assuming these will eventually go away. Right?***

Part III

Appendixes

This section includes five appendixes, followed by a glossary, bibliography, and index.

Appendix A is a list of the 65816 instruction set, addressing modes, opcodes and execution times.

Appendix B discusses how the Assembler uses memory and includes the Cortland Memory Map.

Appendix C is a comparison of Cortland Assembler and the ORCA/M Assembler.

Appendix D is a chart of the ASCII character set.

Appendix E is a summary of error messages generated by the Cortland Assembler.

This page is left intentionally blank

Appendix A

Instruction Set Summary

Opcode Hex	Mnemonic	Adressing Mode	# Of Bytes	# Of Cycles
00	BRK	Stack/Interrupt	2 **	7[9]
01	ORA	DP Indexed Indirect,X	2	6[1,2]
02	COP	Stack/Interrupt	2 **	7[9]
03	ORA	Stack Relative (also SR)	2 **	4[1]
04	TSB	Direct Page	2	5[2,5]
05	ORA	Direct Page (also DP)	2	3[1,2]
06	ASL	Direct Page (DP)	2	5[2,5]
07	ORA	DP Indirect Long	2	6[1,2]
08	PHP	Stack (Push)	1	3
09	ORA	Immediate	2 *	2[1]
0A	ASL	Accumulator	1	2
0B	PHD	Stack (Push)	1	4
0C	TSB	Absolute	3	6[5]
0D	ORA	Absolute	3	4[1]
0E	ASL	Absolute	3	6[5]
0F	ORA	Absolute Long	4	5[1]
10	BPL	Program Counter Relative	2	2[7,8]
11	ORA	DP Indirect Indexed, Y	2	5[1,2,3]
12	ORA	DP Indirect	2	5[1,2]
13	ORA	SR Indirect Indexed, Y	2	7[1]
14	TRB	Direct Page	2	5[2,5]
15	ORA	DP Indexed, X	2	4[1,2]
16	ASL	DP Indexed, X	2	6[2,5]
17	ORA	DP Indirect Long, Indexed	2	6[1,2]

18	CLC	Implied	1	2
19	ORA	Absolute Indexed, Y	3	4[1,3]
1A	INC	Accumulator	1	2
1B	TCS	Implied	1	2
1C	TRB	Absolute	3	6[5]
1D	ORA	Absolute Indexed, X	3	4[1,3]
1E	ASL	Absolute Indexed, X	3	7[5,6]
1F	ORA	Absolute Long Indexed, X	4	5[1]
20	JSR	Absolute	3	6
21	AND	DP Indexed Indirect, X	2	6[1,2]
22	JSR	Absolute Long	4	8
23	AND	Stack Relative (SR)	2	4[1]
24	BIT	Direct Page (DP)	2	3[1,2]
25	AND	Direct Page (DP)	2	3[1,2]
26	ROL	Direct Page (also DP)	2	5[2,5]
27	AND	DP Indirect Long	2	6[1,2]
28	PLP	Stack (Pull)	1	4
29	AND	Immediate	2*	2[1]
2A	ROL	Accumulator	1	2
2B	PLD	Stack (Pull)	1	5
2C	BIT	Absolute	3	4[1]
2D	AND	Absolute	3	4[1]
2E	ROL	Absolute	3	6[5]
2F	AND	Absolute Long	4	5[1]
30	BMI	Program Counter Relative	2	2[7,8]
31	AND	DP Indirect Indexed, Y	2	5[1,2,3]
32	AND	DP Indirect	2	5[1,2]
33	AND	SR Indirect Indexed, Y	2	7[1]
34	BIT	DP Indexed, X	2	4[1,2]
35	AND	DP Indexed, X	2	4[1,2]
36	ROL	DP Indexed, X	2	6[2,5]
37	AND	DP Indirect Long Indexed, Y	2	6[1,2]

38	SEC	Implied	1	2
39	AND	Absolute Indexed, Y	3	4[1,3]
3A	DEC	Accumulator	1	2
3B	TSC	Implied	1	2
3D	AND	Absolute Indexed, X	3	4[1,3]
3E	ROL	Absolute Indexed, X	3	7[5,6]
3F	AND	Absolute Long Indexed, X	4	5[1]
40	RTI	Stack /RTI	1	6[9]
41	EOR	DP Indexed Indirect, X	2	6[1,2]
*42	WDM	*reserved*	2	2[1,6]
43	EOR	Stack Relative (also SR)	2	4[1]
*44	MVP	Block Move	3	13
45	EOR	Direct Page (also DP)	2	3[1,2]
46	LSR	Direct Page (also DP)	2	5[2,5]
47	EOR	DP Indirect Long	2	6[1,2]
48	PHA	Stack (Push)	1	3[1]
49	EOR	Immediate	2*	2[1]
4A	LSR	Accumulator	1	2
4B	PHK	Stack (Push)	1	3
4C	JMP	Absolute	3	3
4D	EOR	Absolute	3	4[1]
4E	LSR	Absolute	3	6[5]
4F	EOR	Absolute Long	4	5[1]
50	BVC	Program Counter Relative	2	2[7,8]
51	EOR	DP Indirect Indexed, Y	2	5[1,2,3]
52	EOR	DP Indirect	2	5[1,2]
53	EOR	SR Indirect Indexed, Y	2	7[1]
*54	MVN	Block Move	3	13
55	EOR	DP Indexed, X	2	4[1,2]
56	LSR	DP Indexed, X	2	6[2,5]
57	EOR	DP Indirect Long Indexed Y	2	6[1,2]
58	CLI	Implied	1	2

59	EOR	Absolute Indexed, Y	3	4[1,3]
5A	PHY	Stack (Push)	1	3[10]
5B	TCD	Implied	1	2
5C	JMP	Absolute Long	4	4
5D	EOR	Absolute Indexed, X	3	4[1,3]
5E	LSR	Absolute Indexed, X	3	7[5,6]
5F	EOR	Absolute Long Indexed, X	4	5[1]
60	RTS	Stack (RTS)	1	6
61	ADC	DP Indexed Indirect, X	2	6[1,2,4]
62	PER	Stack (PC Relative Long)	3	6
63	ADC	Stack Relative (SR)	2	4[1,4]
64	STZ	Direct Page	2	3[1,2]
65	ADC	Direct Page (DP)	2	3[1,2,4]
66	ROR	Direct Page (also DP)	2	5[2,5]
67	ADC	DP Indirect Long	2	6[1,2,4]
68	PLA	Stack (Pull)	1	4[1]
69	ADC	Immediate	2*	2[1,4]
6A	ROR	Accumulator	1	2
6B	RTL	Stack (RTL)	1	6
6C	JMP	Absolute Indirect	3	5[11,12]
6D	ADC	Absolute	3	4[1,4]
6E	ROR	Absolute	3	6[5]
6F	ADC	Absolute Long	4	5[1,4]
70	BVS	Program Counter Relative	2	2[7,8]
71	ADC	DP Indirect Indexed, Y	2	5[1,2,3,4]
72	ADC	DP Indirect	2	5[1,2,4]
73	ADC	SR Indirect Indexed, Y	2	7[1,4]
74	STZ	Direct Page Indexed, X	2	4[1,2]
75	ADC	DP Indexed, X	2	4[1,2,4]
76	ROR	DP Indexed, X	2	6[2,5]
77	ADC	DP Indirect Long Indexed, Y	2	6[1,2,4]
78	SEI	Implied	1	2

79	ADC	Absolute Indexed, Y	3	4[1,3,4]
7A	PLY	Stack/Pull	1	4[10]
7B	TDC	Implied	1	2
7C	JMP	Absolute Indexed Indirect	3	6
7D	ADC	Absolute Indexed, X	3	4[1,3,4]
7E	ROR	Absolute Indexed, X	3	7[5,6]
7F	ADC	Absolute Long Indexed, X	4	51,4
80	BRA	Program Counter Relative	2	36
81	STA	DP Indexed Indirect, X	2	61,2
82	BRL	Program Counter Relative Long	3	4
83	STA	Stack Relative (also SR)	2	4[1]
84	STY	Direct Page	2	3[2,10]
85	STA	Direct Page (also DP)	2	3[1,2]
86	STX	Direct Page	2	3[2,10]
87	STA	DP Indirect Long	2	6[1,2]
88	DEY	Implied	1	2
89	BIT	Immediate	2*	2[1]
8A	TXA	Implied	1	2
8B	PHB	Stack (Push)	1	3
8C	STY	Absolute	3	4[10]
8D	STA	Absolute	3	4[1]
8E	STX	Absolute	3	4[10]
8F	STA	Absolute Long	4	5[1]
90	BCC	Program Counter Relative	2	2[7,8]
91	STA	DP Indirect Indexed, Y	2	6[1,2]
92	STA	DP Indirect	2	5[1,2]
93	STA	SR Indirect Indexed, Y	2	7[1]
94	STY	Direct Page Indexed, X	2	4[2,10]
95	STA	DP Indexed, X	2	4[1,2]
96	STX	Direct Page Indexed, Y	2	4[2,10]
97	STA	DP Indirect Long Indexed, Y	2	6[1,2]
98	TYA	Implied	1	2

99	STA	Absolute Indexed, Y	3	5[1]
9A	TXS	Implied	1	2
9B	TXY	Implied	1	2
9C	STZ	Absolute	3	4[1]
9D	STA	Absolute Indexed, X	3	5[1]
9E	STZ	Absolute Indexed, X	3	5[1]
9F	STA	Absolute Long Indexed, X	4	5[1]
A0	LDY	Immediate	2+	2[10]
A1	LDA	DP Indexed Indirect, X	2	6[1,2]
A2	LDX	Immediate	2+	2[10]
A3	LDA	Stack Relative (alsoSR)	2	4[1]
A4	LDY	Direct Page (also DP)	2	3[2,10]
A5	LDA	Direct Page (also DP)	2	3[1,2]
A6	LDX	Direct Page (also DP)	2	3[2,10]
A7	LDA	DP Indirect Long	2	6[1,2]
A8	TAY	Implied	1	2
A9	LDA	Immediate	2*	2[1]
AA	TAX	Implied	1	2
AB	PLB	Stack (Pull)	1	4
AC	LDY	Absolute	3	4[10]
AD	LDA	Absolute	3	4[1]
AE	LDX	Absolute	3	4[10]
AF	LDA	Absolute Long	4	5[1]
B0	BCS	Program Counter Relative	2	2[7,8]
B1	LDA	DP Indirect Indexed, Y	2	5[1,2,3]
B2	LDA	DP Indirect	2	5[1,2]
B3	DLA	SR Indirect Indexed, Y	2	7[1]
B4	LDY	DP Indexed, X	2	4[2,10]
B5	LDA	DP Indexed, X	2	4[1,2]
B6	LDX	DP Indexed, Y	2	4[2,10]
B7	LDA	DP Indirect Long Indexed, Y	2	6[1,2]
B8	CLV	Implied	1	2

B9	LD	Absolute Indexed, Y	3	4[1,3]
BA	TSX	Implied	1	2
BB	TYX	Implied	1	2
BC	LDY	Absolute Indexed, X	3	4[1,10]
BD	LDA	Absolute Indexed, X	3	4[1,3]
BE	LDX	Absolute Indexed, Y	3	4[3,10]
BF	LDA	Absolute Long Indexed, X	4	5[1]
C0	CPY	Immediate	2+	2[10]
C1	CMP	DP Indexed Indirect, X	2	6[1,2]
C2	REP	Immediate	2	3
C3	CMP	Stack Relative (also SR)	2	4[1]
C4	CPY	Direct Page (also DP)	2	3[2,10]
C5	CMP	Direct Page (also DP)	2	3[1,2]
C6	DEC	Direct Page (also DP)	2	5[2,5]
C7	CMP	DP Indirect Long	2	6[1,2]
C8	INY	Implied	1	2
C9	CMP	Immediate	2*	2[1]
CA	DEX	Implied	1	2
CB	WAI	Implied	1	3[15]
CC	CPY	Absolute	3	4[10]
CD	CMP	Absolute	3	4[1]
CE	DEC	Absolute	3	6[5]
CF	CMP	Absolute Long	4	5[1]
D0	BNE	Program Counter Relative	2	2[7,8]
D1	CMP	DP Indirect Indexed, Y	2	5[1,2,3]
D2	CMP	DP Indirect	2	5[1,2]
D3	CMP	SR Indirect Indexed, Y	2	7[1]
D4	PEI	Stack (Direct Page Indirect)	2	6[2]
D5	CMP	DP Indexed, X	2	4[1,2]
D6	DEC	DP Indexed, X	2	6[2,5]
D7	CMP	DP Indirect Long Indexed, Y	2	6[1,2]
D8	STP	Implied	1	2

D9	CMP	Absolute Indexed, Y	3	$4^{1,3}$
DA	PHX	Stack (Push)	1	$3^{1,10}$
DB	STP	Implied	1	3^{14}
DC	JMP	Absolute Indirect Long	3	6
DD	CMP	Absolute Indexed, X	3	$4^{1,3}$
DE	DEC	Absolute Indexed, X	3	$7^{5,6}$
DF	CMP	Absolute Long Indexed, X	4	5^{1}
E0	CPX	Immediate	2+	2^{10}
E1	SBC	DP Indexed Indirect, X	2	$6^{1,2,4}$
E2	SEP	Immediate	2	3
E3	SBC	Stack Relative (also SR)	2	$4^{1,4}$
E4	CPX	Direct Page (also DP)	2	$3^{2,10}$
E5	SBC	Direct Page (also DP)	2	$3^{1,2,4}$
E6	INC	Direct Page (also DP)	2	$5^{2,5}$
E7	SBC	DP Indirect Long	2	$6^{1,2,4}$
E8	INX	Implied	1	2
E9	SBC	Immediate	2*	$2^{1,4}$
EA	NOP	Implied	1	2
EB	XBA	Implied	1	3
EC	CPX	Absolute	3	4^{10}
ED	SBC	Absolute	3	$4^{1,4}$
EE	INC	Absolute	3	6^{5}
EF	SBC	Absolute Long	4	$5^{1,4}$
F0	BEQ	Program Counter Relative	2	$2^{7,8}$
F1	SBC	DP Indirect Indexed, Y	2	$5^{1,2,3,4}$
F2	SBC	DP Indirect	2	$5^{1,2,4}$
F3	SBC	SR Indirect Indexed, Y	2	$7^{1,4}$
F4	PEA	Stack (Absolute)	3	5
F5	SBC	DP Indexed, X	2	$4^{1,2,4}$
F6	INC	DP Indexed, X	2	$6^{2,5}$
F7	SBC	DP Indirect Long Indexed, Y	2	$6^{1,2,4}$
F8	SED	Implied	1	2

F9	SBC	Absolute Indexed, Y	3	$4^{1,3,4}$
FA	PLX	Stack/Pull	1	4^{10}
FB	XCE	Implied	1	2
FC	JSR	Absolute Indexed Indirect	3	8
FD	SBC	Absolute Indexed, X	3	$4^{1,3,4}$
FE	INC	Absolute Indexed, X	3	$7^{5,6}$
FF	SBC	Absolute Long Indexed, X	4	$5^{1,4}$

* Add 1 byte if m=0 (16-bit memory/accumulator)

** Opcode is 1 byte, but program counter value pushed onto stack is incremented by two, allowing for optional signature byte

+ Add 1 byte if x=0 (16-bit index registers)

1 Add 1 cycle if m=0 (16-bit memory/accumulator)

2 Add 1 cycle if low byte of Direct Page register is other than zero (DL<>0)

3 Add 1 cycle if adding index crosses a page boundary

4 Add 1 cycle if 65C02 and d=1 (decimal mode, 65C02)

5 Add 2 cycles if m=0 (16-bit memory/accumulator)

6 Subtract 1 cycle if 65C02 and no page boundary crossed

7 Add 1 cycle if branch taken

8 Add 1 more cycle if branch taken crosses page boundary on 6502, 65C02, or 65816/65802's emulation mode (e=1)

9 Add 1 cycle for 65802/65816 native mode (e=0)

10 Add 1 cycle if x=0 (16-bit index registers)

11 Add 1 cycle if 65C02

12 6502: If low byte of addr is $FF (i.e., addr is $xxFF): yields incorrect result

13 7 cycles per byte moved

14 Uses 3 cycles to shut the processor down; additional cycles are required by reset to restart it

15 Uses 3 cycles to shut the processor down; additional cycles are required by interrupt to restart it

16 Byte and cycle counts subject to change in future processors which expand WDM into 2-byte opcode portions of instructions of varying lengths

This page is left intentionally blank

Appendix B
Memory Management

Using Memory

The Assembler appropriates 4K of memory for each of the following:

- global symbol table
- local symbol table
- AINPUT string buffer
- symbolic parameter table

Macro buffers and source file buffers are allocated from remaining memory; each buffer is as large as the macro or source file.

The global and local symbol tables and symbolic parameter table are extensible in 4K increments, and are extended as needed for as long as memory is available.

Note that there must be enough physical memory to load a given source or macro file. With this the case, processing is considerably faster than with a system using virtual memory techniques to allow files as large as a disk will hold.

The Cortland Memory Map

To be supplied

This page is left intentionally blank

Appendix C

Comparison of Cortland and ORCA/M Assemblers

This appendix presents a comparison of the Cortland Assembler and the Assembler on which it was based, ORCA/M Version 4.0. For a comparison of the Shell, Editor, Linker and Debugger, refer to the *Cortland Programmers Workshop*. For a comparison of ProDOS and ProDOS 16, refer to the *Cortland Operating System Reference*.

Labels

The underline character

The ORCA/M Assembler ignores the underline character (_) in labels. For the Cortland Assembler, the underline character is significant. Thus, THISLABEL and THIS_LABEL are not synonymous. In addition, the underline character can be used anywhere that an alphabetic character can be used, including the first character in a label.

The tilde character

The tilde (~) character is allowed in labels, and can be used anywhere that an alphabetic character can be used. It is suggested that you reserve the underscore character (~) for use in system labels, so that you can develop libraries whose names will not interfere with names chosen by users of high-level languages. Use of the tilde replaces the ORCA/M convention of starting all system-level labels with SYS, and requiring that the user not start a label with those characters.

Label Length

Labels are now significant up to 255 characters. Previously, labels could be of any length, but only the first ten characters were significant.

Case Sensitivity

To specify case sensitivity, use the directive

```
CASE ON|OFF
```

The default is CASE OFF.

The directive

    OBJCASE ON

causes a label sent to the object module to be case-sensitive, whether or not it is treated as case-sensitive inside the assembler. Specifying

    OBJCASE OFF

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 behaviour is to be different from the local behaviour, specify the OBJCASE directive last.

Directives

New Directives

Six new directives have been added:

INSTIME
ABSADDR
PRIVATE
PRIVDATA
CASE
OBJCASE

Refer to Chapter 5, "Directives", for a description of these directives.

The Directives EQU and GEQU

For the Cortland Workshop Assembler, operands of the EQU and GEQU directives do not need to be constants or constant expressions.

The Directive DC

A new value type, E, has been added to the DC directive. The type E stands for extended floating point, and results in a SANE format 80-bit number.

MACROS

ORCA/M macros have been completely replaced by Cortland macros. The graphics, integer math, floating point and miscellaneous macros have been replaced by Cortland Tools macros, including the SANE macros, and utility macros. ProDOS macros have been updated to support the 65816.

The Shell

This section describes the changes that have been made to the Shell commands associated with using the Assembler. These commands are discusssed in Chapter 2, "Using the Assembler", and in Chapter 4, "The Shell", of the *Cortland Programmer's Workshop*.

Command Line Structure

The command line input for the Shell commands ASSEMBLE, ASML, AMSLG, LINK and ALINK has been restructured. The input filename, KEEP parameter, and NAMES parameter are unchanged. The parameters SYMBOL and LIST that were followed by the options ON or OFF are now replaced by the switches +S|-S and +L|-L. +L lists the symbol table, if any, and +S lists the source file. Specifying the minus option suppresses the listing. If no switch option is specified, the Assembler selects the defaults +S and +L unless you specify the OFF options with the directives SYMBOL and LIST in your source code.

DISASM

The disassembly command, DISASM, has been deleted.

DUMPOBJ

A new command, DUMPOBJ, has been added to allow you to view the contents of an object module format file. For a complete description of DUMPOBJ, refer to Chapter 4, "The Shell", of the *Cortland Programmer's Workshop* .

LINK

Link now allows you to link several files with a single command. To do this, enclose the multiple input filenames in parentheses. Like the initial filename, they should appear as the first parameter, excepting switches. Filenames can be separated by spaces, tabs or commas. The .ROOT file must exist for the first file. It is optional for all others.

MACGEN

MACGEN now accepts all parameters on the command line, although it can prompt for them if they are not provided. The first parameter is the name of the assembly language file to scan. Next comes the name of the macro file to create. Finally, one or more macro files to search can be specified. The old switches and wildcard options still apply.

In addition, MACGEN now works differently. Instead of opening the output file right away, MACGEN opens a file called SYSMAC on the work prefix, and writes the macros there. After all macros have been resolved, this work file is copied to the correct destination and the work file is deleted. This allows an old file to be scanned for macros first. For example, if a program called MYPROG already has a macro file called

MYPROG.MACROS, but one new macro is needed from the file LIB.MACROS, you can now do the macro file generation like this:

```
    MACGEN    MYPROG  MYPROG.MACROS MYPROG.MACROS LIB.MACROS
```

MAKELIB

MAKELIB is a new utility which has been added to make searching libraries faster.

This section is to be supplied; the implementation and syntax of the MAKELIB command is changing as of the Byte Works review

Object Module Format

The Cortland Workshop Assembler generates files that conform to the object module format.

The Assembler Listing

Machine code generated by a source line in ORCA/M assembly language appeared as an unbroken string of characters. In Cortland Workshop assembly language, a space is inserted between each byte of code.

Memory Management

The Cortland Workshop Assembler allocates 4K each for the global symbol table, local symbol table, AINPUT string buffer, and symbolic parameter table. Macro buffers and source file buffers are allocated from remaining memory, with each buffer being as large as the file. Tables are extensible in blocks of 4K and are extended, as needed, for as long as memory is available.

Note that there must be sufficient physical memory to load a given source or macro file. Compare this to the ORCA/M Assembler, which uses virtual memory techniques to allow files as large as a disk would hold. The change speeds up processing considerable.

Appendix D

The ASCII Character Set

CharDec	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**

This page is left intentionally blank

Appendix E

Error Messages

When the Assembler assembles a program, it finds and identifies errors in the source program. These errors fall into two broad categories: those that are recoverable and those that are terminal.

Recoverable 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 severity level follows the error message.

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. With each error message description, every possible cause for an error is explained, and ways to correct the problem are outlined.

An error message's severity level tells you how severe an error is. The Assembler can report four possible error levels: these are described in the table below. After each assembly containing errors, the Assembler prints out the highest error level found.

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 - no correction is possible, but the Assembler knew how much space to leave. You can use the Debugger can be to fix the problem without reassembly.
16	Error - it was not even possible to tell how much space to leave. You will need to reassemble to fix 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 adressing 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 progam 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 these cases, the Assembler assumes a length for the instruction which is then overridden before pass 2 starts.

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      &P,&P2
MEND
```

The call

```
EXAMPLE       A,P1=B
```

would produce this error, since P1 is set to A as a positional parameter, then to B as a keyword parameter.

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 parenthesis in an expression, but generally the Assembler will handle as many terms as will fit on a line, and about five or six levels of parenthesis. Check for any kind of syntax error in the expression itself.

Invalid Operand [8]

An operand was used on an instruction that does not support the addressing mode.

Label Syntax [16]

1. A symbolic parameter was expected in the label field, but one was not found. Symbolic parameters must begin with the & character, and are followed by an alphabetic character and one or more alphanumeric characters. Directives which require a symbolic parameter in the label field are:

   SETA
   SETB
   SETC
   AMID
   ASEARCH
   AINPUT

2. A directive that requires a label was used without one. The directives which must have a label in the label field are:

   START
   DATA

3. The label field of a statement contained a string which does not conform to the standard label syntax. The label must begin in column 1, and cannot contain imbedded blanks. Each label starts with an alphabetic character (A to Z) or underscore (_), and is followed by zero or more characters which can be can be alphabetic (A through Z), numeric (0 through 9), the tilde character (~), or the underscore character (_). Labels are significant up to 255 characters in length. There must be at least one space between it and the op code.

4. A macro model statement had something in the label field, but it was not a symbolic parameter. 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 zero page when the linker resolves the references, force zero page addressing by preceding the expression with a character, such as

```
        LDA     (<LABEL),Y
```

If the expression is a constant expression, correct it so that it is 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 Operand Syntax Error [4]

The operand of the macro model statement contained something other than a sequence of undefined symbolic parameters separated by commas. The macro model statement is the line immediately following a MACRO directive. If it has an operand at all, the operand must consist of a list of symbolic parameters separated by commas, with no imbedded spaces.

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 was used outside a code segment which must appear inside a code segment. Only the following directives can be used outside a code segment:

AIF	AGO	ORG	GEQU
MERR	SETCOM	EJECT	ERR
GEN	MSB	LIST	SYMBOL
PRINTER	65C02	65816	LONGA
LONGI	IEEE	TRACE	EXPAND
ALIGN	TITLE	RENAME	KEEP
COPY	APPEND	MCOPY	MDROP
MLOAD			

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, and it must come before any code is generated.

3. The RENAME directive, which must appear outside a program segment, was used inside a program segment.

4. An ORG directive with a constant operand was used inside a program segment, or an ORG that was not a displacement off of the location counter was used outside a program segment, or two ORGs were used before 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 END [23]

A START or DATA directive was encountered before the previous code segment was ended with an END directive. Each code segment must end with an END directive.

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 about 1E-38 to 1E38, while double precision numbers are limited to about 1E-308 to 1E308. If this error occurs, the Assembler will insert the IEEE format representation for 0 on an underflow, and infinity for an overflow.

2. A decimal number was found in the operand field which was not in the range minus 2147483647 to plus 2147483647. Since all integers are represented as four-byte signed numbers, decimal numbers must be in the above range.

3. A binary, octal or hexadecimal constant was found which requires more than 32 bits to represent. All numbers must be represented by no more than 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, read the section of the reference manual that deals with operand formats for the specific instruction or directive in question.

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 aligned itself, or was not aligned to a byte value greater than or equal to the ALIGN directive used in the progam 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 one-byte signed displacement from the end of the instruction, giving a range of minus 32765 to 32770 from the beginning of the instruction.

Sequence Symbol Not Found [4]

An AIF or AGO directive attempted to branch, but could not find the sequence symbol named in the operand field. A sequence symbol serves as the destination for a conditional assembly branch. It consists of a period in column one, followed by the sequence symbol name 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. Symbolic parameters come in one of three types: A (arithmetic), B (boolean), and C (character). All symbolic parameters defined in the parameter list of a macro call are 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. A subscript of 0 will also cause this errror.

Too Many MACRO Libs [2]

An MCOPY or MLOAD directive was encountered, and four macro libraries were already in use. The best solution is to combine all the macros needed 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 the MDROP directive to get rid of macro libraries that are no longer needed.

Too Many Positional Parameters [4]

The macro call statement used more parameters in the operand that 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 S attribute was requested for an undefined operation code, or for an operation code that does not use ON or OFF as its operand. The S attribute is only defined for these directives:

ERR	PRINTER	EXPAND
MSB	65C02	IEEE
GEN	65816	TRACE
LIST	LONGA	CASE
SYMBOL	LONGI	OBJCASE

Unidentified Operation [16]

1. An operation code was encountered which was not a valid insruction 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.

Undefined Symbolic Parameter [8]

An & 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.

Unresolved Label Not Allowed [2]

1. The operand of a directive 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 the final location that will be assigned by the Link Editor.

2. The length or type attribute of an undefined symbolic parameter was requested. Only the count attribute is allowed for an undefined symbolic parameter.

Terminal Errors

Some errors are so severe that the Assembler cannot keep going; these are called terminal errors. When the Assembler encounters a terminal 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 Editor, which loads the file that the Assembler was working on and places the line that caused the terminal error at the top of the display screen.

File Could Not Be Opened

A ProDOS error occurrred during an attempt to open a source or macro file.

This is generally 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 file name 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 occurrrs 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. In other words, the problem is 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 file name used in the KEEP directive for errors. This error will occur if the file name of the keep file exceeds ten characters, since the assembler must be able to append ".ROOT" to the keep file name, and ProDOS restricts file names to fifteen characters.

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 twelve bytes. Since 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 seventeen 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 zero 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. Using shorter subroutines will 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.

This page is left intentionally blank

Bibliography

The sources for the alpha draft of the *Cortland Workshop Assembler Reference* include:

- *ORCA/M User's Manual;* The Byte Works, 1984
- *Cortland Development Core and ORCA/M Macro Assembler* (ERS) Version 00.5, 25 March 1986.
- David Eyes and Ron Lichty, *Programming the 65816*, Simon and Schuster, 1986
- Michael Fisher, *65816/65802 Assembly Language Programming*, Osborne McGraw-Hill, 1896
- William Labiak, *Programming the 65816*, Sybex, 1986

This page is left intentionally blank

GLOSSARY

absolute segment: A segment that can be loaded only at one specific location in memory. Compare with **relocatable segment.**

accumulator: The register in the 65816 microprocessor where most computations are performed.

addressing mode:

AND: A logical operator that produces a true result if both its operands are true, and a false result if either or both its operands are false. Compare **OR, exclusive OR, NOT.**

argument:

arithmetic operator

array: An ordered collection of information of a given, defined type. Each element of the array can be referred to by a numerical subscript.

assembly: The process of translating source code into object code.

atttribute: returns information about a label or symbolic parameter.

base address: In indexed addressing, the fixed component of an address.

boolean operator: An operator, such as AND, that combines logical values to produce a logical result, such as true or false; named for English mathematician and logician George Boole. Also known as a **logical operator.** Compare **arithmetic operator, relational operator.**

binary file format: The ProDOS loadable file format, consisting of one absolute memory image along with its destination address.

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.**

case sensitivity: Ability to distinguish between uppercase characters and lowercase characters. Programming languages are *case sensitive* if they require all-uppercase letters, all-lowercase letters, or proper use of uppercase and lowercase. With the Cortland Assembler, you can use the directives CASE and OBJCASE to set case sensitivity.

code segment: A segment that contains program code as compared with data or variable definitions. See data segment.

command file: A program that runs other programs. See **EXEC file.**

command interpreter:

comment: Source text intented for the user, ignored during assembly.

comment line: The area of a source text line reserved for comments.

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: The process of programming the occurrence and characteristics of macro expansions.

conditional branch: A **branch** whose execution depends on the truth of a condition or the value of an expression. Compare **unconditional branch.**

constant: In a program, a symbol that represents a fixed, unchanging value. Compare **variable.**

cortland toolbox:

cross reference table:

data segment: A segment that contains program data.

debug: A colloquial term that means to locate and correct an error or the cause of a problem or malfunction in a computer program.

debugger:

delimiter: A character that is used for punctuation to mark the beginning or end of a sequence of characters, and which therefore is not considered part of the sequence itself. For example, Cortland assembly language uses the single quotation mark (') as a delimiter for string constants: the string 'DOG' consists of the three characters *D, O,* and *G,* and does not include the quotation marks.

denormalized number: A number represented in **floating-point** format, in which the first bit of the significand is zero. Compare **normalized number.**

destination: An address into which an instruction places data.

dictionary segment:

dimension: The maximum size of one of the subscripts of an array.

direct page:

directive: A source text instruction to the Assembler.

dot operator: Used to concatenate symbolic parameters, indicated by a period (.).

dynamic segment: A segment that can be loaded and unloaded during execution as needed. Compare with **static segment.**

editor:

effective address: In assembly-language programming, the address of the memory location on which a particular instruction operates, which may be arrived at by indexed addressing or some other addressing method.

element: A member of a set or collection; specifically, one of the individual variables making up an array.

embedded: Contained within. For example, the string 'BE HERE NOW' is said to contain two embedded spaces.

emulation mode:

equate:

error severity code:

error message: A message displayed or printed to tell you of an error or problem in your assembly. Error messages are accompanied by a **severity level code.**

exclusive OR: A logical operator that produces a true result if one of its operands is true and the other false, and a false result if its operands are both true or both false. Compare **OR, AND,** and **NOT.**

EXEC file:

expression:

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.** Compare **floating-point.**

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. Compare **fixed-point.**

global label:

global symbol table:

high-order byte:

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 65816 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.

keyword parameter:

label:

label scope:

library file: A file containing program segments, each of which can be used in any number of programs. The Linker can search through the library file for segments that have been referenced in the program source file.

library segment dictionary: An optional first segment of a libary file that contains the names and locations of all the segments in the file. The Linker uses the library segment dictionary to find segments in library files as a faster alternative to searching through the entire file.

linker: The program that processes object files generated by compilers and assemblers to produce load files. The Linker can combine several object files into one load file, combine several object segments into one load segment, and search libraries. It resolves all symbolic references, and generates a file that can be loaded into memory and executed.

link editor:

loader: See System Loader

load file: The output of the Linker. Load files can be loaded into memory and executed without further processing.

load segment: A segment in a load file. Load segments contain memory images that the system loader can load directly into memory, followed by a relocation dictionary that provides relocation to the system loader. Load segments can be static code, static data, dynamic code, or dynamic data.

label: A name that identifies a place in a source text. See also **local label, global label, label scope, attributes**

label scope:

local label:

local symbol table: A symbol table that contains symbols with local scope.

logical operator: An operator, such as AND, that combines logical values to produce a logical result, such as true or false; sometimes called a *Boolean operator*. Compare **arithmetic operator, relational operator.**

loop: A section of a program that is executed repeatedly until a limit or condition is met, such as an index variable's reaching a specified ending value.

low-order byte:

macro:

macro call: a request to execute a subroutine or macro.

macro definition:

macro expansion:

macro header: In a macro definition, the directive MACRO.

macro library:

memory manager: The part of the operating system 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.

mnemonic: A sequence of characters that designate an instruction or directive.

model statement:

native mode:

NOT: A unary logical operator that produces a true result if its operand is false, and a false result if its operand is true. Compare **AND, OR, exclusive OR.**

null: An undefined value. *Null* is different from zero; zero is a value just like other numbers, whereas *null* means no value at all (of the expected type). A *null string* does not contain anything. For example, ' ' is not a null string because it contains a space character; '' represents a null string.

object file: The output from the Assembler and input to the Linker. An object file conforms to the Cortland 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: The general format used in object files, library files, and load files.

object segment: A segment in an object file.

OMF file: Any file in object module format.

opcode: See **operation code.**

operand: A value to which an operator is applied. The value on which an operation code operates. Compare **argument.**

operation code: The part of a machine-language instruction that specifies the operation to be performed. Often called *opcode.*

operator: A symbol or sequence of characters, such as + or AND, specifying an operation to be performed on one or more values (the operands) to produce a result. See **arithmetic operator, relational operator, logical operator, unary operator, binary operator.**

OR: A logical operator that produces a true result if either or both of its operands are true, and a false result if both of its operands are false. Compare **exclusive OR, AND, NOT.**

page:

parameter:

partial assembly:

pass one:

pass two:

pathname: The full name by which an operating system identifies a file. A pathname is a sequence of filenames, each preceded by a slash, that specifies the path—from volume directory to file—the operating system takes to locate that file. Compare **filename.**

pop: To remove the top entry from a **stack,** moving the stack pointer to the entry below it. Synonymous with *pull.* Compare **push.**

positional parameter:

precedence: The order in which operators are applied in evaluating an expression. Precedence varies from language to language, but usually resembles the precedence rules of algebra.

ProDOS 16:

program counter:

program segment:

push: To add an entry to the top of a **stack,** moving the stack pointer to point to it. Compare **pop.**

quickDraw:

real number: In computer usage, a number that may include a fractional part; represented inside the computer in **floating-point** form. Because a real number is of infinite precision, this representation is usually approximate. Compare **integer.**

relational operator: An operator, such as >, that operates on numeric values to produce a logical result. Compare **arithmetic operator, logical operator.**

relocate: The process of modifying a file or segment at load time so that it will execute correctly at the location in memory at which it is loaded. The Linker resolves symbolic references and prepares relocation dictionaries that the Loader uses to relocate a program after loading it into memory. See also **relocatable segment.**

relocation dictionary: A portion of a load segment that contains relocation information necessary to modify the memory image immediately preceding it. When the segment is loaded into memory, the relocation dictionary is used to patch location-dependent addresses into the code.

return address: The point in a program to which control returns on completion of a subroutine or function.

SANE: See **Standard Apple Numeric Environment.**

scope:

segment: An individual component of an OMF file. Each file contains one or more segments. The header of an object segment has both an object-segment name and a load-segment name. The Linker normally places all object segments that have the same load-segment name into a single load segment. You can also use a LinkEd file to assign specific object segments to specific load segments at link time. Load segments can be static code, static data, dynamic code, or dynamic data. The object module format also defines several special segment types to support the loader, library files and so forth.

segment jump table: A segment in a load file, created by the Linker, that provides the information the loader needs to locate dynamic segments as they are needed during program execution.

sequence symbol: The destination of a conditional assembly branch, indicated by a period (.) followed by a label.

set symbol:

shell:

source program:

stack: A list in which entries are added (pushed) or removed (popped) at one end only (the top of the stack), causing them to be removed in last-in, first-out (LIFO) order.

step value: The amount by which the index variable changes on each pass through a loop.

standard apple numeric environment:

static segment: A static segment is loaded at program boot time, and is not unloaded or moved during execution. Compare with **dynamic segment.**

string: An item of information consisting of a sequence of text characters.

subdirectory: A directory within a directory. A file containing the names and locations of other files.

subscript: A numeric expression whose value is the index of an element in a sublist or array.

substring: A string that is part of another string.

symbol table:

symbolic parameter:

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.

unconditional branch: A branch that does not depend on the truth of any condition. Compare **conditional branch.**

variable: The symbol used in a program to represent a memory location where a value can be stored. Compare **constant.**

wildcard character: The asterisk character (*) that can be used as shorthand to represent a sequence of characters in a pathname. For example, if you request a listing of *.TEXT files in a particular application, you would see a list of all files ending with the suffix .TEXT.

X register: One of the two index registers in the 65816 microprocessor.

Y register: One of the two index registers in the 65816 microprocessor.

This page is left intentionally blank

Index

To be supplied.

This page is left intentionally blank

ANY COMMENTS?

We would like to know your feelings about this manual.

- What did you like or dislike about it?
- Were you able to find all the information you need?
- Was the information complete and accurate?
- Was it organized in a helpful way?
- Do you have any suggestions for improvement?

Please send your comments and suggestions to

J. Chapman, Technical Publications
Apple Computer, Inc.
20525 Mariani Avenue, MS 22K
Cupertino, CA 95014

You may write you comments separately or on a marked-up copy of the manual (we'll return your marked-up copy if you like).

Thanks for your help!

Quick Reference Guide for Cortland Workshop Assembler

Pre-Alpha Draft
June 20, 1986
Engineering Part Number: 030-3131
Marketing Part Number: A2L6001

Writer: Catherine Williamson
Apple User Education Department

Revision History

Document Design, *Quick Reference Guide for Cortland Workshop Assembler* , Catherine Williamson, March 17, 1986.

Pre-Alpha Draft, *Quick Reference Guide for Cortland Workshop Assembler* , Catherine Williamson, June 20, 1986.

About This Pocket Guide

The purpose of the *Quick Reference Guide to the Cortland Workshop Assembler* is to give Assembly Language programmers a handy summary of the information they are most likely to use most often. The sections in this Guide include:

- Using The Assembler
- 65816 Instruction Set
- 65816 Instruction List
- General Assembler Directives
- Macro Directives
- ASCII Character Set

The Cortland Tools will be summarized in the *Cortland Tools: Parts I and II*.*

Are there any other topics people would like to see covered here?

Using The Assembler

ASM65816	Change the default language to 65816 assembly language
EDIT	Edit an existing file
NEW	Open a new edit window
ASSEMBLE	Assembly a program
ASML	Assemble and link a program
ASMLG	Assemble, link and execute a program
LINK	Link an object module
ALINK	Process an Advanced Linker file
MACGEN	Build a macro Library
DEBUG	Debug a File
XREF	Create a cross reference table
MAKELIB	Make a dictionary segment

ASM65816

EDIT *filename*

NEW *filename*

ASSEMBLE [+L|-L] [+S|-S] *sourcefile* [KEEP=*outfile*][NAMES=(*seg1*[,*seg2*[,...]])]
[*language1*=(*option*...)[*language2*=(*option*...)...]]

ASML [+L|-L] [+S|-S] *sourcefile* [KEEP=*outfile*][NAMES=(*seg1*[,*seg2*[,...]])]
[*language1*=(*option*...)[*language2*=(*option*...)...]]

ASMLG [+L|-L] [+S|-S] *sourcefile* [KEEP=*outfile*][NAMES=(*seg1*[,*seg2*[,...]])]
[*language1*=(*option*...)[*language2*=(*option*...)...]]

LINK [+L|-L] [+S|-S] *sourcefile* [KEEP=*outfile*][NAMES=(*seg1*[,*seg2*[,...]])]
[*language1*=(*option*...)[*language2*=(*option*...)...]]

ALINK [+L|-L] [+S|-S] *sourcefile* [KEEP=*outfile*]

Assembler Command Options

+L|-L	+L generates a source listing. Defaults to +L.
+S|-S	+S produces a symbol table. Defaults to +S.
sourcefile	The full pathname and filename of the source file.
KEEP=*outfile*	The name of the output file.

NAMES=(*seg1,seg2,...*)
Specifies the names of the segments to be assembled.

[*language1=option...*)[*language2=option...*)...]]
Passes parameters directly to the Cortland compilers.

MACGEN [+C] [-C] infile outfile macrofile1 [macrofile2...]

+C|-C	+C removes all excess blanks from a macro file. +C is the default.
infile	The full pathname and filename of the source file.
outfile	The full pathname and filename of the macro file to be created by MACGEN.

macrofile1 [macrofile2...]
The full pathnames and filenames of the macro libraries to be searched for the macros referenced in *infile.*

DEBUG *filename*

XREF [+L\|-L] [+X\|-X] [+F\|-F] [(subrange1,...,subrange5)]filename

+L| -L	+L lists the file. The default is +L.
+F|-F	+X lists a frequency count. The default is -F.
+X|-X	+X parameter generates a cross reference. The default is +X

(*subrange1,...subrange5*)
specify a subrange in a cross reference table

filename	The full pathname and filename of the source file to be processed.

MAKELIB *filename*

65816 Instruction Set Summary

Data Movement Instructions

Load/Store Instructions:

LDA	Load accumulator from memory
LDX	Load the X index register
LDY	Load the Y index register
STA	Store the accumulator
STX	Store the X index register
STY	Store the Y index register

Push Instructions:

PHA	Push the accumulator
PHP	Push status register (flags)
PHX	Push X index register
PHY	Push Y index register
PHB	Push data bank register
PHK	Push program bank register
PHD	Push direct page register
PEA	Push effective absolute address
PEI	Push effective indirect address
PER	Push effective relative address

Pull Instructions:

PLA	Pull the accumulator
PLP	Pull status register (flags)
PLX	Pull X index register
PLY	Pull Y index register
PLB	Pull data bank register
PLD	Pull direct page register

Transfer Instructions:

TAX	Transfer A to X
TAY	Transfer A to Y
TSX	Transfer S to X
TXS	Transfer X to S
TXA	Transfer X to A
TYA	Transfer Y to A
TCD	Transfer C accumulator to D
TDC	Transfer D to C accumulator
TCS	Transfer C accumulator to S
TSC	Transfer S to C accumulator
TXY	Transfer X to Y
TYX	Transfer Y to X

Exchange Instructions:

XBA	Exchange B and A accumulators
XCE	Exchange carry and emulation bits

Store Zero to Memory:

STZ	Store zero to memory

Block Moves:

MVN	Move block in negative direction
MVP	Move block in positive direction

Flow Of Control Instructions

BCC	Branch if carry clear
BCS	Branch if carry set
BEQ	Branch if equal
BMI	Branch if minus
BNE	Branch if not equal
BPL	Branch if plus
BRA	Branch always
BRL	Branch always long
BVC	Branch if overflow clear
BVS	Branch if overflow set
JMP	Jump
JSR	Jump to subroutine
JSL	Jump to subroutine long
RTS	Return from subroutine
RTL	Return from subroutine long

Arithmetic Instructions

DEC	Decrement
DEX	Decrement index register X
DEY	Decrement index register Y
INC	Increment
INX	Increment Index register X
INY	Increment index register Y

Logic And Bit Manipulation Instructions

Logic Instructions:

AND	Logical AND
EOR	Logical exclusive-OR
ORA	Logical OR (inclusive OR)

Bit Manipulation Instructions:

BIT	Test bits
TRB	Test and reset bits
TSB	Test and set bits

Shift and Rotate Instructions:

ASL	Shift bits left
LSR	Shift bits left
ROL	Rotate bits left
ROR	Rotate bits right

System Control Instructions

BRK	Break (software interrupt)
RTI	Return from interrupt
NOP	No operation
SEC	Set carry flag
CLC	Clear carry flag
SED	Set decimal mode
CLD	Clear decimal mode
SEI	Set interrupt disable flag
CLI	Clear overflow flag
CLV	Clear overflow flag
SEP	Set status register bits
REP	Clear status register bits
COP	Co-processor of software interrupt
STP	Stop the clock
WAI	Wait for interrupt
WDM	Reserved for expansion

General Assembler Directives

Directive	Action
Program Control Directives	
START	Start subroutine
PRIVATE	Define private code segment
DATA	Define data segment
PRIVDATA	Define private data segment
USING	Using data segment
ENTRY	Define entry point
END	End program segment
Data Definition Directives	
DC	Declare constant
DS	Declare storage
Symbol Definition Directives	
EQU	Equate
GEQU	Global equate
RENAME	Rename opcodes
Memory Designation Directives	
ALIGN	Align to a boundary
ORG	Designate origin
MEM	Reserve memory
File Control Directives	
APPEND	Append a file
COPY	Copy a file
KEEP	Keep object module
Assembler Option Directives	
IEEE	Generate IEEE format numbers
LONGA	Select accumulator size
LONGI	Select index register size
MSB	Set the most significant bit of characters
65C502	Enable 65C02 code

65816	Enable 65816 code
MERR	Set the maximum error level
CASE	Specify case-sensitivity
OBJCASE	Specify case-sensitivity in object module

Listing Option Directives

ERR	Print errors
EXPAND	Expand DC statements
LIST	List output
PRINTER	Send output to printer
SYMBOL	Print symbol tables
EJECT	Eject the page
SETCOM	Set comment column
TITLE	Print header
ABSADDR	Allow absolute addresses
INSTIME	Show instruction times

Macro Directives

Directive Action

Macro Language Directives

MACRO	Start a macro definition
MNOTE	Macro note
MEXIT	Exit macro
MEND	End a macro definition

Macro Library Directives

MCOPY	Copy Macro Library
MDROP	Drop A Macro Library
MLOAD	Load A Macro Library

Listing Directives

GEN	Generate macro expansions
TRACE	Trace macros

Conditional Assembly Directives

Defining Parameters

LCLA	Define a local arithmetic symbolic parameter
LCLB	Define a local boolean symbolic parameter
LCLC	Define a local string symbolic parameter
GLBA	Define a global arithmetic symbolic parameter
GLBB	Define a global boolean symbolic parameter
GLBC	Define a global string symbolic parameter
SETA	Assign a value to an arithmetic symbolic parameter
SETB	Assign a value to a boolean string symbolic parameter
SETC	Assign a value to a string symbolic parameter

String Manipulation Directives

ASEARCH	Assembler String Search
AMID	Assembler Mid String

Defining Parameters Using Assembler Input

AINPUT	Assembler Input

Branching Directives

AGO	Unconditional Branch
AIF	Conditional Branch

Miscellaneous Directives

ACTR	Assembly Counter
ANOP	Assembler No Operation

Instruction Set Summary

Opcode Hex	Mnemonic	Adressing Mode	# Of Bytes	# Of Cycles
00	BRK	Stack/Interrupt	2 **	7[9]
01	ORA	DP Indexed Indirect,X	2	6[1,2]
02	COP	Stack/Interrupt	2 **	7[9]
03	ORA	Stack Relative (also SR)	2 **	4[1]
04	TSB	Direct Page	2	5[2,5]
05	ORA	Direct Page (also DP)	2	3[1,2]
06	ASL	Direct Page (DP)	2	5[2,5]
07	ORA	DP Indirect Long	2	6[1,2]
08	PHP	Stack (Push)	1	3
09	ORA	Immediate	2 *	2[1]
0A	ASL	Accumulator	1	2
0B	PHD	Stack (Push)	1	4
0C	TSB	Absolute	3	6[5]
0D	ORA	Absolute	3	4[1]
0E	ASL	Absolute	3	6[5]
0F	ORA	Absolute Long	4	5[1]
10	BPL	Program Counter Relative	2	2[7,8]
11	ORA	DP Indirect Indexed, Y	2	5[1,2,3]
12	ORA	DP Indirect	2	5[1,2]
13	ORA	SR Indirect Indexed, Y	2	7[1]
14	TRB	Direct Page	2	5[2,5]
15	ORA	DP Indexed, X	2	4[1,2]
16	ASL	DP Indexed, X	2	6[2,5]
17	ORA	DP Indirect Long, Indexed	2	6[1,2]
18	CLC	Implied	1	2
19	ORA	Absolute Indexed, Y	3	4[1,3]
1A	INC	Accumulator	1	2

1B	TCS	Implied	1	2
1C	TRB	Absolute	3	6[5]
1D	ORA	Absolute Indexed, X	3	4[1,3]
1E	ASL	Absolute Indexed, X	3	7[5,6]
1F	ORA	Absolute Long Indexed, X	4	5[1]
20	JSR	Absolute	3	6
21	AND	DP Indexed Indirect, X	2	6[1,2]
22	JSR	Absolute Long	4	8
23	AND	Stack Relative (SR)	2	4[1]
24	BIT	Direct Page (DP)	2	3[1,2]
25	AND	Direct Page (DP)	2	3[1,2]
26	ROL	Direct Page (also DP)	2	5[2,5]
27	AND	DP Indirect Long	2	6[1,2]
28	PLP	Stack (Pull)	1	4
29	AND	Immediate	2*	2[1]
2A	ROL	Accumulator	1	2
2B	PLD	Stack (Pull)	1	5
2C	BIT	Absolute	3	4[1]
2D	AND	Absolute	3	4[1]
2E	ROL	Absolute	3	6[5]
2F	AND	Absolute Long	4	5[1]
30	BMI	Program Counter Relative	2	2[7,8]
31	AND	DP Indirect Indexed, Y	2	5[1,2,3]
32	AND	DP Indirect	2	5[1,2]
33	AND	SR Indirect Indexed, Y	2	7[1]
34	BIT	DP Indexed, X	2	4[1,2]
35	AND	DP Indexed, X	2	4[1,2]
36	ROL	DP Indexed, X	2	6[2,5]
37	AND	DP Indirect Long Indexed, Y	2	6[1,2]
38	SEC	Implied	1	2
39	AND	Absolute Indexed, Y	3	4[1,3]
3A	DEC	Accumulator	1	2

3B	TSC	Implied	1	2
3D	AND	Absolute Indexed, X	3	4[1,3]
3E	ROL	Absolute Indexed, X	3	7[5,6]
3F	AND	Absolute Long Indexed, X	4	5[1]
40	RTI	Stack /RTI	1	6[9]
41	EOR	DP Indexed Indirect, X	2	6[1,2]
*42	WDM	*reserved*	2	2[1,6]
43	EOR	Stack Relative (also SR)	2	4[1]
*44	MVP	Block Move	3	13
45	EOR	Direct Page (also DP)	2	3[1,2]
46	LSR	Direct Page (also DP)	2	5[2,5]
47	EOR	DP Indirect Long	2	6[1,2]
48	PHA	Stack (Push)	1	3[1]
49	EOR	Immediate	2*	2[1]
4A	LSR	Accumulator	1	2
4B	PHK	Stack (Push)	1	3
4C	JMP	Absolute	3	3
4D	EOR	Absolute	3	4[1]
4E	LSR	Absolute	3	6[5]
4F	EOR	Absolute Long	4	5[1]
50	BVC	Program Counter Relative	2	2[7,8]
51	EOR	DP Indirect Indexed, Y	2	5[1,2,3]
52	EOR	DP Indirect	2	5[1,2]
53	EOR	SR Indirect Indexed, Y	2	7[1]
*54	MVN	Block Move	3	13
55	EOR	DP Indexed, X	2	4[1,2]
56	LSR	DP Indexed, X	2	6[2,5]
57	EOR	DP Indirect Long Indexed Y	2	6[1,2]
58	CLI	Implied	1	2
59	EOR	Absolute Indexed, Y	3	4[1,3]
5A	PHY	Stack (Push)	1	3[10]
5B	TCD	Implied	1	2

5C	JMP	Absolute Long	4	4
5D	EOR	Absolute Indexed, X	3	$4^{1,3}$
5E	LSR	Absolute Indexed, X	3	$7^{5,6}$
5F	EOR	Absolute Long Indexed, X	4	5^{1}
60	RTS	Stack (RTS)	1	6
61	ADC	DP Indexed Indirect, X	2	$6^{1,2,4}$
62	PER	Stack (PC Relative Long)	3	6
63	ADC	Stack Relative (SR)	2	$4^{1,4}$
64	STZ	Direct Page	2	$3^{1,2}$
65	ADC	Direct Page (DP)	2	$3^{1,2,4}$
66	ROR	Direct Page (also DP)	2	$5^{2,5}$
67	ADC	DP Indirect Long	2	$6^{1,2,4}$
68	PLA	Stack (Pull)	1	4^{1}
69	ADC	Immediate	2*	$2^{1,4}$
6A	ROR	Accumulator	1	2
6B	RTL	Stack (RTL)	1	6
6C	JMP	Absolute Indirect	3	$5^{11,12}$
6D	ADC	Absolute	3	$4^{1,4}$
6E	ROR	Absolute	3	6^{5}
6F	ADC	Absolute Long	4	$5^{1,4}$
70	BVS	Program Counter Relative	2	$2^{7,8}$
71	ADC	DP Indirect Indexed, Y	2	$5^{1,2,3,4}$
72	ADC	DP Indirect	2	$5^{1,2,4}$
73	ADC	SR Indirect Indexed, Y	2	$7^{1,4}$
74	STZ	Direct Page Indexed, X	2	$4^{1,2}$
75	ADC	DP Indexed, X	2	$4^{1,2,4}$
76	ROR	DP Indexed, X	2	$6^{2,5}$
77	ADC	DP Indirect Long Indexed, Y	2	$6^{1,2,4}$
78	SEI	Implied	1	2
79	ADC	Absolute Indexed, Y	3	$4^{1,3,4}$
7A	PLY	Stack/Pull	1	4^{10}
7B	TDC	Implied	1	2

7C	JMP	Absolute Indexed Indirect	3	6
7D	ADC	Absolute Indexed, X	3	4[1,3,4]
7E	ROR	Absolute Indexed, X	3	7[5,6]
7F	ADC	Absolute Long Indexed, X	4	5[1,4]
80	BRA	Program Counter Relative	2	3[6]
81	STA	DP Indexed Indirect, X	2	6[1,2]
82	BRL	Program Counter Relative Long	3	4
83	STA	Stack Relative (also SR)	2	4[1]
84	STY	Direct Page	2	3[2,10]
85	STA	Direct Page (also DP)	2	3[1,2]
86	STX	Direct Page	2	3[2,10]
87	STA	DP Indirect Long	2	6[1,2]
88	DEY	Implied	1	2
89	BIT	Immediate	2*	2[1]
8A	TXA	Implied	1	2
8B	PHB	Stack (Push)	1	3
8C	STY	Absolute	3	4[10]
8D	STA	Absolute	3	4[1]
8E	STX	Absolute	3	4[10]
8F	STA	Absolute Long	4	5[1]
90	BCC	Program Counter Relative	2	2[7,8]
91	STA	DP Indirect Indexed, Y	2	6[1,2]
92	STA	DP Indirect	2	5[1,2]
93	STA	SR Indirect Indexed, Y	2	7[1]
94	STY	Direct Page Indexed, X	2	4[2,10]
95	STA	DP Indexed, X	2	4[1,2]
96	STX	Direct Page Indexed, Y	2	4[2,10]
97	STA	DP Indirect Long Indexed, Y	2	6[1,2]
98	TYA	Implied	1	2
99	STA	Absolute Indexed, Y	3	5[1]
9A	TXS	Implied	1	2
9B	TXY	Implied	1	2

9C	STZ	Absolute	3	4^1
9D	STA	Absolute Indexed, X	3	5^1
9E	STZ	Absolute Indexed, X	3	5^1
9F	STA	Absolute Long Indexed, X	4	5^1
A0	LDY	Immediate	2+	2^{10}
A1	LDA	DP Indexed Indirect, X	2	$6^{1,2}$
A2	LDX	Immediate	2+	2^{10}
A3	LDA	Stack Relative (alsoSR)	2	4^1
A4	LDY	Direct Page (also DP)	2	$3^{2,10}$
A5	LDA	Direct Page (also DP)	2	$3^{1,2}$
A6	LDX	Direct Page (also DP)	2	$3^{2,10}$
A7	LDA	DP Indirect Long	2	$6^{1,2}$
A8	TAY	Implied	1	2
A9	LDA	Immediate	2*	2^1
AA	TAX	Implied	1	2
AB	PLB	Stack (Pull)	1	4
AC	LDY	Absolute	3	4^{10}
AD	LDA	Absolute	3	4^1
AE	LDX	Absolute	3	4^{10}
AF	LDA	Absolute Long	4	5^1
B0	BCS	Program Counter Relative	2	$2^{7,8}$
B1	LDA	DP Indirect Indexed, Y	2	$5^{1,2,3}$
B2	LDA	DP Indirect	2	$5^{1,2}$
B3	DLA	SR Indirect Indexed, Y	2	7^1
B4	LDY	DP Indexed, X	2	$4^{2,10}$
B5	LDA	DP Indexed, X	2	$4^{1,2}$
B6	LDX	DP Indexed, Y	2	$4^{2,10}$
B7	LDA	DP Indirect Long Indexed, Y	2	$6^{1,2}$
B8	CLV	Implied	1	2
B9	LD	Absolute Indexed, Y	3	$4^{1,3}$
BA	TSX	Implied	1	2
BB	TYX	Implied	1	2

BC	LDY	Absolute Indexed, X	3	$4^{1,10}$
BD	LDA	Absolute Indexed, X	3	$4^{1,3}$
BE	LDX	Absolute Indexed, Y	3	$4^{3,10}$
BF	LDA	Absolute Long Indexed, X	4	5^{1}
C0	CPY	Immediate	2+	2^{10}
C1	CMP	DP Indexed Indirect, X	2	$6^{1,2}$
C2	REP	Immediate	2	3
C3	CMP	Stack Relative (also SR)	2	4^{1}
C4	CPY	Direct Page (also DP)	2	$3^{2,10}$
C5	CMP	Direct Page (also DP)	2	$3^{1,2}$
C6	DEC	Direct Page (also DP)	2	$5^{2,5}$
C7	CMP	DP Indirect Long	2	$6^{1,2}$
C8	INY	Implied	1	2
C9	CMP	Immediate	2*	2^{1}
CA	DEX	Implied	1	2
CB	WAI	Implied	1	3^{15}
CC	CPY	Absolute	3	4^{10}
CD	CMP	Absolute	3	4^{1}
CE	DEC	Absolute	3	6^{5}
CF	CMP	Absolute Long	4	5^{1}
D0	BNE	Program Counter Relative	2	$2^{7,8}$
D1	CMP	DP Indirect Indexed, Y	2	$5^{1,2,3}$
D2	CMP	DP Indirect	2	$5^{1,2}$
D3	CMP	SR Indirect Indexed, Y	2	7^{1}
D4	PEI	Stack (Direct Page Indirect)	2	6^{2}
D5	CMP	DP Indexed, X	2	$4^{1,2}$
D6	DEC	DP Indexed, X	2	$6^{2,5}$
D7	CMP	DP Indirect Long Indexed, Y	2	$6^{1,2}$
D8	STP	Implied	1	2
D9	CMP	Absolute Indexed, Y	3	$4^{1,3}$
DA	PHX	Stack (Push)	1	$3^{1,10}$

DB	STP	Implied	1	3[14]
DC	JMP	Absolute Indirect Long	3	6
DD	CMP	Absolute Indexed, X	3	4[1,3]
DE	DEC	Absolute Indexed, X	3	7[5,6]
DF	CMP	Absolute Long Indexed, X	4	5[1]
E0	CPX	Immediate	2+	2[10]
E1	SBC	DP Indexed Indirect, X	2	6[1,2,4]
E2	SEP	Immediate	2	3
E3	SBC	Stack Relative (also SR)	2	4[1,4]
E4	CPX	Direct Page (also DP)	2	3[2,10]
E5	SBC	Direct Page (also DP)	2	3[1,2,4]
E6	INC	Direct Page (also DP)	2	5[2,5]
E7	SBC	DP Indirect Long	2	6[1,2,4]
E8	INX	Implied	1	2
E9	SBC	Immediate	2*	2[1,4]
EA	NOP	Implied	1	2
EB	XBA	Implied	1	3
EC	CPX	Absolute	3	4[10]
ED	SBC	Absolute	3	4[1,4]
EE	INC	Absolute	3	6[5]
EF	SBC	Absolute Long	4	5[1,4]
F0	BEQ	Program Counter Relative	2	2[7,8]
F1	SBC	DP Indirect Indexed, Y	2	5[1,2,3,4]
F2	SBC	DP Indirect	2	5[1,2,4]
F3	SBC	SR Indirect Indexed, Y	2	7[1,4]
F4	PEA	Stack (Absolute)	3	5
F5	SBC	DP Indexed, X	2	4[1,2,4]
F6	INC	DP Indexed, X	2	6[2,5]
F7	SBC	DP Indirect Long Indexed, Y	2	6[1,2,4]
F8	SED	Implied	1	2
F9	SBC	Absolute Indexed, Y	3	4[1,3,4]
FA	PLX	Stack/Pull	1	4[10]

FB	XCE	Implied	1	2
FC	JSR	Absolute Indexed Indirect	3	8
FD	SBC	Absolute Indexed, X	3	4[1,3,4]
FE	INC	Absolute Indexed, X	3	7[5,6]
FF	SBC	Absolute Long Indexed, X	4	5[1,4]

* Add 1 byte if m=0 (16-bit memory/accumulator)

** Opcode is 1 byte, but program counter value pushed onto stack is incremented by two, allowing for optional signature byte

+ Add 1 byte if x=0 (16-bit index registers)

1 Add 1 cycle if m=0 (16-bit memory/accumulator)

2 Add 1 cycle if low byte of Direct Page register is other than zero (DL<>0)

3 Add 1 cycle if adding index crosses a page boundary

4 Add 1 cycle if 65C02 and d=1 (decimal mode, 65C02)

5 Add 2 cycles if m=0 (16-bit memory/accumulator)

6 Subtract 1 cycle if 65C02 and no page boundary crossed

7 Add 1 cycle if branch taken

8 Add 1 more cycle if branch taken crosses page boundary on 6502, 65C02, or 65816/65802's emulation mode (e=1)

9 Add 1 cycle for 65802/65816 native mode (e=0)

10 Add 1 cycle if x=0 (16-bit index registers)

11 Add 1 cycle if 65C02

12 6502: If low byte of addr is $FF (i.e., addr is $xxFF): yields incorrect result

13 7 cycles per byte moved

14 Uses 3 cycles to shut the processor down; additional cycles are required by reset to restart it

15 Uses 3 cycles to shut the processor down; additional cycles are required by interrupt to restart it

16 Byte and cycle counts subject to change in future processors which expand WDM into 2-byte opcode portions of instructions of varying lengths

The ASCII Character Set

CharDec	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