EMULOGIC CROSS ASSEMBLER REFERENCE MANUAL

FOR THE 6500 FAMILY

First Edition (June 1984)

No part of this publication may be reproduced by any means without prior written permission from Emulogic, Inc. Use of this document is restricted to customers of Emulogic, Inc., and its employees and agents.

The information contained herein is subject to change without notice. Emulogic, Inc., assumes no responsibility for any errors that may appear in this document.

Details and specifications concerning the use and operation of Emulogic equipment and software are included in various technical manuals available through local sales representatives.

Copyright c 1984 Emulogic, Inc. All rights reserved.

Copyright c 1984 Digital Equipment Corporation. All rights reserved. Permission was obtained from Digital Equipment Corporation to reproduce material used in this document.

EMULOGIC is a registered trademark of Emulogic, Inc.

The following are registered trademarks of Digital Equipment Corporation:

DEC	RSX	RT-11	MACRO 11
PDP	PDP-11	VAX	VMS

Printed in U.S.A.

PREFACE

The _Emulogic Cross Assembler Reference Manual_ contains detailed information on the following subjects:

- o Source program format
- o Symbols and expressions
- o The microprocessor instruction set
- o Cross Assembler directives
- o Macro directives

A companion document, the _Emulogic Cross Assembler User's Guide_, tells you how to use the Cross Assembler, Linker and Librarian.

Organization:

Chapter 1 lists the features of the Cross Assembler and describes the two-pass assembly process.

Chapter 2 describes the format to use in coding source programs.

Chapter 3 lists the character set and describes the symbols, terms and expressions that form the elements of the Cross Assembler instructions.

Chapter 4 is a brief introduction to relocation and linking. (For detailed information, see Chapter 3 of the _Emulogic Cross Assembler User's Guide_.)

Chapter 5 contains the microprocessor's instruction set.

Chapter 6 describes Cross Assembler directives.

Chapter 7 describes macro directives.

Appendix A lists ASCII characters and Radix-50 characters.

Appendix B lists the Cross Assembler's special characters and directives.

Appendix C contains error messages.

EMULOGIC CROSS ASSEMBLER REFERENCE MANUAL

FOR THE 6500 FAMILY

TABLE OF CONTENTS

PREFACE iii

CHAPTER 1 THE EMULOGIC CROSS ASSEMBLER 1-1

1.1 INTRODUCTION 1-1
1.2 ASSEMBLY PASS 1 1-2
1.3 ASSEMBLY PASS 2 1-3

CHAPTER 2 SOURCE PROGRAM FORMAT 2-1

2.1 PROGRAMMING STANDARDS AND CONVENTIONS 2-1
2.2 STATEMENT FORMAT 2-1
2.2.1 Label Field 2-2
2.2.2 Operator Field 2-4
2.2.3 Operand Field 2-4
2.2.4 Comment Field 2-5
2.3 FORMAT CONTROL 2-7

CHAPTER 3 SYMBOLS AND EXPRESSIONS 3-1

3.1 CHARACTER SET 3-1
3.1.1 Separating and Delimiting Characters 3-4
3.1.2 Illegal Characters 3-4
3.1.3 Unary and Binary Operators 3-5
3.2 CROSS-ASSEMBLER SYMBOLS 3-8
3.2.1 Permanent Symbols 3-8
3.2.2 User-Defined and Macro Symbols 3-8
3.3 DIRECT ASSIGNMENT STATEMENTS 3-11
3.4 LOCAL SYMBOLS 3-13
3.5 CURRENT LOCATION COUNTER 3-15
3.6 NUMBERS 3-18
3.7 TERMS 3-19
3.8 EXPRESSIONS 3-20

CHAPTER 4 RELOCATION AND LINKING 4-1

CHAPTER 5 THE 6500 FAMILY INSTRUCTION SET 5-1

5.1	INTRODUCTION	5-1
5.2	PROGRAMMING NOTES	5-2
5.3	REGISTERS	5-3
5.4	6500 FAMILY INSTRUCTION SET	5-4
5.5	SAMPLE 6500 ASSEMBLY LISTING	5-19
CHAPTER 6	GENERAL CROSS ASSEMBLER DIRECTIVES	6-1
6.1	LISTING CONTROL DIRECTIVES	6-4
6.1.1	.LIST and .NLIST Directives	6-4
6.1.2	.TITLE Directive	6-10
6.1.3	.SBTTL Directive	6-10
6.1.4	.IDENT Directive	6-12
6.1.5	.PAGE Directive/Page Ejection	6-13
6.1.6	.REM Directive/Begin Remark Lines	6-13
6.2	FUNCTION DIRECTIVES	6-14
6.2.1	.ENABL and .DSABL Directives	6-14
6.3	DATA STORAGE DIRECTIVES	6-18
6.3.1	.BYTE Directive	6-18
6.3.2	.WORD Directive	6-19
6.3.3	.LONG Directive	6-20
6.3.4	ASCII Conversion Characters	6-21
6.3.5	.ASCII Directive	6-22
6.3.6	.ASCIZ Directive	6-25
6.3.7	.RAD50 Directive	6-26
6.3.8	Temporary Radix-50 Control Operator	6-28
6.4	RADIX AND NUMERIC CONTROL FACILITIES	6-29
6.4.1	Radix Control and Unary Control Operators	6-29
6.4.1.1	.RADIX Directive	6-29
6.4.1.2	Temporary Radix Control Operators	6-30
6.5	LOCATION COUNTER CONTROL DIRECTIVES	6-32
6.5.1	.EVEN Directive	6-32
6.5.2	.ODD Directive	6-33
6.5.3	.BLKB, .BLKW and .BLKL Directives	6-33
6.6	TERMINATING DIRECTIVE: .END DIRECTIVE	6-35
6.7	PROGRAM SECTIONING DIRECTIVES	6-36
6.7.1	.PSECT Directive	6-36
6.7.2	.ASECT Directive	6-41
6.8	SYMBOL CONTROL DIRECTIVES	6-42
6.8.1	.GLOBL Directive	6-42
6.9	CONDITIONAL ASSEMBLY DIRECTIVES	6-44
6.9.1	Conditional Assembly Block Directives	6-44
6.9.2	Subconditional Assembly Block Directives	6-48
6.9.3	Immediate Conditional Assembly Directives	6-52

CHAPTER 7 MACRO DIRECTIVES 7-1

7.1 DEFINING MACROS 7-1
7.1.1 .MACRO Directive 7-1
7.1.2 .ENDM Directive 7-2
7.1.3 .MEXIT Directive 7-3
7.1.4 MACRO Definition Formatting 7-4
7.2 CALLING MACROS 7-5
7.3 ARGUMENTS IN MACRO DEFINITIONS AND MACRO CALLS 7-6
7.3.1 Macro Nesting 7-7
7.3.2 Passing Numeric Arguments as Symbols 7-8
7.3.3 Number of Arguments in Macro Calls ...7-10
7.3.4 Creating Local Symbols Automatically 7-11
7.3.5 Keyword Arguments 7-12
7.3.6 Concatenation of Macro Calls 7-13
7.4 MACRO ATTRIBUTE DIRECTIVES: .NARG, .NCHR and .NTYPE 7-15
7.4.1 .NARG Directive 7-15
7.4.2 .NCHR Directive 7-17
7.4.3 .NTYPE Directive 7-19
7.5 .ERROR AND .PRINT DIRECTIVES 7-20
7.6 INDEFINITE REPEAT BLOCK DIRECTIVES: .IRP and .IRPC 7-22
7.6.1 .IRP Directive 7-22
7.6.2 .IRPC Directive 7-26
7.7 REPEAT BLOCK DIRECTIVE: .REPT, .ENDR 7-27
7.8 MACRO LIBRARY DIRECTIVE: .MCALL 7-28

APPENDIX A CROSS ASSEMBLER CHARACTER SETS A-1

A.1 ASCII CHARACTER SET A-1
A.2 RADIX-50 CHARACTER SET A-5

APPENDIX B CROSS ASSEMBLER LANGUAGE AND DIRECTIVES B-1

B.1 SPECIAL CHARACTERS B-1
B.2 CROSS ASSEMBLER OPERATORS AND DIRECTIVES B-3

APPENDIX C ERROR MESSAGES C-1

FIGURES

FIGURE	6-1	Assembly Listing Table of Contents	6-11
	6-2	Example of .ENABL and .DSABL Directives	6-17
	7-1	Example of .NARG Directive	7-16
	7-2	Example of .NCHR Directive	7-18
	7-3	Example of .IRP and IRPC Directives	7-25

TABLES

TABLE	3-1	Special Characters used in the Cross Assembler	3-2
	3-2	Legal Separating Characters	3-4
	3-3	Legal Argument Delimiters	3-5
	3-4	Legal Unary Operators	3-6
	3-5	Legal Binary Operators	3-7
	6-1	General Cross-Assembler Directives	6-2
	6-2	Symbolic Arguments of Listing Control Directives	6-7
	6-3	Symbolic Arguments of Function Control Directives	6-16
	6-4	Symbolic Arguments of .PSECT Directive	6-37
	6-5	Program Section Default Value	6-41
	6-6	Legal Condition Tests for Conditional Assembly Directives	6-46
	6-7	Subconditional Assembly Block Directives	6-49

CHAPTER 1

THE EMULOGIC CROSS ASSEMBLER

1.1 INTRODUCTION

The Emulogic Cross Assembler assembles one or more ASCII source files containing assembler statements into a single relocatable binary object file. The output of the Cross Assembler consists of a binary object file and a file containing the table of contents, the assembly listing, and the symbol table list. An optional cross-reference listing of symbols and macros is also available.

These are some of the features of the Cross Assembler:

- o Source and command string control of assembly functions,
- o Device and filename specifications for input and output files,
- o Error listing on command output device,
- o Alphabetized, formatted symbol table listing (with optional cross-reference listing of symbols),
- o Relocatable object modules,
- o Global symbols for linking object modules,
- o Conditional assembly directives,
- o Program sectioning directives,
- o User-defined macros and macro libraries, and
- o Extensive source and command string control of listing functions.

1.2 ASSEMBLY PASS 1

During pass 1, the Cross Assembler locates and reads all required macros from libraries, builds symbol tables and program section tables for the program, and performs a rudimentary assembly of each source statement.

In the first step of assembly pass 1, the Cross Assembler initializes all the impure data areas (areas containing both code and data) that will be used internally for the assembly process. These areas include all dynamic storage and buffer areas used as file storage regions.

The Cross Assembler then calls a system subroutine which transfers a command line into memory. This command line contains the specifications of all files to be used during assembly. After scanning the command line for proper syntax, the Cross Assembler initializes the specified output files. These files are opened to determine if valid output file specifications have been passed in the command line.

The Cross Assembler now initiates a routine which retrieves source lines from the input file. If no input file is open, as is the case at the beginning of assembly, the Cross Assembler opens the next input file specified in the command line and starts assembling the source statements. The Cross Assembler first determines the length of the instructions, then assembles them according to length as one word, two words, or three words.

At the end of assembly pass 1, the Cross Assembler reopens the output files described above. Such information as the object module name, the program version number, and the global symbol directory (GSD) for each program section are output to the object file to be used later in linking the object modules. After writing out the GSD for a given program section, the Cross Assembler scans through the symbol tables to find all the global symbols that are bound to that particular program section. The Cross Assembler then writes out GSD records to the object file for these symbols. This process is done for each program section.

1.3 ASSEMBLY PASS 2

On pass 2 the Cross Assembler writes the object records to the output file while generating both the assembly listing and the symbol table listing for the program. A cross-reference listing may also be generated.

Basically, assembly pass 2 consists of the same steps performed in assembly pass 1, except that all source statements containing errors that the Cross Assembler detects are flagged with an error code as the assembly listing file is created. The object file that is created at the conclusion of pass 2 contains all the object records, together with relocation records that hold the information necessary for linking the object file.

The information in the object file, when passed to the Linker, enables the global symbols in the object modules to be associated with absolute memory addresses, thereby forming an executable body of code.

CHAPTER 2

SOURCE PROGRAM FORMAT

2.1 PROGRAMMING STANDARDS AND CONVENTIONS

Programming standards and conventions allow code written by a person (or group) to be easily understood by other people. These standards also make the program easier to

Plan
Comprehend
Test
Modify
Convert

The actual standard used must meet local user requirements.

2.2 STATEMENT FORMAT

A source program is composed of assembly-language statements. Each statement must be completed on one line. Although a line may contain 132 characters (a longer line causes an error (L) in the assembly listing), a line of 80 characters is recommended because of constraints imposed by listing format and terminal line size. Blank lines, although legal, have no significance in the source program.

A cross assembler statement may have as many as four fields. These fields are identified by their order within the statement and/or by the separating characters between the fields. The general format of the cross assembler statement is:

```
[Label:]   Operator    Operand    [;Comment(s)]
```

The label and comment fields are optional. The operator and operand fields are interdependent; in other words, when both fields are present in a source statement, each field is evaluated by the Cross Assembler in the context of the other.

A statement may contain an operator and no operand, but the reverse is not true. A statement containing an operand with no operator is illegal and is interpreted by the Cross Assembler during assembly as an implicit .WORD directive.

The Cross Assembler interprets and processes source program statements one by one. Each statement causes the Cross Assembler to either perform a specified assembly process or to generate one or more binary instructions or data words.

2.2.1 Label Field

A label is a user-defined symbol which is assigned the value of the current location counter and is entered into the user-defined symbol table. The current location counter is used by the Cross Assembler to assign memory addresses to the source program statements as they are encountered during the assembly process. Thus, a label is a means of symbolically referring to a specific statement.

When a program section is absolute, the value of the current location counter is absolute; its value references an absolute virtual memory address (such as location 100). Similarly, when a program section is relocatable, the value of the current location counter is relocatable; a relocation bias calculated at link time is added to the apparent value of the current location counter to establish its effective absolute virtual address at execution time. (For a discussion of program sections and their attributes, see Section 6.7.)

NOTE

> Examples in this document will use mnemonics for the LSI-11 chip.

If present, a label must be the first field in a source statement and must be terminated by a colon (:). For example, if the value of the current location counter is absolute 100, the statement:

```
ABCD:   MOV    A,B
```

(where ABCD is the label, MOV an LSI-11 chip-specific operator, and A and B chip-specific operands) assigns the value 100 to the label ABCD. If the location counter value were relocatable, the final value of ABCD would be 100+K, where K represents the relocation bias of the program section, as calculated by the Linker at link time.

More than one label may appear within a single label field. Each label so specified is assigned the same address value. For example, if the value of the current location counter is 100, the multiple labels in the following statement are each assigned the value 100:

```
ABC:     $DD:     A7.7:    MOV      A,B
```

Multiple labels may also appear on successive lines. For example, the statements

```
ABC:
$DD:
A7.7:    MOV      A,B
```

likewise cause the same value to be assigned to all three labels. This second method of assigning multiple labels is preferred because positioning the fields consistently within the source program makes the program easier to read (see Section 2.3).

A double colon (::) defines the label as a global symbol. For example, the statement

```
ABCD::   MOV      A,B
```

establishes the label ABCD as a global symbol. The distinguishing attribute of a global symbol is that it can be referenced from within an object module other than the module in which the symbol is defined or by independently assembled object modules. References to this label in other modules are resolved when the modules are linked as a composite executable image.

The legal characters for defining labels are

A through Z
0 through 9
. (Period)
$ (Dollar Sign)

Although a label may be any length, only the first six characters are significant and, therefore, must be unique among all the labels in the source program. If the first six characters in two or more labels are the same, the assembly listing will show an error code (M) signaling a multiple definition.

A symbol used as a label must not be redefined within the source program. If the symbol is redefined, a label with a multiple definition results, causing the Cross Assembler to generate an error code (M) in the assembly listing. Furthermore, any statement in the source program which references a multi-defined label generates an error code (D) in the assembly listing.

2.2.2 Operator Field

The operator field specifies the action to be performed. It may consist of an instruction mnemonic (op code), a cross assembler directive, or a macro call. Chapters 6 and 7 describe these three types of operators.

When the operator is an instruction mnemonic, a machine instruction is generated and the Cross Assembler evaluates the addresses of the operands which follow. When the operator is a directive, the Cross Assembler performs certain control actions or processing operations during assembly of the source program. When the operator is a macro call, the Cross Assembler inserts the code generated by the macro expansion.

Leading and trailing spaces or tabs in the operator field have no significance; such characters serve only to separate the operator field from the preceding and following fields.

A space, tab, or any non-RAD50 character terminates an operator (Appendix A contains a table of Radix-50 characters). The following examples use an LSI-11 chip-specific operator, MOV :

```
MOV A,B   ;The space terminates the operator MOV.

MOV     A,B     ;The tab terminates the operator MOV.

MOV!A,B   ;The ! character terminates the operator MOV.
```

Although the statements above are all equivalent in function, the second statement is the recommended form because it conforms to the Cross Assembler's coding conventions.

2.2.3 Operand Field

When the operator is an instruction mnemonic (op code), the operand field contains legal program variables that are to be evaluated/manipulated by the operator. The operand field may also supply arguments to the Cross Assembler's directives and macro calls, as described in Chapters 6 and 7, respectively.

Operands may be expressions or symbols, depending on the operator. Multiple expressions used in the operand field of a cross-assembler statement must be separated by a comma; multiple symbols similarly used may be delimited by any legal separator (a comma, tab, and/or space). An operand should be preceded by an operator field; if it is not, the statement is treated by the Cross Assembler as an implicit .WORD directive.

When the operator field contains an op code, associated operands are always expressions, as shown in the following statement:

```
        MOV     R0,A+2(R1)
```

On the other hand, when the operator field contains a cross-assembler directive or a macro call, associated operands are normally symbols, as shown in the following statement:

```
        .MACRO  ALPHA   SYM1,SYM2
```

Refer to the description of each cross-assembler directive (Chapter 6) to determine the type and number of operands required in issuing the directive.

The operand field is terminated by a semicolon when the field is followed by a comment. For example, in the following statement:

```
        LABEL:  MOV     A,B     ;Comment field
```

the tab between MOV and A terminates the operator field and defines the beginning of the operand field; a comma separates the operands A and B; and a semicolon terminates the operand field and defines the beginning of the comment field. When no comment field follows, the operand field is terminated by the end of the source line.

2.2.4 Comment Field

The comment field normally begins in column 33 and extends through the end of the line. This field is optional and may contain any ASCII characters except null, RUBOUT, carriage-return, line-feed, vertical-tab or form-feed. All other characters appearing in the comment field (even special characters reserved for use in the Cross Assembler) are checked only for ASCII legality and then included in the assembly listing as they appear in the source text.

All comment fields must begin with a semicolon (;). When lengthy comments extend beyond the end of the source line (column 80), the comment may be resumed in a following line. Such a line must contain a leading semicolon, and it is suggested that the body of the comment be continued in the same columnar position in which the comment began. A comment line can also be included as an entirely separate line within the code body.

Comments do not affect assembly processing or program execution. However, comments are necessary in source listings for later analysis, debugging, or documentation purposes.

2.3 FORMAT CONTROL

Horizontal formatting of the source program is controlled by the space and tab characters. These characters have no effect on the assembly process unless they are embedded within a symbol, number, or ASCII text string, or unless they are used as the operator field terminator. Thus, the space and tab characters can be used to provide an orderly and readable source program.

A standard source line format is shown below:

```
Label    - begins in column 1
Operator - begins in column 9
Operands - begin in column 17
Comments - begin in column 33.
```

These formatting conventions are not mandatory; free-field coding is permissible. However, note the increased readability after formatting in the example below.

unformatted line:

```
REGTST:BIT MASK,VALUE;COMPARES BITS IN OPERANDS.
```

formatted line using the above column settings:

```
REGTST: BIT     MASK,VALUE      ;Compares bits in operands.
```

Page formatting and assembly listing considerations are discussed in Chapter 6 in the context of cross-assembler directives that may be specified to accomplish desired formatting operations.

CHAPTER 3

SYMBOLS AND EXPRESSIONS

This chapter describes the components of the Cross Assembler´s instructions: the character set, the conventions observed in constructing symbols, and the use of numbers, operators, terms and expressions.

3.1 CHARACTER SET

The following characters are legal in cross-assembler source programs:

1. The letters A through Z. Both uppercase and lowercase letters are acceptable, although, upon input, lowercase letters are converted to uppercase (see Section 6.2.1, .ENABL LC).

2. The digits 0 through 9.

3. The characters "." (period) and "$" (dollar sign).

4. The special characters listed in Table 3-1 below.

Character	Designation	Function
:	Colon	Label terminator.
::	Double colon	Label terminator; defines the label as a global label.
=	Equal sign	Direct assignment operator and macro keyword indicator.
==	Double equal sign	Direct assignment operator; defines the symbol as a global symbol.
<CTRL>I	Tab	Item or field terminator.
	Space	Item or field terminator.
#	Number sign	**
@	At sign	**
(	Left parenthesis	**
)	Right parenthesis	**
.	Period	Current location counter.
,	Comma	Operand field separator.
;	Semicolon	Comment field indicator.
<	Left angle bracket	Initial argument or expression indicator.
>	Right angle bracket	Terminal argument or expression indicator.

TABLE 3-1: Special Characters Used in the Cross Assembler

+	Plus sign	Arithmetic addition operator or **
-	Minus sign	Arithmetic subtraction operator or **
*	Asterisk	Arithmetic multiplication operator.
/	Slash	Arithmetic division operator.
&	Ampersand	Logical AND operator.
!	Exclamation point	Logical inclusive OR operator.
"	Double quote	Double ASCII character indicator
'	Single quote	Single ASCII character indicator; or concatenation indicator.
^	Up arrow or circumflex	Universal unary operator or argument indicator.
\	Backslash	Macro call numeric argument indicator.
[	Left square bracket	**
]	Right square bracket	**

TABLE 3-1 (cont.): Special Characters Used in the Cross Assembler

** Refer to chapter 5 of this manual for chip-specific syntax.

3.1.1 Separating and Delimiting Characters

Legal separating characters and legal argument delimiters are defined in Tables 3-2 and 3-3 respectively.

Character	Definition	Usage
Space	One or more spaces and/or tabs	A space is a legal separator between instruction fields and between symbolic arguments within the operand field. Spaces within expressions are ignored.
,	Comma	A comma is a legal separator between symbolic arguments within the operand field. Multiple expressions used in the operand field must be separated by a comma.

TABLE 3-2: Legal Separating Characters

3.1.2 Illegal Characters

A character is illegal for one of two reasons:

1. If a character is not an element of the recognized cross-assembler character set, it is replaced in the listing by a question mark, and an error code (I) is printed in the assembly listing. The exception to this is an embedded null which, when detected, terminates the scan of the current line.

2. If a legal cross-assembler character is used in a source statement with illegal or questionable syntax, an error code (Q) is printed in the assembly listing.

Character	Definition	Usage
<...>	Paired angle brackets	Paired angle brackets may be used anywhere in a program to enclose an expression for treatment as a single term. Paired angle brackets are also used to enclose a macro argument, particularly when that argument contains separating characters
^x...x	Up-arrow (unary operator) construction, where the up-arrow is followed by an argument that is bracketed by any paired printing characters (x).	This construction is equivalent in function to the paired angle brackets described above and is generally used only where the argument itself contains angle brackets.

TABLE 3-3: Legal Argument Delimiters

3.1.3 Unary and Binary Operators

Legal cross-assembler unary operators are described in Table 3-4. Unary operators are used in connection with single terms (arguments or operands) to indicate an action to be performed on that term during assembly. Because a term preceded by a unary operator is considered to contain that operator, a term so specified can be used alone or as an element of an expression.

Unary Operator	Explanation	Example	Effect
+	Plus sign	+A	Produces the positive value of A.
-	Minus sign	-A	Produces the negative (2's complement) value of A.
^	Up-arrow, universal unary operator	^D127	Interprets 127 as a decimal number
		^O34	Interprets 34 as an octal number.
		^H0B3	Interprets B3 as a hexadecimal number. If the first character after H is A-F, precede the first character with a zero.
		^B11000111	Interprets 11000111 as a binary number.
		^RABC	Evaluates ABC in Radix-50 form.

TABLE 3-4: Legal Unary Operators

Unary operators can be used adjacent to each other or in constructions involving multiple terms, as shown below:

```
-^D50       (Equivalent to -<^D50>)
^C^012      (Equivalent to ^C<^012>)
```

Legal cross-assembler binary operators are described in Table 3-5. In contrast to unary operators, binary operators specify actions to be performed on multiple items or terms within an expression.

Binary Operator	Explanation	Example
+	Addition	A+B
-	Subtraction	A-B
*	Multiplication	A*B (signed 16-bit product returned)
/	Division	A/B (signed 16-bit quotient returned)
&	Logical AND	A&B
!	Logical inclusive OR	A!B

TABLE 3-5: Legal Binary Operators

All binary operators have equal priority. Terms enclosed by angle brackets are evaluated first, and remaining operations are performed from left to right, as shown in the examples below:

```
.WORD    1+2*3                    ;Equals 9.
.WORD    1+<2*3>                  ;Equals 7.
```

3.2 CROSS-ASSEMBLER SYMBOLS

The Cross Assembler maintains a symbol table for each of the three symbol types that may be defined in a cross-assembler source program: the Permanent Symbol Table (PST), the User Symbol Table (UST), and the Macro Symbol Table (MST). The PST contains all the permanent symbols defined within (and thus automatically recognized by) the Cross Assembler and is part of the cross-assembler image. The UST (for user-defined symbols) and MST (for macro symbols) are constructed as the source program is assembled.

3.2.1 Permanent Symbols

Permanent symbols consist of the instruction mnemonics and cross-assembler directives (see chapters 5, 6 and 7 and Appendix B). These symbols are a permanent part of the cross-assembler image and need not be defined before being used in the operator field of a cross-assembler source statement.

3.2.2 User-Defined and Macro Symbols

User-defined symbols are those symbols that are equated to a specific value through a direct assignment statement, that appear as labels, or that act as dummy arguments. These symbols are added to the User Symbol Table as they are encountered during assembly.

Macro symbols are those symbols used as macro names. They are added to the Macro Symbol Table as they are encountered during assembly.

The following rules govern the creation of user-defined and macro symbols:

1. Symbols can be composed of alphanumeric characters, dollar signs ($), and periods (.) only.

2. The first character of a symbol must be an alphanumeric (except in the case of local symbols--see Section 3.4).

3. The first six characters of a symbol must be unique. A symbol can be written with more than six legal characters, but the seventh and subsequent characters are checked only for ASCII legality and are not otherwise evaluated or recognized by the Cross Assembler.

4. Spaces, tabs, and illegal characters must not be embedded within a symbol. The legal cross-assembler character set is defined in Section 3.1.

The value of a symbol depends upon its use in the program. A symbol in the operator field may be any one of the three symbol types described above: permanent, user-defined, or macro. To determine the value of an operator-field symbol, the Cross Assembler searches the symbol tables in the following order:

1. Macro Symbol Table

2. Permanent Symbol Table

3. User-Defined Symbol Table

This search order allows permanent symbols to be used as macro symbols. But the user must keep in mind the sequence in which the search for symbols is performed in order to avoid incorrect interpretation of the symbol's use.

When a symbol appears in the operand field, the search order is:

1. User-Defined Symbol Table

2. Permanent Symbol Table

Depending on their use in the source program, user-defined symbols have either a local (internal) attribute or a global (external) attribute.

Normally, the Cross Assembler treats all user-defined symbols as local; that is, their definition is limited to the module in which they appear. However, symbols can be explicitly declared to be global symbols through one of three methods:

1. Use of the .GLOBL directive (see Section 6.8.1).

2. Use of the double colon (::) in defining a label.

3. Use of the double equal sign (==).

All symbols within a module that remain undefined at the end of assembly are treated as default global references if the .ENABLE directive has been used.

NOTE

At the end of assembly, statements containing undefined symbols are flagged with an error code (U) in the assembly listing.

Global symbols provide linkages between independently assembled object modules within the task image. A global symbol defined as a label, for example, may serve as an entry-point address to another section of code within the image. Such symbols are referenced from other source modules in order to transfer control throughout execution. These global symbols are resolved at link time, ensuring that the resulting image is a logically coherent and complete body of code.

3.3 DIRECT ASSIGNMENT STATEMENTS

The general format for a direct assignment statement is:

```
symbol=expression
    or
symbol==expression
```

where: expression - can have only one level of forward reference (see 5. below).

- cannot contain an undefined global reference.

The direct assignment statements above allow the user to equate a symbol with a specific value. After the symbol has been defined, it is entered into the User-Defined Symbol Table. If the general format is used (= or ==), the value of the symbol may be changed in subsequent direct assignment statements.

A direct assignment statement embodying the double equal (==) sign, as shown above, defines the symbol as global (see Section 6.8.1).

The following examples illustrate the coding of direct assignment statements.

Example 1:

```
A=10                    ;Direct assignment

B==30                   ;Global assignment

A=15                    ;Legal reassignment
```

Example 2

```
C:
D=.                                 ;The symbol D is equated to
E:      MOV     #1,ABLE             ;".", and the labels C and E
                                    ;are assigned a value that
                                    ;is equal to the address
                                    ;of the MOV instruction.
```

The code in the second example above would not normally be used and is shown only to illustrate the performance of the Cross Assembler in such situations. Refer to section 3.5 for a description of the period (.) as the current location counter symbol.

The following conventions apply to the coding of direct assignment statements:

1. An equal sign (=) or double equal sign (==) must separate the symbol from the expression defining the symbol's value. Spaces preceding and/or following the direct assignment operators, although permissible, have no significance in the resulting value.

2. The symbol being assigned in a direct assignment statement is placed in the label field.

3. Only one symbol can be defined in a single direct assignment statement.

4. A direct assignment statement may be followed only by a comment field.

5. Only one level of forward referencing is allowed. The following example would cause an error code (U) in the assembly listing on the line containing the illegal forward reference:

```
X=Y     (Illegal forward reference)

Y=Z     (Legal forward reference)

Z=1
```

Although one level of forward referencing is allowed for local symbols, no forward referencing is allowed for global symbols. In other words, the expression being assigned to a global symbol can contain only previously defined symbols. A forward reference in a direct assignment statement defining a global symbol will cause an error code (A) to be generated in the assembly listing.

3.4 LOCAL SYMBOLS

Local symbols are specially formatted symbols used as labels within a block of coding that has been delimited as a Local Symbol Block (LSB). Local symbols are of the form n$, where n is a decimal integer from 1 to 65535, inclusive. Examples of local symbols are:

```
1$
27$
59$
104$
```

A local symbol block is delimited in one of three ways:

1. The range of a local symbol block usually consists of those statements between two normally constructed symbolic labels. Note that a statement of the form:

   ALPHA=EXPRESSION

   is a direct assignment statement (see Section 3.3) but does not create a label and thus does not delimit the range of a local symbol block.

2. The range of a local symbol block is normally terminated upon encountering a .PSECT or .ASECT directive in the source program.

3. The range of a local symbol block is delimited through cross-assembler directives, as follows:

   Starting delimiter: .ENABL LSB

   Ending delimiter: .ENABL LSB

   or

   one of the following:

   Symbolic label (See Section 2.2.1)
   .PSECT (see Section 6.7.1)
   .ASECT (see Section 6.7.2)

   encountered after a .DSABL LSB (see Section 6.2.1).

Local symbols provide a convenient means of generating labels for branch instructions and other such references within local symbol blocks. Using local symbols reduces the possibility of symbols with multiple definitions appearing within a user program. In addition, the use of local symbols differentiates entry-point labels from local labels, since local symbols cannot be referenced from outside their respective local symbol blocks. Thus, local symbols of the same name can appear in other local symbol blocks without conflict. Local symbols do not appear in cross-reference listings and require less symbol table space than other types of symbols. Their use is recommended.

When defining local symbols, use the range from 1$ to 29999$ first. Local symbols within the range 30000$ through 65535$, inclusive, can be generated automatically as a feature of the Cross Assembler. Such local symbols are useful in the expansion of macros during assembly (see Section 7.3.4).

Be sure to avoid multiple definitions of local symbols within the same local symbol block. For example, if the local symbol 10$ is defined two or more times within the same local symbol block, each symbol represents a different address value. Such a multi-defined symbol causes an error code (P) to be generated in the assembly listing.

3.5 CURRENT LOCATION COUNTER

The period (.) is the symbol for the current location counter. When used in the operand field of an instruction, the period represents the address of the first word of the instruction. When used in the operand field of a cross-assembler directive, it represents the address of the current byte or word, as shown in the example below.

```
SAL=0
          .WORD   1234,.+4,SAL     ;The operand .+4 in the .WORD
                                   ;directive represents a value
                                   ;that is stored as the second
                                   ;of three words during
                                   ;assembly.
```

Assume that the current value of the location counter is 500. During assembly, the Cross Assembler reserves storage in response to the .WORD directive (see Section 6.3.2), beginning with location 500. The operands accompanying the .WORD directive determine the values so stored. The value 1234 is thus stored in location 500. The value represented by .+4 is stored in location 502; this value is derived as the current value of the location counter (which is now 502), plus the absolute value 4, thereby depositing the value 506 in location 502. Finally, the value of SAL, previously equated to 0, is deposited in location 504. Figure 3-1 illustrates the result of the example.

LOCATION	CONTENTS
500	1234
502	506
504	0

Figure 3-1: Sample Assembly Results

At the beginning of each assembly pass, the Cross Assembler resets the location counter. Normally, consecutive memory locations are assigned to each byte of object data generated. However, the value of the location counter can be changed through a direct assignment statement of the following form:

```
.=expression
```

The current location counter symbol (.) is either absolute or relocatable, depending on the attribute of the current program section.

The attribute of the current location counter can be changed only through the program sectioning directives (.PSECT and .ASECT), as described in Section 6.7. Therefore, assigning to the counter an expression having an attribute different than that of the current program section will generate an error code (A) in the assembly listing.

Furthermore, an expression assigned to the counter may not contain a forward reference (a reference to a symbol that is not previously defined). The user must also be sure that the expression assigned will not force the counter into another program section, even if both sections involved have the same relocatability. Either of these conditions causes the Cross Assembler to generate incorrect object file code, and may cause statements following the error to be flagged with an error code (P) in the assembly listing.

The following coding illustrates the use of the current location counter:

```
           .RADIX 16
           .ASECT
.=500                                ;Set location counter to
                                     ;absolute 500(hex).
FIRST:     MOV       .+10,COUNT      ;The label "FIRST" has the value
                                     ;500(hex).
                                     ;.+10 equals 510(hex). The
                                     ;contents of the location
                                     ;510(hex) will be deposited
                                     ;in the location "COUNT".
.=520                                ;The assembly location counter
                                     ;now has a value of
                                     ;absolute 520(hex).
SECOND:    MOV       .,INDEX         ;The label "SECOND" has the
                                     ;value 520(hex).
                                     ;The contents of location
                                     ;520(hex), that is, the binary
                                     ;code for the instruction
                                     ;itself, will be deposited in
                                     ;the location "INDEX".
           .PSECT
.=.+20                               ;Set location counter to
                                     ;relocatable 20 of the
                                     ;unnamed program section.
THIRD:     .WORD     0               ;The label "THIRD" has the
                                     ;value of relocatable 20.
```

Storage areas may be reserved in the program by advancing the location counter. For example, if the current value of the location counter is 1000, each of the following statements:

```
.=.+40

or

.BLKB 40

or

.BLKW 20

or
.BLKL 10  (supported only for cross assemblers
          having 32-bit data)
```

reserves 40 bytes of storage space in the source program. The .BLKB, .BLKW, and .BLKL directives, however, are the preferred ways to reserve storage space (see Section 6.5.3).

3.6 NUMBERS

The Cross Assembler assumes that all numbers in the source program are to be interpreted in octal radix, unless otherwise specified. This default radix can be altered with the .RADIX directive (see Section 6.4.1.1). Also, individual numbers can be designated as decimal, binary, octal, or hexadecimal numbers through temporary radix control operators (see Section 6.4.1.2).

For every statement in the source program that contains a digit that is not in the current radix, an error code (N) is generated in the assembly listing. However, the Cross Assembler continues with the scan of the statement and evaluates each such number encountered as a decimal value.

Negative numbers must be preceded by a minus sign; the Cross Assembler translates such numbers into two's complement form. Positive numbers may (but need not) be preceded by a plus sign.

A number containing more than 16 significant bits (greater than FFFF (hex)) is truncated from the left and flagged with an error code (T) in the assembly listing (except for cross assemblers supporting 32-bit data).

Numbers are always considered to be absolute values; therefore, they are never relocatable.

3.7 TERMS

A term is a component of an expression and may be one of the following:

1. A number, as defined in Section 3.6.

2. A symbol, as defined in Section 3.2. Symbols are evaluated as follows:

   A. A period (.) specified in an expression causes the value of the current location counter to be used.

   B. A defined symbol is located in the User-Defined Symbol Table (UST) and its value is used.

   C. A permanent symbol's basic value is used.

   D. An undefined symbol is assigned a value of zero and inserted in the User-Defined Symbol Table as an undefined default global reference. If the .DSABL GBL directive (see Section 6.2.1) is in effect, the automatic global reference default function of the Cross Assembler is inhibited, and the statement containing the undefined symbol is flagged with an error code (U) in the assembly listing.

3. A single quote followed by a single ASCII character, or a double quote followed by two ASCII characters. This type of expression construction is explained in detail in Section 6.3.3.

4. An expression enclosed in angle brackets (<>). Any expression so enclosed is evaluated and reduced to a single term before the remainder of the expression in which it appears is evaluated. Angle brackets, for example, may be used to alter the left-to-right evaluation of expressions (as in A*B+C versus A*<B+C>), or to apply a unary operator to an entire expression (as in -<A+B>).

5. A unary operator followed by a symbol or number.

3.8 EXPRESSIONS

Expressions are combinations of terms joined together by binary operators (see Table 3-5). The evaluation of an expression includes the determination of its attributes. A resultant expression value may be any one of three types (as described later in this section): relocatable, absolute or external.

Expressions are evaluated from left to right with no operator hierarchy rules, except that unary operators take precedence over binary operators. A term preceded by a unary operator is considered to contain that operator. (Terms are evaluated, where necessary, before their use in expressions.) Multiple unary operators are valid and are treated as follows:

```
-+-A
```

is equivalent to:

```
-<+<-A>>
```

NOTE

> The maximum depth of an expression is governed by the Cross Assembler's expression stack space. If an expression exceeds the Cross Assembler's maximum expression depth, the statement is marked with an (E) error, and processing continues.

A missing term, expression, or external symbol is interpreted as a zero. A missing or illegal operator terminates the expression analysis, causing error codes (A) and/or (Q), to be generated in the assembly listing, depending on the context of the expression itself. For example, the expression:

```
A + B 177777
```

is evaluated as

```
A + B
```

because the first non-blank character following the symbol B is not a legal binary operator, an expression separator (a comma), or an operand field terminator (a semicolon or the end of the source line).

NOTE

Spaces within expressions can serve as delimiters only between symbols. In other words, the expressions

A + B

and

A+B

are the same, but the symbols

B17

and

B 17

are different because B 17 is not a single symbol.

At assembly time the value of an external (global) expression is equal to the value of the absolute part of that expression. For example, the expression EXTERN+A, where "EXTERN" is an external symbol, has a value at assembly time that is equal to the value of the internal (local) symbol A. When evaluated at link time, however, this expression takes on the resolved value of the symbol EXTERN, plus the value of symbol A.

Expressions, when evaluated by the Cross Assembler, are one of three types: relocatable, absolute or external. The following distinctions are important:

1. An expression is relocatable if its value is fixed relative to the base address of the relocatable program section in which it appears; it will have an offset value added at link time. Terms that contain labels defined in relocatable program sections will have a relocatable value; similarly, a period (.) in a relocatable program section, representing the value of the current location counter, will also have a relocatable value.

2. An expression is absolute if its value is fixed. An expression whose terms are numbers and ASCII conversion

characters will reduce to an absolute value. A relocatable expression or term minus a relocatable term, where both elements being evaluated belong to the same program section, is an absolute expression. This is because every term in a program section has the same relocation bias. When one term is subtracted from another, the resulting bias is zero. The Cross Assembler can then treat the expression as absolute and reduce it to a single term upon completion of the expression scan. Terms that contain labels defined in an absolute section will also have an absolute value.

3. An expression is external (or global) if it contains a single global reference (plus or minus an absolute expression value) that is not defined within the current program. Thus, an external expression is only partially defined following assembly and must be resolved at link time.

The evaluation of relocatable and external expressions is completed at link time.

CHAPTER 4

RELOCATION AND LINKING

The output of the Cross Assembler is an object module composed of relocatable machine language code, relocation information, and a corresponding global symbol table list that defines the use of symbols within the program. To form an executable program, the object module must be processed by the Emulogic Linker, ELINKx (where x is 2, 3, 4 or 6 depending upon your particular microprocessor; see Table 3-3 in section 3.6 of your Emulogic Cross Assembler User's Guide).

ELINKx produces an executable load module with all locations resolved as absolute locations. This absolute load file in LDA format (or XDA format for the Z8000 and 68000 microprocessors) is the only loadable file format produced by Emulogic's Linker.

To allow the value of an expression to be fixed at link time, the Cross Assembler outputs certain instructions in the object file, together with other required parameters. For relocatable expressions in the object module, the base of the associated relocatable program section is added to the value of the relocatable expression provided by the Cross Assembler. For external expression values, the value of the external term in the expression (since the external symbol must be defined in one of the other object modules being linked together) is determined and then added to the absolute portion of the external expression, as provided by the Cross Assembler.

All instructions that require modification at link time are flagged in the assembly listing, as illustrated in the example below. The apostrophe (') following the octal expansion of the instruction indicates that simple relocation is required; the letter G indicates that the value of an external symbol must be added to the absolute portion of an expression.

EXAMPLE:

```
005065     CLR     RELOC             ;Assuming that the value of the
000040'                              ;symbol "RELOC", 40, is
                                     ;relocatable, the relocation
                                     ;bias will be added to this
                                     ;value.

005065     CLR     EXTERN            ;The value of the symbol "EXTERN"
000000G                              ;is assembled as zero and is
```

```
                                        ;resolved at link time.

005065      CLR      EXTERN+6           ;The value of the symbol "EXTERN"
000006G                                 ;is resolved at link time
                                        ;and added to the absolute
                                        portion (+6) of the expression.
```

For directions on using the Linker, refer to Chapter 3 in the Emulogic Cross Assembler User´s Guide.

CHAPTER 5

THE 6500 FAMILY INSTRUCTION SET

5.1 INTRODUCTION

This chapter provides information for writing software programs to run on the 6500 family of microprocessors. The chapter contains

- programming notes,
- a summary of the 6500 instruction set, and
- a sample Emulogic 6500 Cross Assembler output listing.

NOTE

You are expected to have some familiarity with the 6500 family of microprocessors and to have access to the chip manufacturer's manuals or equivalent documentation. This chapter is not a tutorial on programming the 6500 family, nor should it serve as your only reference.

5.2 PROGRAMMING NOTES

1. The Emulogic 6500 Cross Assembler lets you reference the low or high byte of a word. Examples:

```
LDA   TAG(L)
ORA   TAG(H)
```

2. You are allowed to force absolute and absolute indexed addressing modes for certain instructions. Examples:

```
AND  TAG(A)
AND  TAG(A),X
```

3. You can force zero page and zero page indexed addressing modes for certain instructions. Examples:

```
LDY  TAG(Z)
LDY  TAG(Z),Y
```

4. If address zero is referenced, the instruction will be assembled with the extended addressing mode instead of the direct addressing mode. To have the instruction assemble address zero as a single byte, you must use the force-zero-page syntax.

5. Addressing references in zero page must be predefined.

6. Avoid using complex forward references because they may result in phasing errors. When a complex forward reference is made, however, it can be forced absolute to avoid a phasing error. Example:

```
LDX    TAG+3(A)
```

(TAG is a forward reference.)

5.3 REGISTERS

16-BIT REGISTER

PC - Program Counter

8-BIT REGISTERS

A - Accumulator
X - Index
Y - Index

STATUS REGISTER

```
bits:   7 6 5 4 3 2 1 0
flags:  N V   B D I Z C
```

N = Negative
V = Overflow
B = Break
D = Decimal mode
I = Interrupt disable
Z = Zero
C = Carry

5.4 6500 FAMILY INSTRUCTION SET

Instruction operands are represented as follows:

Operand	Meaning
ii	Immediate operand (8 bit)
nn	Immediate operand (16 bit)
^^	8-bit relative branch address
aa	8-bit address variable (zero page)
aaaa	16-bit absolute address
x	X index register
y	Y index register
A	Accumulator
OPER	Operand (absolute and absolute indexed addressing modes)
ZPAGE	Zero page
(L)	Low-order byte
(H)	High-order byte
(A)	Force absolute
(Z)	Force zero page

TAG = absolute, global, or relocatable reference

MNEMONIC	OPERANDS	DESCRIPTION	EXAMPLE
ADC	#ii	Add immediate to accumulator with carry	ADC #20
ADC	aa	Add memory to accumulator with carry	ADC 3F
ADC	aaaa	Add memory to accumulator with carry	ADC 0FFF
ADC	aa,x	Add memory indexed to accumulator with carry	ADC 3,X
ADC	aaaa,x	Add memory indexed to accumulator with carry	ADC TAG,X
ADC	aaaa,y	Add memory indexed to accumulator with carry	ADC 245,Y

ADC	(aa,x)	Add memory indexed indirect to accumulator with carry	ADC (4,X)
ADC	(aa),y	Add memory indirect indexed to accumulator with carry	ADC (7),Y
ADC	#nn(L)	Add with carry the low-order byte of the immediate data to the accumulator	ADC #TAG(L)
ADC	#nn(H)	Add with carry the high-order byte of the immediate data to the accumulator	ADC #TAG(H)
ADC	OPER	Add with carry memory to accumulator	ADC TAG(A)
ADC	OPER,X	Add with carry memory to accumulator indexed by X	ADC TAG(A),X
ADC	OPER,Y	Add with carry memory to accumulator indexed by Y	ADC TAG(A),Y
ADC	ZPAGE	Add with carry memory to accumulator	ADC TAG(Z)
ADC	ZPAGE,X	Add with carry memory to accumulator indexed by X	ADC TAG(Z),X
AND	#ii	AND immediate with accumulator	AND #125.
AND	aa	AND memory with accumulator	AND 10
AND	aaaa	AND memory with accumulator	AND 258
AND	aa,x	AND memory indexed with accumulator	AND 5A,X
AND	aaaa,x	AND memory indexed with accumulator	AND 300,X
AND	aaaa,y	AND memory indexed with accumulator	AND TAG,Y
AND	(aa,x)	AND memory indexed indirect with accumulator	AND (0FE,X)
AND	(aa),y	AND memory indirect indexed	AND (25),Y

		with accumulator	
AND	#nn(L)	Logical AND the low-order byte of the immediate data and the accumulator	AND #TAG(L)
AND	#nn(H)	Logical AND the high-order byte of the immediate data and the accumulator	AND #TAG(H)
AND	OPER	Logical AND memory with accumulator	AND TAG(A)
AND	OPER,X	Logical AND memory with accumulator indexed by X	AND TAG(A),X
AND	OPER,Y	Logical AND memory with accumulator indexed by Y	AND TAG(A),Y
AND	ZPAGE	Logical AND memory with accumulator	AND TAG(Z)
AND	ZPAGE,X	Logical AND memory with accumulator indexed by X	AND TAG(Z),X
ASL	A	Shift left accumulator one bit	ASL A
ASL	aa	Shift left memory one bit	ASL 3B
ASL	aaaa	Shift left memory one bit	ASL TAG
ASL	aa,x	Shift left memory indexed one bit	ASL 0A4,X
ASL	aaaa,x	Shift left memory indexed one bit	ASL 1452,X
ASL	OPER	Shift left one bit (memory or accumulator)	ASL TAG(A)
ASL	OPER,X	Shift left one bit (memory or accumulator) indexed by X	ASL TAG(A),X
ASL	ZPAGE	Shift left one bit (memory or accumulator)	ASL TAG(Z)
ASL	ZPAGE,X	Shift left one bit (memory or accumulator) indexed by X	ASL TAG(Z),X

BCC	^^	Branch on carry clear	BCC 12
BCS	^^	Branch on carry set	BCS TAG
BEQ	^^	Branch on result zero	BEQ 34
BIT	aa	Test bits in memory with accumulator	BIT 0FF
BIT	aaaa	Test bits in memory with accumulator	BIT 257
BIT	OPER	Test bits in memory with accumulator	BIT TAG(A)
BIT	ZPAGE	Test bits in memory with accumulator	BIT TAG(Z)
BMI	^^	Branch on result minus	BMI 0A
BNE	^^	Branch on result not zero	BNE 2E
BPL	^^	Branch on result plus	BPL TAG1
BRK		Force break	BRK
BVC	^^	Branch on overflow clear	BVC 24
BVS	^^	Branch on overflow set	BVS 0AB
CLC		Clear carry flag	CLC
CLD		Clear decimal load	CLD
CLI		Clear interrupt disable bit	CLI
CLV		Clear overflow flag	CLV
CMP	#ii	Compare immediate to accumulator	CMP #250.
CMP	aa	Compare memory to accumulator	CMP 86
CMP	aaaa	Compare memory to accumulator	CMP 257
CMP	aa,x	Compare memory indexed to accumulator	CMP 4,X

CMP	aaaa,x	Compare memory indexed to accumulator	CMP OFFFF,X
CMP	aaaa,y	Compare memory indexed to accumulator	CMP TAG,Y
CMP	(aa,x)	Compare memory indexed indirect to accumulator	CMP (25,X)
CMP	(aa),y	Compare memory indirect indexed to accumulator	CMP (6F),Y
CMP	#nn(L)	Compare the low-order byte of the immediate data with the accumulator	CMP #TAG(L)
CMP	#nn(H)	Compare the high-order byte of the immediate data with the accumulator	CMP #TAG(H)
CMP	OPER	Compare memory with accumulator	CMP TAG(A)
CMP	OPER,X	Compare memory with accumulator indexed by X	CMP TAG(A),X
CMP	OPER,Y	Compare memory with accumulator indexed by Y	CMP TAG(A),Y
CMP	ZPAGE	Compate memory with accumulator	CMP TAG(Z)
CMP	ZPAGE,X	Compare memory with accumulator indexed by X	CMP TAG(Z),X
CPX	#ii	Compare immediate and index X	CPX #77
CPX	aa	Compare memory and index X	CPX 45
CPX	aaaa	Compare memory and index X	CPX 284
CPX	#nn(L)	Compare the low-order byte of the immediate data with Index X	CPX #TAG(L)
CPX	#nn(H)	Compare the high-order byte of the immediate data with Index X	CPX #TAG(H)
CPX	OPER	Compare memory and index X	CPX TAG(A)

CPX	ZPAGE	Compare memory and index X	CPX TAG(Z)
CPY	#ii	Compare immediate and index Y	CPY #2
CPY	aa	Compare memory and index Y	CPY 7F
CPY	aaaa	Compare memory and index Y	CPY 0FFA
CPY	#nn(L)	Compare the low-order byte of the immediate data with Index Y	CPY #TAG(L)
CPY	#nn(H)	Compare the high-order byte of the immediate data with Index Y	CPY #TAG(H)
CPY	OPER	Compare memory and index Y	CPY TAG(A)
CPY	ZPAGE	Compare memory and index Y	CPY TAG(Z)
DEC	aa	Decrement memory by one	DEC 80
DEC	aaaa	Decrement memory by one	DEC TAG
DEC	aa,x	Decrement memory indexed by one	DEC 7F,X
DEC	aaaa,x	Decrement memory indexed by one	DEC 3FA,X
DEC	OPER	Decrement memory by one	DEC TAG(A)
DEC	OPER,X	Decrement memory by one indexed by X	DEC TAG(A),X
DEC	ZPAGE	Decrement memory by one	DEC TAG(Z)
DEC	ZPAGE,X	Decrement memory by one indexed by X	DEC TAG(Z),X
DEX		Decrement index X by one	DEX
DEY		Decrement index Y by one	DEY
EOR	#ii	Exclusive OR immediate with accumulator	EOR #9
EOR	aa	Exclusive OR memory with	EOR 0F7

		accumulator	
EOR	aaaa	Exclusive OR memory with accumulator	EOR 100
EOR	aa,x	Exclusive OR memory indexed with accumulator	EOR 25,X
EOR	aaaa,x	Exclusive OR memory indexed with accumulator	EOR TAG,X
EOR	aaaa,y	Exclusive OR memory indexed with accumulator	EOR 200,Y
EOR	(aa,x)	Exclusive OR memory indexed indirect with accumulator	EOR (5,X)
EOR	(aa),y	Exclusive OR memory indirect indexed with accumulator	EOR (0FF),Y
EOR	#nn(L)	Exclusive OR the low-order byte of the immediate data and the accumulator	EOR #TAG(L)
EOR	#nn(H)	Exclusive OR the high-order byte of the immediate data and the accumulator	EOR #TAG(H)
EOR	OPER	Exclusive OR memory with accumulator	EOR TAG(A)
EOR	OPER,X	Exclusive OR memory with accumulator indexed by X	EOR TAG(A),X
EOR	OPER,Y	Exclusive OR memory with accumulator indexed by Y	EOR TAG(A),Y
EOR	ZPAGE	Exclusive OR memory with accumulator	EOR TAG(Z)
EOR	ZPAGE,X	Exclusive OR memory with accumulator indexed by X	EOR TAG(Z),X
INC	aa	Increment memory by one	INC 88
INC	aaaa	Increment memory by one	INC 986
INC	aa,x	Increment memory indexed by one	INC 35,X

INC	aaaa,x	Increment memory indexed by one	INC 458,X
INC	OPER	Increment memory by one	INC TAG(A)
INC	OPER,X	Increment memory by one indexed by X	INC TAG(A),X
INC	ZPAGE	Increment memory by one	INC TAG(Z)
INC	ZPAGE,X	Increment memory by one indexed by X	INC TAG(Z),X
INX		Increment index X by one	INX
INY		Increment index Y by one	INY
JMP	aaaa	Jump to new location	JMP 6
JMP	(aaaa)	Jump to new location indirect	JMP (TAG)
JMP	OPER	Jump to new location	JMP TAG(A)
JSR	aaaa	Jump to new location saving return address	JSR 101
JSR	OPER	Jump to new location saving return address	JSR TAG(A)
LDA	#ii	Load accumulator with immediate	LDA #55
LDA	aa	Load accumulator with memory	LDA 99
LDA	aaaa	Load accumulator with memory	LDA 105
LDA	aa,x	Load accumulator with memory indexed	LDA 0FF,X
LDA	aaaa,x	Load accumulator with memory indexed	LDA TAG,X
LDA	aaaa,y	Load accumulator with memory indexed	LDA 3FF,Y
LDA	(aa,x)	Load accumulator with memory indexed indirect	LDA (0DD,X)
LDA	(aa),y	Load accumulator with memory indirect indexed	LDA (55),Y

LDA	#nn(L)	Load the accumulator with the low-order byte of the immediate data	LDA #TAG(L)
LDA	#nn(H)	Load the accumulator with the high-order byte of the immediate data	LDA #TAG(H)
LDA	OPER	Load the accumulator with memory	LDA TAG(A)
LDA	OPER,X	Load the accumulator with memory indexed by X	LDA TAG(A),X
LDA	OPER,Y	Load the accumulator with memory indexed by Y	LDA TAG(A),Y
LDA	ZPAGE	Load the accumulator with memory	LDA TAG(Z)
LDA	ZPAGE,X	Load the accumulator with memory indexed by X	LDA TAG(Z),X
LDX	#ii	Load index X with immediate data	LDX #123.
LDX	aa	Load index X with memory	LDX 25
LDX	aaaa	Load index X with memory	LDX 100
LDX	aa,y	Load index X with memory indexed	LDX 0AB,Y
LDX	aaaa,y	Load index X with memory indexed	LDX 105,Y
LDX	#nn(L)	Load Index X with the low-order byte of the immediate data	LDX #TAG(L)
LDX	#nn(H)	Load Index X with the high-order byte of the immediate data	LDX #TAG(H)
LDX	OPER	Load Index X with memory	LDX TAG(A)
LDX	OPER,Y	Load Index X with memory indexed by Y	LDX TAG(A),Y
LDX	ZPAGE	Load Index X with memory	LDX TAG(Z)
LDX	ZPAGE,Y	Load Index X with memory indexed by Y	LDX TAG(Z),Y
LDY	#ii	Load index Y with immediate data	LDY #34

LDY	aa	Load index Y with memory	LDY 0FA
LDY	aaaa	Load index Y with memory	LDY 555
LDY	aa,x	Load index Y with memory indexed	LDY 2,X
LDY	aaaa,x	Load index Y with memory indexed	LDY 4F5,X
LDY	#nn(L)	Load Index Y with the low-order byte of the immediate data	LDY #TAG(L)
LDY	#nn(H)	Load Index Y with the high-order byte of the immediate data	LDY #TAG(H)
LDY	OPER	Load Index Y with memory	LDY TAG(A)
LDY	OPER,X	Load Index Y with memory indexed by X	LDY TAG(A),X
LDY	ZPAGE	Load Index Y with memory	LDY TAG(Z)
LDY	ZPAGE,X	Load Index Y with memory indexed by X	LDY TAG(Z),X
LSR	A	Shift right accumulator one bit	LSR A
LSR	aa	Shift right memory one bit	LSR 55
LSR	aaaa	Shift right memory one bit	LSR 375
LSR	aa,x	Shift right memory indexed one bit	LSR 41,X
LSR	aaaa,x	Shift right memory indexed one bit	LSR TAG,X
LSR	OPER	Shift right one bit	LSR TAG(A)
LSR	OPER,X	Shift right one bit indexed by X	LSR TAG(A),X
LSR	ZPAGE	Shift right one bit	LSR TAG(Z)
LSR	ZPAGE,X	Shift right one bit with memory indexed by X	LSR TAG(Z),X
NOP		No operation	NOP
ORA	#ii	OR immediate with accumulator	ORA #154.

ORA	aa	OR memory with accumulator	ORA 0F1
ORA	aaaa	OR memory with accumulator	ORA TAG1
ORA	aa,x	OR memory indexed with accumulator	ORA 56,X
ORA	aaaa,x	OR memory indexed with accumulator	ORA 678,X
ORA	aaaa,y	OR memory indexed with accumulator	ORA 34,Y
ORA	(aa,x)	OR memory indexed indirect with accumulator	ORA (45,X)
ORA	(aa),y	OR memory indirect indexed with accumulator	ORA (77),Y
ORA	#nn(L)	Logical OR the low-order byte of the immediate data with the accumulator	ORA #TAG(L)
ORA	#nn(H)	Logical OR the high-order byte of the immediate data with the accumulator	ORA #TAG(H)
ORA	OPER	Logical OR memory with accumulator	ORA TAG(A)
ORA	OPER,X	Logical OR memory with accumulator indexed by X	ORA TAG(A),X
ORA	OPER,Y	Logical OR memory with accumulator indexed by Y	ORA TAG(A),Y
ORA	ZPAGE	Logical OR memory with accumulator	ORA TAG(Z)
ORA	ZPAGE,X	Logical OR memory with accumulator indexed by X	ORA TAG(Z),X
PHA		Push accumulator on stack	PHA
PHP		Push processor status on stack	PHP
PLA		Pull accumulator from stack	PLA

PLP		Pull processor status from stack	PLP
ROL	A	Rotate left accumulator one bit	ROL A
ROL	aa	Rotate left memory one bit	ROL 23
ROL	aaaa	Rotate left memory one bit	ROL 0FFF
ROL	aa,x	Rotate left memory indexed one bit	ROL 34,X
ROL	aaaa,x	Rotate left memory indexed one bit	ROL TAG,X
ROL	OPER	Rotate one bit left	ROL TAG(A)
ROL	OPER,X	Rotate one bit left indexed by X	ROL TAG(A),X
ROL	ZPAGE	Rotate one bit left	ROL TAG(Z)
ROL	ZPAGE,X	Rotate one bit left indexed by X	ROL TAG(Z),X
ROR	A	Rotate right accumulator one bit	ROR A
ROR	aa	Rotate right memory one bit	ROR 63
ROR	aaaa	Rotate right memory one bit	ROR 1627
ROR	aa,x	Rotate right memory indexed one bit	ROR 3,X
ROR	aaaa,x	Rotate right memory indexed one bit	ROR 0CA4,X
ROR	OPER	Rotate one bit right	ROR TAG(A)
ROR	OPER,X	Rotate one bit right indexed by X	ROR TAG(A),X
ROR	ZPAGE	Rotate one bit right	ROR TAG(Z)
ROR	ZPAGE,X	Rotate one bit bit right indexed by X	ROR TAG(Z),X
RTI		Return from interrupt	RTI
RTS		Return from subroutine	RTS

SBC	#ii	Subtract immediate from accumulator with borrow	SBC #0FF
SBC	aa	Subtract memory from accumulator with borrow	SBC 65.
SBC	aaaa	Subtract memory from accumulator with borrow	SBC 250
SBC	aa,x	Subtract memory indexed from accumulator with borrow	SBC 0ED,X
SBC	aaaa,x	Subtract memory indexed from accumulator with borrow	SBC 1000,X
SBC	aaaa,y	Subtract memory indexed from accumulator with borrow	SBC 8219,Y
SBC	(aa,x)	Subtract memory indexed indirect from accumulator with borrow	SBC (88,X)
SBC	(aa),y	Subtract memory indirect indexed from accumulator with borrow	SBC (0DC),Y
SBC	#nn(L)	Subtract the low-order byte of the immediate data from the accumulator with borrow	SBC #TAG(L)
SBC	#nn(H)	Subtract the high-order byte of the immediate data from the accumulator with borrow	SBC #TAG(H)
SBC	OPER	Subtract memory from accumulator with borrow	SBC TAG(A)
SBC	OPER,X	Subtract memory from accumulator with borrow indexed by X	SBC TAG(A),X
SBC	OPER,Y	Subtract memory from accumulator with borrow indexed by Y	SBC TAG(A),Y
SBC	ZPAGE	Subtract memory from accumulator with borrow	SBC TAG(Z)
SBC	ZPAGE,X	Subtract memory from accumulator with borrow indexed by X	SBC TAG(Z),X
SEC		Set carry flag	SEC

SED		Set decimal mode	SED
SEI		Set interrupt disable status	SEI
STA	aa	Store accumulator in memory	STA 2
STA	aaaa	Store accumulator in memory	STA 258
STA	aa,x	Store accumulator in memory indexed	STA OBA,X
STA	aaaa,x	Store accumulator in memory indexed	STA TAG,X
STA	aaaa,y	Store accumulator in memory indexed	STA 56,Y
STA	(aa,x)	Store accumulator in memory indexed indirect	STA (65,X)
STA	(aa),y	Store accumulator in memory indirect indexed	STA (52),Y
STA	OPER	Store accumulator in memory	STA TAG(A)
STA	OPER,X	Store accumulator in memory indexed by X	STA TAG(A),X
STA	OPER,Y	Store accumulator in memory indexed by Y	STA TAG(A),Y
STA	ZPAGE	Store accumulator in memory	STA TAG(Z)
STA	ZPAGE,X	Store accumulator in memory indexed by X	STA TAG(Z),X
STX	aa	Store index X in memory	STX OFF
STX	aaaa	Store index X in memory	STX TAG1
STX	aa,y	Store index X in memory indexed	STX 79,Y
STX	OPER	Store index X in memory	STX TAG(A)
STX	ZPAGE	Store index X in memory	STX TAG(Z)
STX	ZPAGE,Y	Store index X in memory indexed by Y	STX TAG(Z),Y

STY	aa	Store index Y in memory	STY 45
STY	aaaa	Store index Y in memory	STY 376
STY	aa,x	Store index Y in memory indexed	STY 12,X
STY	OPER	Store index Y in memory	STY TAG(A)
STY	ZPAGE	Store index Y in memory	STY TAG(Z)
STY	ZPAGE,X	Store index Y in memory indexed by X	STY TAG(Z),X
TAX		Transfer accumulator to index X	TAX
TAY		Transfer accumulator to index Y	TAY
TSX		Transfer stack pointer to index X	TSX
TXA		Transfer index X to accumulator	TXA
TXS		Transfer index X to stack pointer	TXS
TYA		Transfer index Y to accumulator	TYA

5.5 SAMPLE 6500 ASSEMBLY LISTING

.MAIN.   X6500 V1.09 16-MAY-84

```
        0010                       .RADIX   16
   0000                            .ASECT
                             ;
                             ;   DIRECT ASSIGNMENT OF LABELS
                             ;
        0032                       PC=32
        003B                       SEMI=3B
        EC18                       DE1=0EC18
        EA84                       PACK=0EA84
        EB9E                       PHXY=0EB9E
        EBAC                       PLXY=0EBAC
        F2E1                       COL0=0F2E1
        F321                       COL1=0F321
        F361                       COL2=0F361
        F3A1                       COL3=0F3A1
        F3E1                       COL4=0F3E1
        A808                       T2L=0A808
        000C                       MOTON=0C
        000E                       MOTOFF=0E
        A000                       DRB=0A000
        A001                       DRA=0A001
        A002                       DDRB=0A002
        A003                       DDRA=0A003
        A004                       T1L=0A004
        A005                       T1CH=0A005
        A007                       T1H=0A007
        A00B                       ACR=0A00B
        A00C                       PCR=0A00C
        A00D                       IFR=0A00D
        0190                       .=190
   0190   00                       SAVA:  .BYTE
   0191   00                       EQFL:  .BYTE
   0192   00                       CRFL:  .BYTE
   0193   00                       PBPTR: .BYTE
   0194   00                       PBUF:  .BYTE
        0200                       .=200
                             ;
                             ;   ENTRY & INITIALIZATION
                             ;
   0200   08                 PRINT: PHP        ;SAVE PROCESSOR STATUS
   0201   78                        SEI        ;DIS. INTERRUPT DURING PRT
   0202   A9   D0                   LDA #0D0
   0204   8D   04   A0              STA T1L
```

```
0207   A9   0C                    LDA #MOTON
0209   8D   0C   A0               STA PCR      ;START MOTOR
020C   2C   00   A0 PR1:          BIT DRB      ;TEST LIMIT SWITCHES
020F   50   53                    BVC RMAR
0211   30   F9                    BMI PR1
                        ;
                        ;    LEFT TO RIGHT PRINT
                        ;
0213   20   CF   02 LMAR:         JSR DEBDEL ;DEBOUNCE DELAY
0216   A0   00                    LDY #0
0218   2C   00   A0 LM1:          BIT DRB
021B   10   FB                    BPL LM1      ;WAIT TO CLEAR MARGIN
021D   A9   01                    LDA #1
021F   8D   05   A0               STA T1CH     ;START DOT RIMER(200)
0222   B9   94   01 LM2:          LDA PBUF,Y ;LOAD WITH CHARACTER
0225   29   3F                    AND #3F
0227   AA                         TAX
0228   A9   20                    LDA #20
022A   99   94   01               STA PBUF,Y ;REPLACE WITH BLANK
022D   BD   E1   F2               LDA COL0,X
0230   20   A6   02               JSR OUTDOT ;OUTPUT COLUMN 0
0233   BD   21   F3               LDA COL1,X
0236   20   A6   02               JSR OUTDOT ;OUTPUT COLUMN 1
0239   BD   61   F3               LDA COL2,X
023C   20   A6   02               JSR OUTDOT ;OUTPUT COLUMN 2
023F   BD   A1   F3               LDA COL3,X
0242   20   A6   02               JSR OUTDOT ;OUTPUT COLUMN 3
0245   BD   E1   F3               LDA COL4,X
0248   20   A6   02               JSR OUTDOT ;OUTPUT COLUMN 4
024B   A9   00                    LDA #0 ;INSERT 1 SPACE BETWEEN CHAR
024D   20   A6   02               JSR OUTDOT
0250   C8                         INY
0251   C0   48                    CPY #72.     ;END OF LINE?
0253   90   CD                    BCC LM2 ;IF NOT, GET MORE CHARACTERS
                        ;
                        ;    EXIT ROUTINE
                        ;
0255   A9   FF          PRXIT:    LDA #0FF
0257   8D   08   A8               STA T2L
025A   20   18   EC               JSR DE1
025D   A9   0E                    LDA #MOTOFF
025F   8D   0C   A0               STA PCR      ;MOTOR OFF
0262   28                         PLP          ;RESTORE PROCESSOR STATUS
0263   60                         RTS
                        ;
                        ;    RIGHT TO LEFT PRINT
                        ;
0264   20   CF   02 RMAR:         JSR DEBDEL
```

```
0267    A0   47              LDY #71.    ;RIGHT BUFFER LIMIT
0269    2C   00   A0 RM1:    BIT DRB
026C    50   FB              BVC RM1
026E    A9   01              LDA #1
0270    8D   05   A0         STA T1CH
0273    B9   94   01 RM2:    LDA PBUF,Y
0276    29   3F              AND #3F
0278    AA                   TAX
0279    A9   20              LDA #20
027B    99   94   01         STA PBUF,Y
027E    BD   E1   F3         LDA COL4,X
0281    20   A6   02         JSR OUTDOT
0284    BD   A1   F3         LDA COL3,X
0287    20   A6   02         JSR OUTDOT
028A    BD   61   F3         LDA COL2,X
028D    20   A6   02         JSR OUTDOT
0290    BD   21   F3         LDA COL1,X
0293    20   A6   02         JSR OUTDOT
0296    BD   E1   F2         LDA COL0,X
0299    20   A6   02         JSR OUTDOT
029C    A9   00              LDA #0
029E    20   A6   02         JSR OUTDOT
02A1    88                   DEY
02A2    10   CF              BPL RM2
02A4    30   AF              BMI PRXIT
                     ;
                     ;   HERE TO OUTPUT 1 COLUMN OF DOTS
                     ;
02A6    49   FF      OUTDOT: EOR #0FF    ;INVERT FOR OUTPUT
02A8    2C   0D   A0 OD1:    BIT IFR
02AB    50   FB              BVC OD1 ;WAIT FOR INTER-DOT TIMEOUT
02AD    8D   01   A0         STA DRA     ;OUTPUT DOTS
02B0    A9   05              LDA #5
02B2    8D   07   A0         STA T1H     ;LOAD INTER-DOT TIME
02B5    A9   86              LDA #86
02B7    8D   04   A0         STA T1L
02BA    A9   FF              LDA #0FF
02BC    2C   0D   A0 OD2:    BIT IFR
02BF    50   FB              BVC OD2     ;WAIT FOR DOT TIMEOUT
02C1    8D   01   A0         STA DRA     ;OFF
02C4    A9   01              LDA #1
02C6    8D   07   A0         STA T1H
02C9    A9   D0              LDA #0D0
02CB    8D   04   A0         STA T1L
02CE    60                   RTS
                     ;
                     ;   DELAY ROUTINE
                     ;
```

```
02CF   A9  10       DEBDEL: LDA #10     ;DEBOUNCE DELAY
02D1   8D  08  A8           STA T2L
02D4   A9  27               LDA #27
02D6   4C  18  EC           JMP DE1
                    ;
                    ;   INITIALIZATION ROUTINE
                    ;
02D9   A9  47       DRI:    LDA #71.
02DB   A9  20               LDA #20
02DD   9D  94  01 DRI1:     STA PBUF,X ;CLEAR BUFFER
02E0   CA                   DEX
02E1   10  FA               BPL DRI1
02E3   A9  00               LDA #0
02E5   8D  93  01           STA PBPTR
02E8   8D  92  01           STA CRFL
02EB   8D  91  01           STA EQFL
02EE   8E  01  A0           STX DRA
02F1   8E  03  A0           STX DDRA
02F4   A9  40               LDA #40
02F6   8D  0B  A0           STA ACR     ;T1 FREE RUN
02F9   60                   RTS
                    ;
                    ;   DRIVER ROUTINE
                    ;
02FA   90  DD       DRIVER: BCC DRI     ;CHECK FOR INITIALIZATION
02FC   68                   PLA         ;GET CHAR TO BE PRINTED
02FD   20  9E  EB           JSR PHXY
0300   8D  90  01           STA SAVA
0303   29  7F               AND #7F
0305   C9  0D               CMP #0D     ;CARRIAGE RETURN?
0307   D0  0E               BNE DR1
0309   0E  92  01           ASL CRFL    ;YES
030C   90  03               BCC CR1     ;FLAG SET?
030E   20  7A  03           JSR PLINE   ;YES,PRINT LINE
0311   38           CR1:    SEC         ;SET CARRY FLAG
0312   6E  92  01           ROR CRFL    ;SET CARRIAGE RETURN FLAG
0315   D0  36               BNE DRXIT
0317   C9  3D       DR1:    CMP #3D     ;IS THERE AN " = "?
0319   D0  1A               BNE DR3
031B   0E  92  01           ASL CRFL    ;YES
031E   90  0E               BCC DR2
0320   20  00  02           JSR PRINT   ;PRINT LINE
0323   A9  00               LDA #0
0325   8D  93  01           STA PBPTR   ;ZERO BUFFER POINTER
0328   38                   SEC
0329   6E  91  01           ROR EQFL    ;SET EQUAL FLAG
032C   D0  1F               BNE DRXIT
032E   0E  91  01 DR2:      ASL EQFL    ;CRFL NOT SET, TEST EQFL
```

```
0331    90    35              BCC STUFF   ;PUT "=" IN BUFFER IF 1ST
0333    B0    18              BCS DRXIT   ;IGNORE IF SECOND
0335    C5    3B       DR3:   CMP SEMI    ;SEMICOLON?
0337    D0    1B              BNE DR5
0339    0E    92    01        ASL CRFL    ;YES
033C    AE    93    01        LDX PBPTR
033F    E0    0C              CPX #12.    ;START OF LINE?
0341    F0    25              BEQ STUFF
0343    A2    1E       DR4:   LDX #30.    ;NO
0345    EC    93    01        CPX PBPTR   ;TAB TO COLUMN 30
0348    90    03              BCC DRXIT
034A    8E    93    01        STX PBPTR
034D    20    AC    EB DRXIT: JSR PLXY
0350    AD    90    01        LDA SAVA
0353    60                    RTS
0354    0E    92    01 DR5:   ASL CRFL    ;NOT CARRIAGE RETURN, = or ;
0357    90    0F              BCC STUFF   ;LOAD
0359    A2    0C              LDX #12.
035B    EC    93    01        CPX PBPTR   ;CHECK FOR BEYOND COLUMN 12
035E    90    05              BCC DR6
0360    8E    93    01        STX PBPTR   ;TAB TO COLUMN 12
0363    B0    03              BCS STUFF   ;LOAD
0365    20    7A    03 DR6:   JSR PLINE   ;PRINT LINE
0368    AD    90    01 STUFF: LDA SAVA    ;GET CHARACTER
036B    AE    93    01        LDX PBPTR   ;GET BUFFER POINTER
036E    E0    48              CPX #72.    ;CHECK FOR FULL
0370    B0    DB              BCS DRXIT
0372    9D    94    01        STA PBUF,X  ;NO, PUT CHAR. IN BUFFER
0375    EE    93    01        INC PBPTR   ;INCREMENT FOR ANOTHER
0378    D0    D3              BNE DRXIT
037A    20    00    02 PLINE: JSR PRINT
037D    A2    00              LDX #0
037F    A5    33              LDA PC+1    ;PC UPPER
0381    20    8F    03        JSR CONVT
0384    A5    32              LDA PC      ;PC LOWER
0386    20    8F    03        JSR CONVT
0389    A9    0C              LDA #12.
038B    8E    93    01        STX PBPTR   ;SET COLUMN POINTER
038E    60                    RTS
                       ;
                       ;   HEX TO ASCII CONVERSION AND LOAD
038F    48             CONVT: PHA
0390    4A                    LSR A
0391    4A                    LSR A
0392    4A                    LSR A
0393    4A                    LSR A
0394    20    9A    03        JSR CONV
0397    68                    PLA
```

```
0398    29   0F                  AND #0F
039A    18           CONV:       CLC          ;CLEAR CARRY FLAG
039B    69   30                  ADC #30
039D    C9   3A                  CMP #3A
039F    90   02                  BCC CONV1
03A1    69   06                  ADC #6
03A3    9D   94   01 CONV1:      STA PBUF,X
03A6    E8                       INX
03A7    60                       RTS
      0001                       .END
```

```
.MAIN.   X6500  V1.09 16-MAY-84
SYMBOL TABLE

ACR   = A00B   DDRB   = A002   DR4   0343   OUTDOT 02A6   PR1   020C
COL0  = F2E1   DEBDEL   02CF   DR5   0354   PACK = EA84   RMAR  0264
COL1  = F321   DE1    = EC18   DR6   0365   PBPTR  0193   RM1   0269
COL2  = F361   DRA    = A001   EQFL  0191   PBUF   0194   RM2   0273
COL3  = F3A1   DRB    = A000   IFR=  A00D   PC   = 0032   SAVA  0190
COL4  = F3E1   DRI      02D9   LMAR  0213   PCR  = A00C   SEMI= 003B
CONV    039A   DRIVER   02FA   LM1   0218   PHXY = EB9E   STUFF 0368
CONVT   038F   DRI1     02DD   LM2   0222   PLINE  037A   T1CH= A005
CONV1   03A3   DRXIT    034D   MOTOFF=000E  PLXY = EBAC   T1H = A007
CRFL    0192   DR1      0317   MOTON=000C   PRINT  0200   T1L = A004
CR1     0311   DR2      032E   OD1   02A8   PRXIT  0255   T2L = A808
DDRA  = A003   DR3      0335   OD2   02BC

. ABS.     03A8        00
           0000        01
ERRORS DETECTED:  0

VIRTUAL MEMORY USED:  288 WORDS  ( 2 PAGES)
DYNAMIC MEMORY AVAILABLE FOR  74 PAGES
,DY1:TST65=DY1:TST65
```