W I C A T

Macro Assembler (WiMAC)

User's Guide and Reference Manual

188-377-301 A

**** PRELIMINARY - For internal use only ****

January 1985

WICAT SYSTEMS INCORPORATED
Orem, Utah

**** PRELIMINARY - For internal use only ****

COPYRIGHT STATEMENT

WIMAC is not a stand-alone product and consequently is not supported as such by WICAT Systems. This manual is provided to aid those who need to interface WIMAC with a programming language. WICAT assumes no responsibility for the use of WIMAC (as an interface or any other way) since it is not supported as a stand-alone product.

The software described in this manual is provided in accordance with a license agreement and may be used or copied only as indicated under the terms of such license.

WICAT Systems Incorporated assumes no responsibility for the use or reliability of software on equipment not supplied by WICAT.

MANUAL INTENT STATEMENT

The purpose of this manual is to help users in program development. The information contained in the manual is subject to change without notice and should not be construed as a commitment by WICAT Systems Incorporated. WICAT Systems Incorporated assumes no responsibility for any errors that may appear in this manual.

**** PRELIMINARY - For internal use only ****

WiMAC Version: 1.0

**** NOTICE ****

This manual describes the WICAT Macro Assembler version 1.0 and later. Version 1.0 of WiMAC supports UniPlus+ System V COFF and LL object formats. This version does not support macros or the WMCS command line format. WiMAC runs under the following operating systems:

WMCS	5.0
Uniplus+ System V	1.0

- Appendix A lists the ASCII character set that can be used in assembler programs.

- Appendix B lists error messages produced by the assembler. An explanation and probable cause for each error is also given.

- Appendix C

- Appendix D order.

- Appendix E lists the 68000 instruction set in alphabetical order.

- Appendix F

MANUAL CONVENTIONS

A description of the symbolic conventions used throughout this manual follows. Familiarize yourself with these conventions before you continue to read.

The following conventions are observed:

- Examples consist of actual assembler programs or program fragments wherever possible.

- Uppercase words and letters, when used in examples, indicate that the word or letter must be typed exactly as shown.

- Lowercase words and letter, when used in format examples, indicate that you are to substitute a word or value of your choice.

- Square brackets ([]) indicate that the enclosed item(s) is(are) optional. The square brackets are <u>not</u> entered as part of any option, they are shown only to aid in the description of the syntax.

- Braces ({}) indicates that the enclosed item(s) can be repeated zero or more times as a group. The braces are <u>not</u> entered as part of the repetitions, they are shown only to aid in the description of the syntax.

- The angle brackets (<>) indicate that the item enclosed must be supplied by the user. For items that require numeric values, the values are interpreted as decimal, unless otherwise stated or modified. The angle brackets are <u>not</u> entered as part of item, they are shown only to aid

Figures

FIGURE

SUMMARY OF TECHNICAL CHANGES

This manual documents WiMAC version V1.0. The following technical changes are new to this release:

CHANGES

None - A new release.

ENHANCEMENTS

None - A new release.

FIXED BUGS

None - A new release.

PREFACE

MANUAL OBJECTIVES

The intent of this publication is to provide sufficient information to develop assembly language programs on WICAT computer systems. The information contained in this manual pertains to the usage and syntax of the assembler only.

MANUAL STRUCTURE

This manual is organized into ten chapters and seven appendices, as follows:

- Chapter 1 introduces the features of the WICAT macro assembler.
- Chapter 2
- Chapter 3 explains how to use the assembler on the appropriate operating system (WMCS, UniPlus+ System V).
- Chapter 4 describes the listing file produced by the assembler.
- Chapter 5 covers the format used in the assembler source statements.
- Chapter 6 describes the components of a assembler source statements: the character set; symbols; numbers; and expressions.
- Chapter 7 explains the general directives (pseudo-opcodes). Pseudo-opcodes discussed in this chapter include listing control, symbol control, data definition and storage, and program sectioning.
- Chapter 8
- Chapter 9

in the description of the syntax.

CHAPTER 1

INTRODUCTION

[To be written later]

CHAPTER 2

FEATURES

[To be written later]

CHAPTER 3

INVOKING THE ASSEMBLER

3.1 WMCS OPERATING SYSTEM

****** NOTE ******

At the time of this writing, WiMAC is used only to assemble compiler generated programs. It is not intended to be used directly by the user. Therefore, all invocation of WiMAC should be done by the COMPILE utility. Refer to the Wicat Multi-user Control System (WMCS) User's Reference Manual for complete documentation on the use of COMPILE.

3.2 UNIPLUS+ SYSTEM V OPERATING SYSTEM

****** NOTE ******

At the time of this writing, WiMAC is used only to assemble compiler generated programs. It is not intended to be used directly by the user. Therefore, all invocation of WiMAC should be done by the CC(1) utility. Refer to the Wicat UniPlus+ System V User's Manual (Section 1) for complete documentation on the use of CC(1).

3.3 DIAGNOSTIC MESSAGES

CHAPTER 4

LISTING FILE

The listing file produced by the assembler can consist of the following five parts:

- o Assembly source statements
- o Symbol table (optional)
- o Program section tables (optional)
- o Cross-reference table (optional)
- o Assembly summary (optional)

Sections 4.1 through 4.5 describe each of these parts. Section 4.6 contains an example of a listing file.

4.1 ASSEMBLY SOURCE STATEMENTS

The assembly source statements comprise the main part of the listing file, and consists of:

- o Page Headers
- o Source lines with hexadecimal code
- o Error and informational messages (if applicable)

Each is described below.

4.1.1 Page Header

The assembler prints a new page in the listing file when it encounters a .PAGE directive in the source, when it encounters a new page (form feed) in the source file, or when the existing page of the listing file is filled. On the top of each page in the listing file, the assembler prints five header lines.

The first line of the header contains the following information:

- Assembler name
- Assembler configuration
- Assembler version number
- Date the listing file was generated
- Time the listing file was generated
- Listing page number

The assembler configuration string consists of three fields. The first field indentifies the host operating system. Typical values are UNIX, WMCS, etc.. The second field lists the input format. The final field shows the output format. The currently defined output formats are LL and COFF.

The second line of the header contains the following information:

- o Wicat proprietary statement (if applicable) (See .LIST WICAT)
- o Source file name.

The third line of the header contains a user-supplied message. If no message has been supplied, this line is left blank. See the .HEADER directive for more information.

The fourth line is blank.

The fifth line contains the source-line column headers.

4.1.2 Source Statements With Hexadecimal Code

This section is the main part of the listing; it contains the source lines and the binary code generated for each line.

The hexadecimal code is printed with the lowest address on the left. The code listed for an instruction contains, from left to right:

- o The opcode
- o The first operand (if applicable)
- o The second operand (if applicable)
- o The third operand (if applicable)

The binary code for data storage is listed from left to right. The number of data items that are listed on one line depends on the size of the data type as shown in table 4-1.

Table 4-1: Data Types per Line

Data Type	Number of Items per Line
Byte	8
Word	5
Long	2
Characters	8
Quadword	1 (double precision real)

Continuation lines will be added as necessary.

If an expression contains an externally-defined symbol, the assembler evaluates the expression by assigning a value of zero to that symbol.

Table 4-2 summaries the source line listing format.

Table 4-2: Source Line Listing Format

Column	Header	Description
1-5	Line	Source line number (decimal)
7-14	Address	Location counter (hexadecimal)
16-19	Opcd	Opcode (hexadecimal)
21-39	Operands	Operands (hexadecimal)
41-	Source Statement	Source line

4.1.3 Error And Informational Messages

4.2 SYMBOL TABLE

The symbol table lists all symbols, except permanent symbols, that are defined or referenced in the module. The symbols are listed by order of appearance in a module. Each new level of nested symbols is indented two spaces.

4.3 PROGRAM SECTION TABLES

The program section tables lists the program sections, their names, their size, and their attributes. This information is presented in two tables, the first table lists all named program sections. This list is in the order in which they were defined. This tables also includes any predefined section names. The second table consists, all defined sections listed in numeric order, followed by their size and attributes. See the .SECT directive for a complete description of the attributes. All section numbers and sizes are listed both in decimal and hexadecimal radix. Decimal numbers always appear with a decimal point (.) after the number.

4.4 CROSS-REFERENCE TABLE

[Not implemented in this version]

4.5 ASSEMBLY SUMMARY

[Not implemented in this version]

4.6 EXAMPLE LISTING

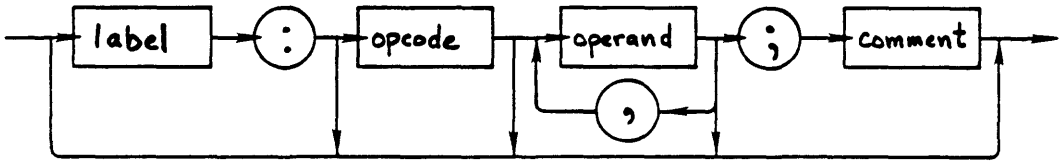
This section shows a complete listing file (figure 4-1) generated by assembling the source program listed in figure 4-1.

CHAPTER 5

SOURCE STATEMENT FORMAT

An assembly source program consists of a sequence of source statements, each of which occupies exactly one line. Multiple statements on a single line are not allowed. Each line can be up to 254 characters long (not including the line terminator). However, no line should exceed 80 characters to ensure that the source fits on one line in the listing file.

A source line consist of four basic fields: label, opcode, operand and comment. The general format of an assembler line is:



Spaces and tabs are allowed anywhere in the line, except inside labels, opcodes, symbols and numbers. At least one space or tab must appear between opcode and operand fields. Blank lines are accepted, but have no significance or meaning. All characters that have an ASCII value of space or less are treated as a space, with the exception of line feed (^J) and form feed (^L).

5.1 LABEL FIELD

A label is a user-defined symbol that references a specific location within a program. This symbol is assigned the value equal to the current location counter. The value of symbol may be absolute or relative depending on the type of section that it is defined in.

A label is a symbol that can contain uppercase letters (A-Z), lowercase letters (a-z), digits (0-9), underline (_), dollar sign ($) and period (.) characters. A label cannot start with a number or a dollar sign. A label must be terminated with a colon (:) and conform to all the rules that govern user-defined symbols (see section 6.2.2). This field is optional.

Once a label is defined within a module or a begin/end block, it cannot be redefined in that module or block. If a label is defined more that once, the assembler displays an error message where the label was defined and again where it was redefined. All labels are local to the module that they are defined in, unless they are exported out of the module with the .GLOBAL directive.

Only one label per source line is allowed. However, multiply labels may have the same value. For example:

```
label1:
label2:
label3:  nop
```

all have the same value (which is the address of the "nop" instruction).

All labels that apply to directives (see chapter 7), must be on the same line as the directive. For example:

```
foo:    .const  10
```

and

```
foo:
        .const  10
```

are not equivalent. The second ".const" declaration will result in an error (Label required).

5.2 OPCODE FIELD

The opcode field specifies the action to be performed by the line. This field may contain either an instruction mnemonic, an assembler directive, or a macro call. Assembler directives are indicated by beginning with a period (.). This style enables the user to quickly identify assembler directives from actual machine instructions. See Appendix E for a complete list of instructions mnemonics. Chapter 7 describes the assembler directives and macro calls.

5.3 OPERAND FIELD

The operand field contains operands for the instruction or arguments for an assembler directive or macro call. The operand field must be separated from the opcode field by a least one delimiter. A delimiter is typically a space or tab, however, any character with an ASCII value of space (hexadecimal 40) or less (except line feed (^J) and form feed (^L)) is considered a delimiter. When two or more operand appear within a statement, they must be separated by a comma (,).

5.4 COMMENT FIELD

The comment field contains text that describes the function of the line. This field must start with a semicolon (;) and be terminated by the end-of-line character. Comments can start anywhere on a line, including column 1. The comment field may contain any printable ASCII character (see Appendix A). Comments are included in the assembly listing but, otherwise, are ignored by the assembler.

CHAPTER 6

SOURCE STATEMENT COMPONENTS

This chapter describes the various components of the assembler source statement. These components consist of characters, symbols, numbers, strings, and expressions.

6.1 CHARACTER SET

The characters that can be used in assembler source statements are listed in Table 6-1. All control characters and DEL are treated as delimiters (spaces), except line feed (^J) and form feed (^L). The null character (^@) should be avoided, for it causes premature intermination of listing source lines.

Table 6-1: Legal Assembler Characters

Character	Character Name	Function
^J	Line feed	Line terminator
^L	Form feed	Page advance
	Space	Opcode/Operand field delimiter
!	Exclamation point	Logical NOT operator
"	Double quote mark	String indicator and terminator
#	Number sign	Immediate data indicator
$	Dollar sign	Hexadecimal radix indicator and character in symbol name
%	Percent sign	Remainder operator (modulus)

Table 6-1 (Cont.): Legal Assembler Characters

Character	Character Name	Function
&	Ampersand	Bitwise AND operator
'	Single Quote Mark	Character literal indicator and terminator
(	Left Parenthesis	Expression grouping delimiter and register indirection indicator
)	Right Parenthesis	Expression grouping delimiter and register indirection indicator
*	Asterisk	Arithmetic multiplication and current location counter
+	Plus	Autoincrement, unary plus, and arithmetic addition
,	Comma	Operand and parameter separator
-	Minus	Autodecrement, unary minus, arithmetic subtraction, and register range (MOVEM instr)
.	Period	Character in symbol name and real number decimal point
/	Slash	Arithmetic division and register separator (MOVEM instr)
0..9	Digits	Numbers and characters in symbol names
:	Colon	Label terminator and expression qualifier
;	Semicolon	Comment field indicator
<	Left-angle bracket	Less than operator
=	Equal sign	Equals operator
>	Right-angle bracket	Greater than operator
?	Question mark	Defined operator

Table 6-1 (Cont.): Legal Assembler Characters

Character	Character Name	Function
@	At sign	Reserved for future use
A..Z	Uppercase letters	Characters in symbol names
[	Right-square bracket	Reserved for future use
\	Backslash	Escape character indicator
]	Left-square bracket	Reserved for future use
^	Circumflex	Bitwise XOR operator
_	Underline	Character in symbol name
`	Reverse Apostrophe	Unary operator delimiter
a..z	Lowercase letters	Characters in symbol names
{	Left brace	Reserved for future use
|	Vertical bar	Bitwise OR operator (exclusive)
}	Right brace	Reserved for future use
~	Tilde	1's complement operator

6.2 SYMBOLS

Two types of symbols can be used in assembly programs: permanent symbols and user-defined symbols. Each is described below.

6.2.1 Permanent Symbols

Permanent symbols consist of specific processor instruction mnemonics (see Appendix E), assembler directives (see Chapter 7) and register names (see Table 6-2). These symbols need not be defined before being used. Instruction mnemonics and assembler directives are reserved symbol names and cannot be redefined by the user.

All permanent symbols are converted internally to uppercase. For example:

```
move    d0,dl
MOVE    D0,Dl
Move    d0,Dl
```

are all the same instruction and registers.

The registers of the 68000 microprocessor must be referenced as described in Table 6-2. Lowercase register names are allowed, but are mapped internally into uppercase register names.

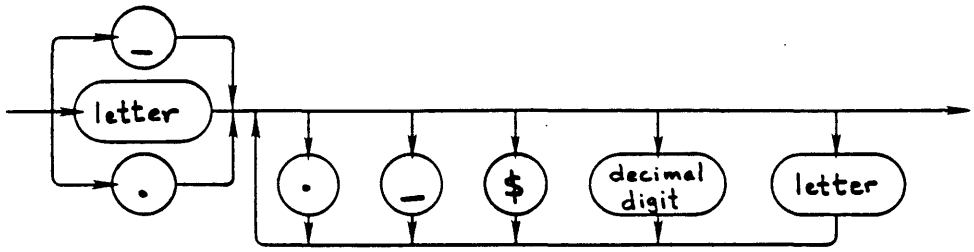
Table 6-2: Assembler Register Names

Register Name	68000 Register
D0-D7	Data registers
A0-A7	Address registers
SP	Stack Pointer registers (A7)
SSP	Supervisor Stack Pointer register (A7)
USP	User Stack Pointer register
PC	Program Counter
CCR	Condition Code Register
SR	Status Register

A complete description of these registers may be found in the Motorola MC68000 16-Bit Microprocessor User's Manual.

6.2.2 User-Defined Symbols

A user-defined symbol is a string of alphanumeric characters. The general format for a user-defined symbol is:



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

1. User-defined symbols can be composed of uppercase letters (A-Z), lowercase letters (a-z), digits (0-9), underlines (_), dollars ($) and periods (.).

2. The first character of a symbol must begin with a letter (A-Z,a-z), underline (_), or a period (.). It cannot begin with a number or a dollar sign.

3. No embedded spaces or other characters are allowed in a symbol.

4. There is no limit to the length of a symbol, however, the input source line is limited to 254 characters, thereby indirectly limiting the length of a symbol to 254 characters.

User-defined symbols can be used as labels, variables, module names, section names, and macro names. These user-defined symbols can also be equated to a specific value by the .ABSADR, .ADDR or .CONST directives (see Chapter 7) and used in any expressions (see Section 6.6).

Symbols can have absolute (constant) or relative values.

6.2.3 Name Spaces

The assembler supports several different name spaces. Name conflicts only occur in the same name space. This is to say, identical symbol names can coexist in different name spaces without conflict. The following name spaces are supported:

- Data Structure Names

- Module and Block Names
- Section Names
- Label and Variable Names
- Macro Names

6.2.4 Case Conversion

Case conversion of user-defined symbols is controlled by the ".ENABLE UPPER", ".ENABLE LOWER" and ".ENABLE MIXED" directives. When the ".ENABLE UPPER" directive is encountered, all subsequent user-defined symbols are converted to uppercase. The ".ENABLE LOWER" directive cause all subsequent user-defined symbols to be converted to lowercase. When the ".ENABLE MIXED" directive is encounter, no case conversion is performed. The default is ".ENABLE MIXED". Table 6-3 summarizes the effect of the various case conversion directives.

Table 6-3: Case Conversion Directives

Directive	Description
.ENABLE LOWER	Convert all subsequent user-defined symbols to lowercase.
.DISABL LOWER	Return to MIXED mode.
.ENABLE UPPER	Convert all subsequent user-defined symbols to uppercase.
.DISABL UPPER	Return to MIXED mode.
.ENABLE MIXED	No case conversion of subsequent user-defined symbols is performed.
.DISABL MIXED	Ignored.

Case conversion only applies to user-defined symbols. Permanent symbols are always mapped to uppercase. See chapter 7 for more information on the ".ENABLE" directive.

6.2.5 Determining Symbol Values

The value of a symbol depends on how it was defined or used in the assembly program.

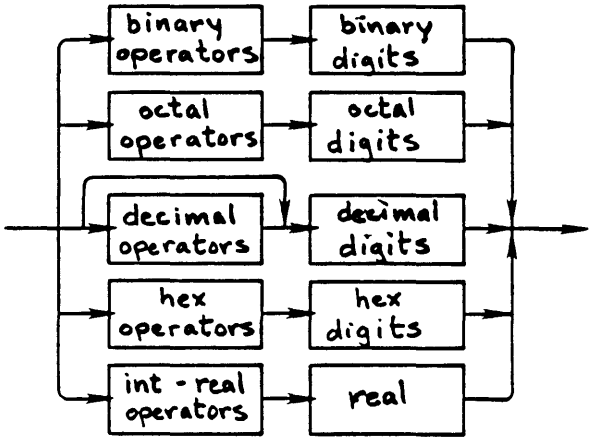
6.3 NUMBERS

Number can be integers, character literals, or reals. Integers and character literals are treated identically, and interpreted as integer numbers. All three types of numbers are described below.

Numbers are always treated as absolute (constant) values.

6.3.1 Integers

The general format for integer numbers is:



Integers must be in the range of -2,147,483,648 to +2,147,483,647 for signed numbers or in the range of 0 to +4,294,967,295 for unsigned numbers.

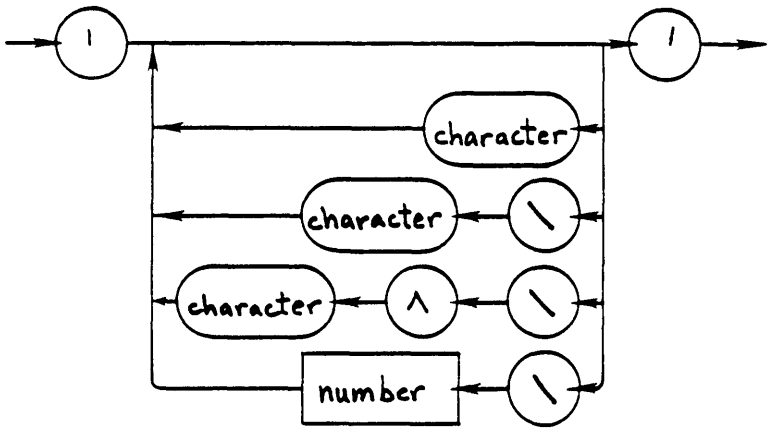
The assembler translates all negative numbers into 2's complement form. Negative numbers must be preceded by a minus sign. For positive numbers, the plus sign is optional.

The assembler interprets all integers in the source program as decimal unless the number is preceded by a radix control operator. See section 6.6.1.1 for a description of the radix operators.

Integers can be used in expressions or as a single values.

6.3.2 Character Literals

Character literals are a string of up to four characters (bytes) that are enclosed by single quote marks ('). The general format for character literals is:



Character literals of more than four characters are illegal and are reported as errors. The high order bit of each character in the literal is cleared (parity bit set to zero). This done to prevent sign extension of characters that are moved into registers.

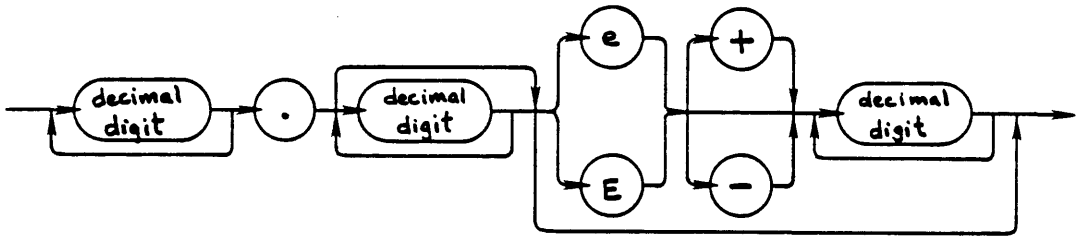
All character literals are considered internally as integer numbers.

Character literals can be used any where an integer number is allowed.

Character literals are always considered as absolute (constant) values.

6.3.3 Reals

The general format for a real number is:



The decimal point can appear anywhere to the right of the first digit. However, a real number cannot start with a decimal point. A real number can be specified with or without an exponent.

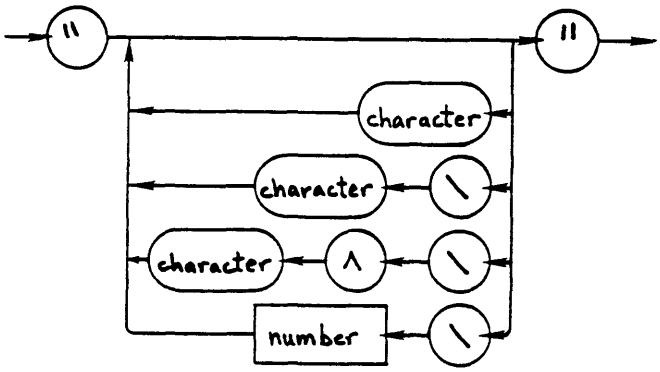
Real numbers can be single-precision (32-bit) or double precision (64-bit). The precision of single precision numbers is 6-7 digits and 15-16 digits for double precision.

The assembler converts all real numbers into standard IEEE format. A complete description of the internal real format can be found in Appendix D.

Real numbers can only be used in the .DC.F and .DC.D directives. Real numbers cannot be used in expressions or with any unary or binary operators, with the exception of unary minus, unary plus and and the unary operators `R, `T, `F, `U and `L.

6.4 STRINGS

Character strings are a string of up to 254 characters (bytes) that are enclosed by double quote marks ("). The general format for character strings is:



Any ASCII character except the line feed and double quote mark characters can appear directly within the string. All characters in the string are converted to their 8-bit ASCII value, and the high order bit is always cleared (parity bit set to zero). It is not recommend that non-printable (control) character be typed directly into the character string. Any character, including null, line feed, double quote and control characters, can be included in a character string, by "escaping" the character. An "escaped" character is introduced by the backslash character (\). The character immediately following the backslash is included in the string. Any of the radix control operators and floating point operator that return an integer can follow the backslash character. In this case, the character whose value is represented by the number is inserted into the character string. All numbers represented in this way are truncated to an 8-bit value. Finally, if the character immediately following the backslash is the circumflex character (^), then the next character is treated as a control character. This is the recommend way of inserting control or non-printing character into a character string.

The assembler performs no case conversion on strings. The assembler does not automatically insert any character at the end of the string.

A null string is represented by two consecutive double quote marks ("") and has a length of zero.

6.5 LOCATION COUNTER

The current location counter always has the value of the address of the current byte. The assembler symbol for the location counter is the asterisk (*). The assembler sets the current location counter at the beginning of each new program section (see .SECT - Chapter 7). The location counter may be set or changed by use of the following directives:

1. .MODULE <name>

2. .SECT <section>

3. .SECT <section> ,ADDRESS = <expression>

4. .ALIGN <keyword> [,<fill>]

5. .ALIGN <expression> [,FILL>]

6. .DS.x <expression> [,<fill>]

When the current location counter is used in the operand field of an instruction, the current location counter has the value of the address of the beginning of the instruction — it does not have the value of the address of the operand.

Asterisk has an absolute value if used in a absolute section, otherwise it has a relative value.

6.6 GENERAL EXPRESSIONS

Expressions consist of constants, absolute symbols, relative symbols, external symbols, functions, and operators. Constants and absolute symbols can be used with any of the operators and have no limitations on their usage in expressions. Relative and external symbols can only be used with the addition and subtraction operators in simple expression. Section 6.6.3 descibes where these symbols are legal. The legal operators are fully described in section 6.6.1.

The assembler evaluates expressions from left to right with the operator precedence rules described in Table 6-11. However, parentheses () can be used to change the order of evaluation. Any portion of an expression that is enclosed in parentheses is first evaluated to a single value, which is then used in evaluating the complete expression.

All expressions are evaluated as signed 32-bit values. The result of any expression with an error is zero.

6.6.1 Operators

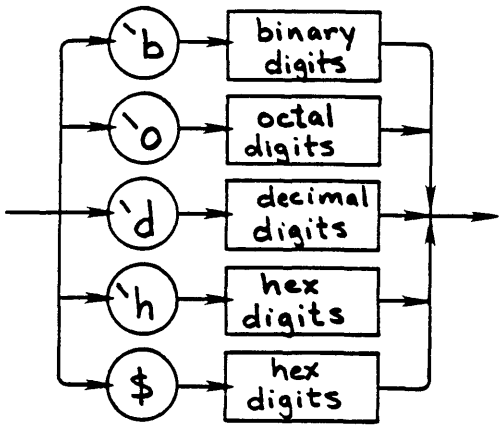
Operators perform a specific function on an expression. All operators accept only interger numbers or character literals as operands, with the exceptions of the real number operators (`R, `T, `F, `U, and `L), which accept floating-point number as operands. The result of all operators is a 32-bit signed integer number.

The assembler operators are broken up into the following six categories:

1. Radix Control Operators
2. Real Number Operators
3. Arithmetic Operators
4. Bitwise Operators
5. Logical Operators
6. Relational Operators

Each is described below.

6.6.1.1 Radix Control Operators - The assembler accepts numbers in four different radixes: binary, octal, decimal, and hexadecimal. The default radix is decimal. The general format for the radix control operators is:



For compatibility with previous assemblers, the dollar sign ($) can be used to specified the hexadecimal radix. The legal characters for each radix are listed below.

Table 6-4: Legal Radix Characters

Format	Radix	Legal Characters
`B	Binary	0 and 1
`O	Octal	0 through 7
`D	Decimal	0 through 9
`H	Hexadecimal	0 through 9, A through F, and a through f
$	Hexadecimal	0 through 9, A through F, and a through f

Radix control operators can be included in the source program anywhere a numeric value is legal. A radix control operator affects only the number immediately following it.

The reverse apostrophe (`) cannot be separated from the B, O, D, and H character that follows it, but the radix operator can be separated by spaces or tabs from the number that follows it.

Table 6-5 summarizes the radix control operators.

Table 6-5: Radix Control Operators

Operator	Operator Name	Operation
`B	Binary	Binary value
`O	Octal	Octal value
`D	Decimal	Decimal value
`H	Hexadecimal	Hexadecimal value
$	Dollar sign	Hexadecimal value

6.6.1.2 Real Number Operators - Real number operators accept real number arguments and return an integer number. The real number operators are useful because it allows real numbers to be used in expressions and instructions that accept only integers. See section 6.3.3 for the format of real numbers.

Real number operators can be included in the source program anywhere a numeric value is legal. A real number operator affects only the number immediately following it.

The reverse apostrophe (`) cannot be separated from the R, T, F, U, and L character that follows it, but the real operator can be separated by spaces or tabs from the real number that follows it.

Table 6-6 summarizes the real number operators.

Table 6-6: Real Number Operators

Operator	Operator Name	Operation
`R	Round	Round real number to the nearest integer.
`T	Truncate	Truncate real number to its integer part.
`F	Coerce	Change apparent type of a single-precision (32-bit) real number to a long integer (32-bit).
`L	Double Lower	Extract as an integer the lower 32-bits of a double-precision (64-bit) real number.
`U	Double Upper	Extract as an integer the upper 32-bits of a double-precision (64-bit) real number.

6.6.1.3 Arithmetic Operators - The arithmetic operators perform the usual arithmetic conversion on their operands.

The assembler prints a warning message if division by zero occurs.

Table 6-7 summarizes the arithmetic operators.

Table 6-7: Arithmetic Operators

Operator	Operator Name	Operation
+	Plus sign	Positive (unary)
-	Minus sign	Negative (unary)
+	Plus sign	Addition
-	Minus sign	Subtraction
*	Asterisk	Multiplication
/	Slash	Division
%	Per Cent	Remainder

6.6.1.4 Bitwise Operators - The bitwise operators perform the usual bitwise conversion on their operands.

The shift operators are used to perform left and right arithmetic shifts. The first operand is shifted left or right by the number of bit positions specified in the second operand. When the first operand is shifted left, the low-order bits are set to zero. When the right shift operator is used and the first operand is signed, the high-order bits are set to the value of the orignal high-order bit (sign bit) (arithmetic shift). When the right shift operator is used and the first operand is unsigned the high-order bits are set to zero (logical shift).

Table 6-8 summarizes the bitwise operators.

Table 6-8: Bitwise Operators

Operator	Operator Name	Operation
~	Tilde	1's complement value
&	Ampersand	Bitwise AND
|	Exclamation	Bitwise OR (inclusive)
^	Vertical bar	Bitwise OR (exclusive)
>>	Right angle brackets	Shift right
<<	Left angle brackets	Shift left

6.6.1.5 Logical Operators - Logical operators return a one (1) if the the result of the operation is true, and zero (0) if the result of the operation is false.

Table 6-9 summarizes the logical operators.

Table 6-9: Logical Operators

Operator	Operator Name	Operation
!	NOT	Logical NOT operator
?	DEFINED	Defined operator
&&	AND	Logical AND operator
||	OR	Logical OR operator

6.6.1.6 Relational Operators - Relational operators return a one (1) if the the result of the operation is true, and zero (0) if the result of the operation is false.

Table 6-10 summarizes the Relational operators.

Table 6-10: Relational Operators

Operator	Operator Name	Operation
=	Equal	Equal
<>	Not equal	Not equal
!=	Not equal	Not equal
<	Less than	Less than
<=	Less than or equal	Less than or equal
>	Greater than	Greater than
>=	Greater than or equal	Greater than or equal

6.6.1.7 Operator Precedences And Associativity - Table 6-11 summarizes the operator precedences and associativity in the assembler. These operators are listed in order of decreasing precedence. Operators grouped together have the same precedence and are associated from left to right.

Table 6-11: Operator Precedence

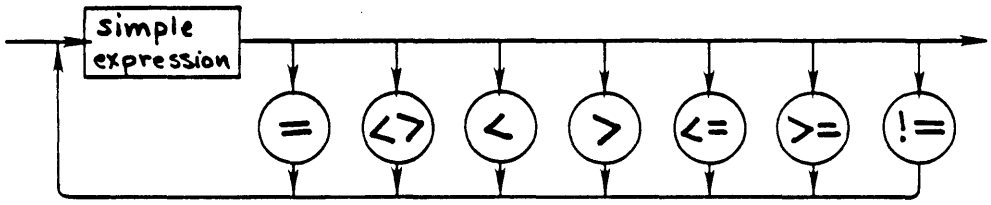
Operator	Function	
`B	Binary value operator	Highest Precedence
`O	Octal value operator	
`D	Decimal value operator	
`H	Hexadecimal value operator	
$	Hexadecimal value operator	
`R	Round real operator	
`T	Truncate real operator	
`F	Coerce real operator	
`U	Upper real operator	
`L	Lower real operator	
+	Unary plus operator	
-	Unary minus operator	
~	1's complement operator	
!	Logical NOT operator	
?	Symbol defined operator	
*	Multiplication operator	
/	Division operator	
%	Remainder operator	
&	Bitwise AND operator	
&&	Logical AND operator	
<<	Shift left operator	
>>	Shift right operator	
+	Addition operator	
-	Subtraction operator	
^	Bitwise exclusive OR operator	
|	Bitwise inclusive OR operator	
||	Logical OR operator	
<	Less than operator	
<=	Less than or equal operator	
>	Greater than operator	
>=	Greater than or equal operator	
<>	Not equal operator	
!=	Not equal operator	
=	Equal operator	Lowest Precedence

6.6.2 Expression

An expression consists either a simple expression or a simple expression followed by any of the relational operators followed by another simple expression. All relational operators have equal precedences. Expressions can be grouped for evaluation by enclosing them in parentheses. The enclosed expressions are evaluated first, and all remaining operations are performed from left to right.

Relative and external values are not allowed as operands to relational operators.

Figure 6-7 summaries the syntax of expressions.



6.6.3 Simple Expressions

A simple expression consists either a term or a sign followed by a term or a simple expression followed by of the simple operator followed by a term. All simple operators have equal precedences. Simple expressions can be grouped for evaluation by enclosing them in parentheses. The enclosed simple expressions are evaluated first, and all remaining operations are performed from left to right.

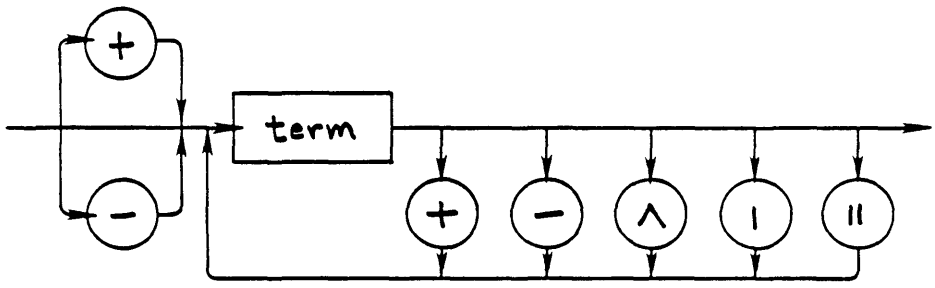
Relative and external values are not allowed as operands to simple operators with the exception of binary plus and minus operators. Table 6-12 summarizes the use of relative and external values in these operators.

Table 6-12:

Operation	Result
Absolute + Absolute	Absolute
Absolute + Relative	Relative
Absolute + External	External - Relative
Relative + Absolute	Relative
Relative + Relative	ERROR
Relative + External	ERROR
External + Absolute	External - Relative
External + Relative	ERROR
External + External	ERROR
Absolute - Absolute	Absolute
Absolute - Relative	ERROR
Absolute - External	ERROR
Relative - Absolute	Relative
Relative - Relative	Absolute *
Relative - External	ERROR
External - Absolute	External - Relative
External - Relative	ERROR
External - External	ERROR

* If and only if both relative values are defined in the same module and section, otherwise, the result is an ERROR.

Figure 6-8 summaries the syntax of simple expressions.

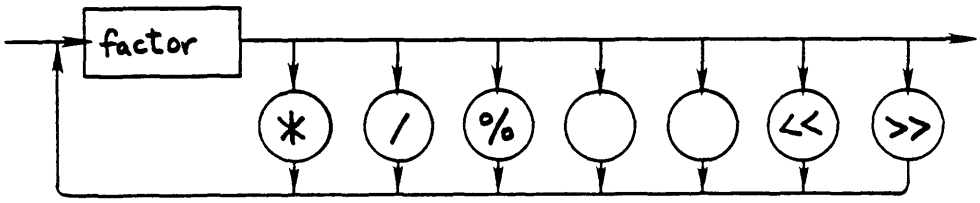


6.6.4 Terms

A term consists either a factor or a term followed by a term operator followed by a factor. All term operators have equal precedences. Terms can be grouped for evaluation by enclosing them in parentheses. The enclosed term are evaluated first, and all remaining operations are performed from left to right.

Relative and external values are not allowed as operands to term operators.

Figure 6-9 summaries the syntax of term.

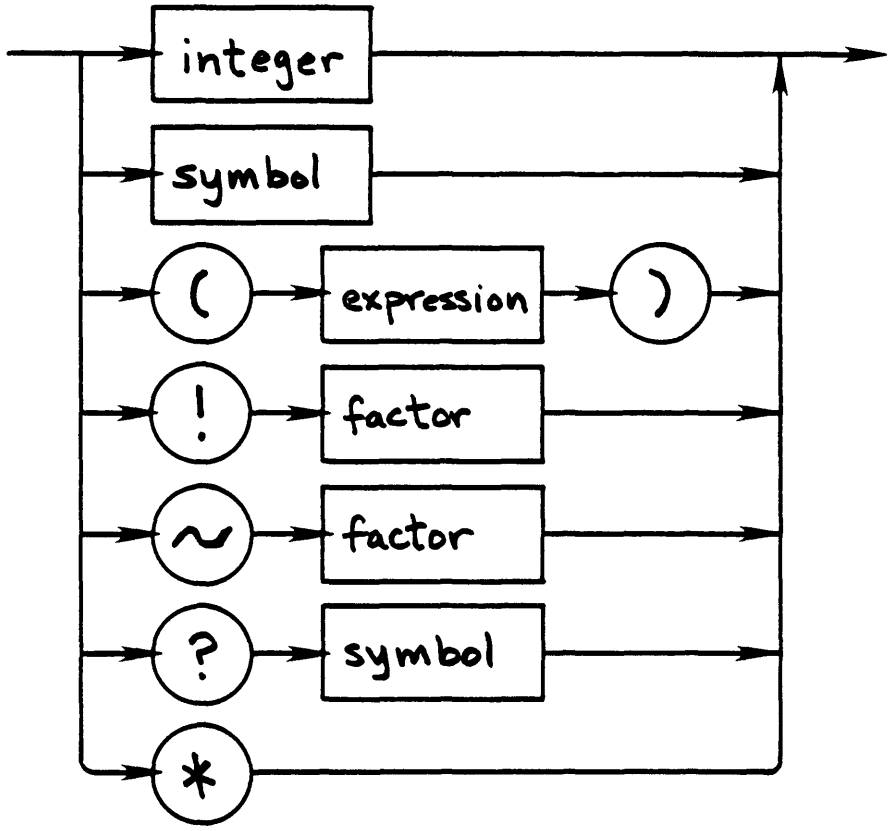


6.6.5 Factors

A factor consists of any of the following:

1. Number
2. Symbol
3. Current location counter (*)
4. One's compliment operator (~)
5. Logical NOT operator (!)
6. Defined symbol operator (?)
7. An expression enclosed by parenthesis

Figure 6-10 summaries the syntax of factors.


integer
symbol
(
expression
)
!
factor
~
factor
?
symbol
*

CHAPTER 7

ASSEMBLER DIRECTIVES

The general assembler directives (pseudo-opcodes) provide facilities for performing various assembler functions. Table 7-1 lists these functions and the directives in each category. The remainder of this chapter describes the directives in detail, showing their formats and giving examples of their use. For ease of reference, the directives are presented in alphabetical order in this chapter. In addition, Appendix C contains a summary of all assembler directives.

Assembler directives are written in the same way as instructions, but (with the exception of the .DC.x and .DS.x directives) do not cause any code to be generated. All assembler directives begin with a period (.). This style of naming directives enables the user to quickly identify assembler directives from actual machine instructions.

Table 7-1: Assembler Directive Summary

Category	Directives
Assembler Option Directives	.DISABL
	.ENABLE
Listing Control Directives	.HEADER
	.LIST
	.NOLIST
	.PAGE
Message Display Directives	.ERROR
	.FATAL
	.PRINT
	.WARN

Table 7-1 (Cont.): Assembler Directive Summary

Category	Directives
Module and Block Directives	.BEGIN
	.END
	.MODULE
Program Sectioning Directive	.SECT
Symbol Assignment Directives	.ABSADR
	.ADDR
	.CONST
Symbol Attribute Directives	.EXTERN
	.GLOBAL
	.LOCAL
	.NONUSR
	.USER
	.WEAK
Data Definition Directives	.ENDS
	.STRUCT
Data Storage Directives	.DC.x
	.DS.x
	.COMMON
Location Control Directive	.ALIGN
Conditional Assembly Directives	.ELSE
	.ENDC
	.IF
Miscellaneous Directives	.CONFIG
	.EOF
	.FILE
	.INCLD
	.LINE
	.LINKER
	.PROCSS

Any directives may have a label, some require it. If a directive requires a label, the label must appear on the same line as the directive. For example:

```
foo:    .const  10
```

and

```
foo:
        .const  10
```

are not equivalent. The first case shows a valid label for the .CONST directive. In the second case, "foo" is not a label for the .CONST directive and this case will cause in an assembler error (Label required).

.ABSADR .ABSADR

NAME: .ABSADR -- Absolute address definition directive.

FORMAT: <label> .ABSADR <address> [,<domain>]

PARAMETERS:

<address> = any legal expression. The expression can contain forward references but cannot contain any external, relative, or unresolved symbols. The expression must evaluate to an assemble-time constant.

<domain> = any legal expression. The expression can contain forward references but cannot contain any external, relative, or unresolved symbols. The expression must evaluate to an assemble-time constant. The domain must be in the range of 0 through 254. The default domain value is zero. This parameter is optional.

DESCRIPTION: The .ABSADR directive is used to assign an absolute address to the user symbol in the label field. This is particularly useful when writing software that accesses absolute address in an address space. An example of this is device drivers that must directly communicate to device specific locations. These locations can be in memory (memory mapped I/O), or on an I/O bus, or in some other address space. The <domain> parameter is used to specify a specific address space. This form is used only on machines that support multiple address spaces (e.g. I/O buses). A domain of zero is considered the native or host address space and is the default.

An absolute address is treated internally by the assembler as a signed constant.

NOTES:

1. This directive must appear inside a module (see .MODULE).

.ABSADR	(Continued)	.ABSADR

2. A label is required by this directive. The <label> cannot be defined anywhere else at this module level.

3. This directive is not supported by all object file formats. Refer to Appendix F for specific object file limitations.

.ADDR .ADDR

NAME: .ADDR — Address definition directive.

FORMAT: <label> .ADDR <address>

PARAMETERS:

<address> = any legal expression. The expression can contain forward references and relative symbols but cannot contain any external or unresolved symbols.

DESCRIPTION: The .ADDR directive assigns the value of the address expression to the user symbol in the label field. The value of the expression is relative to the beginning of the section in which the definition appears. If the definition appears in an absolute section, the symbol is assigned the value of an absolute address (see .ABSADR), otherwise the symbol value is assigned a relative value.

.ADDR performs the same function as a program label definition, except .ADDR can appear anywhere within the section. The .ADDR directive performs an out of line address definition, a label definition is an in line address definition. For example the following code segments are equivalent.

```
        .sect   10                      .sect   10
        bra     foo                     bra     foo
foo:    .addr   * + 100
        .ds.b   100                     .ds.b   100
        nop                     foo:    nop
```

An address value is treated internally by the assembler as a signed constant.

NOTES:

1. This directive must appear inside a section (see .SECT).

.ADDR	(Continued)	.ADDR

2. A label is required by this directive. The <label> cannot be defined anywhere else at this module level.

.ALIGN .ALIGN

NAME: .ALIGN -- Align location counter directive.

FORMAT: .ALIGN <address> [,<fill>]
.ALIGN <keyword> [,<fill>]

PARAMETERS:

<address> = any legal expression. The expression cannot contain any forward references, external symbols, relative symbols, or unresolved symbols. The expression must evaluate to an assemble-time constant in the first pass.

<keyword> = any keyword listed in Table 7-2. Either the long form or short form of the keyword may be used.

<fill> = any legal expression. The expression can contain forward references, but cannot contain any external, relative, or unresolved symbols. The expression must evaluate to an assemble-time constant. The fill value must be in the range of 0 through 255. The default fill value is zero. This parameter is optional.

DESCRIPTION: The .ALIGN directive aligns the location counter to the boundary specified by either the <address> or the <keyword> parameter. If the <address> parameter is used, the location counter is set to the value of the <address> expression. If the <keyword> parameter is used, the location counter is aligned to the address that is the next multiple of the value listed in Table 7-2 under "Size in Bytes". If the optional <fill> value is supplied, the bytes skipped by the location counter (if any) are filled with the specified value, otherwise, the bytes are zero filled. If the fill value is larger than 255, the value is truncated and a warning message is printed.

.ALIGN (Continued) .ALIGN

Table 7-2:

Long Form	Short Form	Size in Bytes	Description
EVEN	EV	1	The EVEN keyword ensures that the current value of the location counter is even. If the location counter is odd, EVEN will add one to the its value. If the location counter is already even, no action is taken.
ODD	OD	1	The ODD keyword ensures that the current value of the location counter is odd. If the location counter is even, ODD will add one to the its value. If the location counter is already odd, no action is taken.
BYTE	BY	1	Align the location counter to a byte boundary. This keyword performs no action.

.ALIGN (Continued) .ALIGN

Table 7-2:

Long Form	Short Form	Size in Bytes	Description
WORD	WD	2	Align the location counter to a word boundary. If the current is aligned to a word no action is taken. This is equivalent to the EVEN keyword.
LONG	LG	4	Align the location counter to a long word boundary. If the current value is aligned to a long word, no action is taken.
PAGE	PG	4096	Align the location to a page boundary. If the current value is aligned to a page, no action is taken.

NOTES:

1. This directive must appear inside a section (see .SECT).

2. If a label is specified, it is assigned the value of the location counter before any aligning is performed.

.BEGIN .BEGIN

NAME: .BEGIN — Begin block directive.

FORMAT: .BEGIN [<name>]

PARAMETERS:

<name> = any legal user-defined symbol. The default name is the null name (a null string). This parameter is optional.

DESCRIPTION: The .BEGIN directive is used to start a new symbol scoping level. It does not change the value of the location counter.

If the optional <name> is specified, it cannot be defined anywhere else in the activating module. A name should be specified in the .BEGIN directive and in the corresponding .END directive, so that the assembler can detect any improperly nested scoping blocks.

This directive is similar in function to an inner block in the C language.

NOTES:

1. This directive must appear inside a module (see .MODULE).

2. If a label is specified, it is assigned the current value of the location counter.

.COMMON .COMMON

NAME: .COMMON — Common region definition directive.

FORMAT: .COMMON <name> [,<size>]

PARAMETERS:

<name> = any legal user-defined symbol.

<size> = any legal expression. The expression can contain forward references but cannot contain any external, relative, or unresolved symbols. The expression must evaluate to an assemble-time constant. The size must be a positive value. The default size value is zero. This parameter is optional.

DESCRIPTION: The .COMMON directive defines <name> as a common region with length of <size> bytes. For commons with the same name and different sizes, the largest size is used. No storage is allocated for common regions by the assembler, this is done by the linker. The space reserved by the .COMMON directive is considered "out-of-line" storage allocation, the user has no control over the placement of this space.

NOTES:

1. This directive must appear inside a section (see .SECT).

2. The space allocated by the common directive is not necessarily allocated in the currently defined section. Appendix F describes where common regions are placed by the various linkers.

.CONFIG .CONFIG

NAME: .CONFIG -- Configuration directive.

FORMAT: .CONFIG "<configuration_string>"

PARAMETERS:

<configuration_string> = any legal character string. Null strings are not allowed.

DESCRIPTION: The .CONFIG directive places the specified configuration string (without the double quotes (")) into the object file. This information can be used by the various linkers and loaders. The assembler does not enforce any format or structure on this string. The format and structure of the configuration string are specified by the linkers and loaders. Refer to Appendix F for the effects of this directive on specific object files.

NOTES:

1. The double quotes (") are not part of the configuration string.

2. This directive is not supported by all object file formats. Refer to Appendix F for specific object file limitations.

.CONST	.CONST

NAME: .CONST -- Constant symbol definition directive.

FORMAT: <label> .CONST <constant>

PARAMETERS:

<constant> = any legal expression. The expression can contain forward references but cannot contain any external, relative, or unresolved symbols. The expression must evaluate to an assemble-time constant.

DESCRIPTION: The .CONST directive assigns the value of the constant expression to the symbol in the label field. .CONST can appear outside module definitions. This directive is useful for assigning values to conditional assembly symbols.

NOTES:

1. A label is required by this directive. The <label> cannot be defined anywhere else at this module level.

2. A .CONST directive can appear any where in the user's program.

.DC.x .DC.x

NAME: .DC.x — Data storage directive.

FORMAT:
```
.DC.B   <constant> {,<constant>}
.DC.W   <constant> {,<constant>}
.DC.L   <constant> {,<constant>}
.DC.F   <fp_constant> {,<fp_constant>}
.DC.D   <fp_constant> {,<fp_constant>}
```

PARAMETERS:

<constant> = any legal expression or character string. The expression can contain forward references, external symbols, and relative symbols but cannot contain any unresolved symbols. Null character strings are allowed, but allocate no space. External and relative symbols are resolved at link-time.

<fp_constant> = any legal real number. Expressions are not allowed.

DESCRIPTION: The .DC directive stores a byte, word or long integer, or a float or double real number. If the <constant> is an expression, the expression is evaluated as a 32-bit value. If this value is larger than the allocated space, the constant is truncated and a warning message is printed. If multiple constants are specified, they must be separated by commas.

The space reserved by the .DC directive is considered "in-line" storage allocation.

.DC.x (Continued) .DC.x

Table 7-3:

Directive	Name	Range of Values Allowed
.DC.B	Byte	-128 to +255
.DC.W	Word	-32768 to +65535
.DC.L	Long	-2147483648 to +4294967295
.DC.F	Float	8.43E-37 to 3.37E+38
.DC.D	Double	4.19E-307 to 1.67E+308

**** WARNING ****

At link-time, a relocatable value or expression can result in value that exceeds the specified size. Not all linkers will issue a truncation warning message.

NOTES:

1. This directive must appear inside a section (see .SECT).

.DISABL .DISABL

NAME: .DISABL — Function control directive.

FORMAT: .DISABL <keyword> {,<keyword>}

PARAMETERS:

<keyword> = any keyword listed in Table 7-4. Either the long form or short form of the keyword can be used.

DESCRIPTION: The .DISABL directive disables, or inhibits, the specified assembler function. .DISABL is the negative form of .ENABLE. Refer to Table 7-4 for specific functions.

If multiple keywords are used, they must be separated by commas.

Table 7-4:

Long Form	Short Form	Default Condition	Description
EXTERNAL	EX	Disabled	When EXTERNAL is disabled, any undefined symbol that is not listed in a .EXTERN directive causes an error.
USER	US	Enabled	When USER is disabled, any symbol that is not listed in a .USER directive is considered a NONUSER symbol (see .NONUSER)

.DISABL (Continued) .DISABL

Table 7-4:

Long Form	Short Form	Default Condition	Description
UPPER	UC	Disabled	When UPPER is disabled, mapping of user-defined symbols to upper-case is terminated and no case mapping is performed on subsequent user-defined symbols (return to MIXED mode).
LOWER	LC	Disabled	When LOWER is disabled, mapping of user-defined symbols to lower-case is terminated and no case mapping is performed on subsequent user-defined symbols (return to MIXED mode).
MIXED	MC	Enabled	This keyword is ignored in the .DISABL directive.
LOCAL	LS	Disabled	When LOCAL is disabled, all subsequently defined local symbols are removed from the object file symbol table.

.DISABL (Continued) .DISABL

Table 7-4:

Long Form	Short Form	Default Condition	Description
NONUSR	NS	Disabled	When NONUSR is disabled, all subsequently defined local symbols are removed from the object file symbol table.
CROSS	CR	Enabled	Not implemented.

NOTES:

1.

.DS.x .DS.x

NAME: .DS.x — Storage allocation directive.

FORMAT:
```
.DS.B   <size> [,<fill>]
.DS.W   <size> [,<fill>]
.DS.L   <size> [,<fill>]
.DS.S   <size> [,<fill>]
```

PARAMETERS:

<size> = any legal expression. The expression cannot contain any forward references, external symbols, relative symbols, or unresolved symbols. The expression must evaluate to an assemble-time constant in the first pass.

<fill> = any legal expression. The expression can contain forward references but cannot contain any external, relative, or unresolved symbols. The expression must evaluate to an assemble-time constant. The fill value must be in the range of 0 through 255. The default fill value is zero. This parameter is optional.

DESCRIPTION: Each .DS directive allocates storage for the different data types. The value of <size> determines the number of data items for which the assembler reserves storage. The total number of bytes reserved is equal to the length of the data type (see Table 7-5) multiplied by the value of <size>. If the optional fill value is specified, then each data location is initialized to that value. Otherwise, the data locations are initialized to zero.

The space reserved by the .DS directive is considered "in-line" storage allocation.

.DS.x	(Continued)	.DS.x

Table 7-5:

Directive	Name	Number of Bytes Allocated
.DS.B	Byte	1 * value of <size>
.DS.W	Word	2 * value of <size>
.DS.L	Long	4 * value of <size>
.DS.S	Struct	1 * size of structure

NOTES:

1. This directive must appear inside a section (see .SECT).

.ELSE

.ELSE

NAME: .ELSE -- Conditional assembly else directive.

FORMAT: .ELSE

PARAMETERS: None.

DESCRIPTION: The .ELSE directive begins the optional ELSE block of the .IF directive. If the expression in the corresponding .IF directive evaluates to zero, the statements between the .ELSE and the corresponding .ENDC are assembled. Otherwise the statements are skipped.

NOTES:

1.

.ENABLE .ENABLE

NAME: .ENABLE -- Function control directive.

FORMAT: .ENABLE <keyword> {,<keyword>}

PARAMETERS:

<keyword> = any keyword listed in Table 7-6. Either the long form or short form of the keyword can be used.

DESCRIPTION: The .ENABLE directive enables the specified assembler functions. .ENABLE and its negative form, .DISABL, control the functions listed in Table 7-6 assembly functions. Refer to Table 7-6 for specific functions.

If multiple keywords are used, they must be separated by commas.

Table 7-6:

Long Form	Short Form	Default Condition	Description
EXTERNAL	EX	Disabled	When EXTERNAL is enabled, all undefined symbols are considered EXTERNAL symbols (see .EXTERN).
USER	US	Enabled	When USER is enabled, any symbol that is not listed in a .NONUSR directive is considered a USER symbol (see .USER).

.ENABLE (Continued) .ENABLE

Table 7-6:

Long Form	Short Form	Default Condition	Description
UPPER	UC	Disabled	When UPPER is enabled, all subsequent user-defined symbols are mapped to uppercase.
LOWER	LC	Disabled	When LOWER is enabled, all subsequent user-defined symbols are mapped to lowercase.
MIXED	MC	Enabled	When MIXED is enabled, any case-conversion options (UPPER, LOWER) is terminated and all subsequent user-defined symbols are not converted.
LOCAL	LS	Disabled	When LOCAL is enabled, all subsequent user-defined LOCAL symbols (see .LOCAL) are included in the object file symbol table. This function is useful when debugging.

.ENABLE (Continued) .ENABLE

Table 7-6:

Long Form	Short Form	Default Condition	Description
NONUSR	NS	Disabled	When NONUSR is enabled, all subsequent user-defined NONUSR symbols (see .NONUSR) are included in the object file symbol table. This function is useful when debugging.
CROSS	CR	Enabled	Not implemented.

NOTES:

1.

.ENDC .ENDC

NAME: .ENDC — Conditional assembly end directive.

FORMAT: .ENDC

PARAMETERS: None.

DESCRIPTION: The .ENDC directive terminates the conditional assembly block started by the .IF directive. See the description of .IF for more information.

NOTES:

1.

.END	.END

NAME: .END — Module and block end directive.

FORMAT: .END [<name>]

PARAMETERS:

<name> = any legal user-defined symbol. This parameter is optional.

DESCRIPTION: .END terminates a .MODULE or .BEGIN definition. If .END is encountered without a corresponding .MODULE or .BEGIN directive, the assembler displays an error message. If the optional name is specified, it must match the name defined in the corresponding .MODULE or .BEGIN directive. The use of the name is strongly recommended so that the assembler can detect any improperly nested modules or begin blocks.

NOTES:

1.

.ENDS .ENDS

NAME: .ENDS — Structure definition end directive.

FORMAT: .ENDS [<name>]

PARAMETERS:

<name> = any legal user-defined symbol. This parameter is optional.

DESCRIPTION: The .ENDS directive terminates a structure definition started by the .STRUCT directive. See the description of .STRUCT for more information. If the optional name is specified, it must match the name defined in the corresponding .STRUCT directive. The name should be specified so that the assembler can detect any improperly nested structures.

NOTES:

1.

.EOF	.EOF

NAME: .EOF — Assembly termination directive.

FORMAT: .EOF

PARAMETERS: None.

DESCRIPTION: The .EOF directive terminates the source program. Subsequent source lines are ignored and not included in the listing file or the object file.

The .EOF directive is not required to terminate a source file. When the assembler encounters a physical end of file, it is interrupted as a .EOF directive. An exception occurs if the assembler encounters a physical end of file inside an include file, then the next 'outer nested' file is read from. However, if the assembler encounters a .EOF inside an include file, all source line processing is terminated at that point.

.ERROR .ERROR

NAME: .ERROR -- Error message print directive.

FORMAT: .ERROR "<message>"

PARAMETERS:

<message> = any legal character string. Null strings are allowed.

DESCRIPTION: .ERROR causes the assembler to display an error message on standard error and in the listing file (if applicable). .ERROR can be used to display an error message when a macro call or conditional assembly contains an undesirable set of conditions.

User-generated error messages have the form:

** <name>-Error <file_name> [User-generated]: <message>

Where:

<name> = to the assembler name.
<file_name> = to the source file that generated the error message.
<message> = to message string to be printed

The '[User-generated]' distinguishes it from error messages generated by the assembler. The double quotes (") do not appear as part of the printed message.

When the assembly is finished, the assembler displays the total number of errors encountered, this includes both assembler and user-generated errors.

NOTES:

1. The line containing the .ERROR directive is not included in the listing file.

.ERROR	(Continued)	.ERROR

.EXTERN .EXTERN

NAME: .EXTERN -- External symbol definition directive.

FORMAT: .EXTERN <symbol> {,<symbol>}

PARAMETERS:

<symbol> = any legal user-defined symbol.

DESCRIPTION: The .EXTERN directive indicated that the specified symbols are external to this module.

If the EXTERNAL keyword is enabled (see the description of .ENABLE), all unresolved symbols are declared external. Thus, if EXTERNAL is enabled, the programmer need not specify symbols as external using the .EXTERN directive. However, if EXTERNAL is disabled, the programmer must explicitly use .EXTERN to declare any symbols that are defined externally but referred to in the current module. If EXTERNAL is disabled and the assembler finds symbols that are not defined in the current module and are not listed in a .EXTERN directive, an error message is printed.

If a symbol is declared as external, and then defined in the current module, an error message is printed.

If multiple symbols are specified, they must be separated by commas.

NOTES:

1. This directive must appear inside a module (see .MODULE).

.FATAL .FATAL

NAME: .FATAL -- Fatal message print directive.

FORMAT: .FATAL "<message>"

PARAMETERS:

<message> = any legal character string. Null strings are allowed.

DESCRIPTION: .FATAL causes the assembler to display a fatal message on standard error and in the listing file (if applicable). .FATAL can be used to display an error message when a macro call or conditional assembly contains a disastrous set of conditions.

User-generated fatal messages have the form:

** <name>-Fatal <file_name> [User-generated]: <message>

Where:

<name> = to the assembler name.
<file_name> = to the source file that generated the fatal message.
<message> = to message string to be printed

The '[User-generated]' distinguishes it from fatal messages generated by the assembler. The double quotes (") do not appear as part of the printed message.

The .FATAL directive causes the assembler to immediately abort in pass one and no pass two processing will occur. All source line processing terminates at that point.

NOTES:

1. The line containing the .FATAL directive is not included in the listing file.

.FATAL	(Continued)	.FATAL

2. The assembler exit value will be set an error value. This value is operating system dependent.

.FILE

NAME: .FILE — Source file definition directive.

FORMAT: .FILE <number> [,"<name>" [,<modify_time>]]

PARAMETERS:

<number> = any legal integer number. The file number is represented as a unsigned integer and has the range of 0 through +4294967295. Negative numbers and expressions are not allowed.

<name> = any legal character string. Null strings are not allowed. This parameter is optional.

<modify_time> = any legal integer number. The modify time is represented as a signed integer and has the range of -2147483648 through +2147483647. The default modify time is zero. This parameter is optional. Expressions are not allowed.

DESCRIPTION: The .FILE directive is used to control the source file name that is displayed whenever an error message is issued by the linker or the runtime loader. This information is also used by the debuggers.

The optional <name> and <modify_time> are only necessary when the first reference to the file is made. If the first reference does not include a file name, an assembler error will result. Subsequent references only need to specify the <number>.

NOTES:

1. The assembler does not enforce any path or file naming convention. This is specified by the host operating system.

2. The double quotes (") are not passed as part of the file name.

.FILE	(Continued)	.FILE

3. File numbers can be redefined by specifying a new file name to the number.

.GLOBAL .GLOBAL

NAME: .GLOBAL -- Global symbol declaration directive.

FORMAT: .GLOBAL <symbol> {,<symbol>}

PARAMETERS:

<symbol> = any legal user-defined symbol.

DESCRIPTION: The .GLOBAL directive indicates that the specified symbols are declared as global symbols and are exported one level out (made visible outside the current module).

If multiple symbols are specified, they must be separated by commas.

NOTES:

1. This directive must appear inside a module (see .MODULE).
2. If a symbol is declared global, but not defined in the current module, an error is printed.

.HEADER .HEADER

NAME: .HEADER -- Listing header directive.

FORMAT: .HEADER "<string>"

PARAMETERS:

<string> = any legal character string. Null strings are allowed.

DESCRIPTION: The .HEADER directive causes the assembler to print the <sting> on the third line of each page of the listing file. A null <string> will clear or blank the previous header sting. This directive is ignored if no listing file was specified.

NOTES:

1. The .HEADER directive takes affect on the next listing page generated, unless it is the first opcode on the page.
2. The header string can be up to 254 characters long, however, it is recommended that it is limited to the width of the listing page.
3. The .HEADER string is initially set to the null string (blank).

.IF .IF

NAME: .IF — Conditional assembly if directive.

FORMAT: .IF <expression>

PARAMETERS:

<expression> = any legal expression. The expression cannot contain any forward references, external symbols, relative symbols, or unresolved symbols. The expression must evaluate to an assemble-time constant in the first pass.

DESCRIPTION: A conditional assembly block is a series of source statements that is assembled only if a certain condition is met at assembly-time. .IF starts the conditional block and .ENDC ends the conditional block. An optional .ELSE can appear between .IF and .ENDC. Each .IF must have an corresponding .ENDC. The .IF directive contains an expression which is evaluated (as 32-bits). If the result is non-zero, all the source lines up until the .ELSE or .ENDC directives are assembled. If the expression evaluates to zero, all source lines up until the .ELSE or .ENDC directives are skipped. If a .ELSE directive is encountered, then the lines between the .ELSE and .ENDC are assembled.

Conditional blocks can be nested, that is a conditional block can be inside of another conditional block. In this case the statements in the inner conditional block are assembled only if the condition is met for both the outer and inner block. .IF directives can be nested 16 levels deep. If a statement attempts to exceed this nesting level depth, the assembler displays an error message.

.IF	(Continued)	.IF

NOTES:

1.

.INCLD .INCLD

NAME: .INCLD — Include file directive.

FORMAT: .INCLD "<file_name>"

PARAMETERS:

<file_name> = any legal character string. Null strings are not allowed.

DESCRIPTION: The .INCLD directive includes the <file_name> in the source stream. Include files may be nested. If a .EOF directive is encountered in an include file, all subsequent source lines are ignored. Nested .INCLD are allowed to 16 levels.

NOTES:

1.

.LINE .LINE

NAME: .LINE -- Source line number definition directive.

FORMAT: .LINE [+|-] [<number>]

PARAMETERS:

<number> = any legal integer number. The line number is represented as a unsigned integer and has the range of 0 through +4294967295. This parameter is optional. Negative numbers and expressions are not allowed.

DESCRIPTION: The .LINE directive is used to control the line number that is displayed whenever an error message is issued by the linker or the runtime loader. This information is also used by the debuggers.

The various functions of the .LINE directive are summarized in Table 7-7.

Table 7-7:

Directive	Description
.LINE	Increment the current source line number by one.
.LINE -	Decrement the current source line number by one.
.LINE +	Increment the current source line number by one.
.LINE <number>	Set the source line number to <number>.
.LINE -<number>	Set the source line number to the current source line number minus <number>.

.LINE (Continued) .LINE

Table 7-7:

Directive	Description
.LINE +<number>	Set the source line number to the current source line number plus <number>.

NOTES:

1.

.LINKER .LINKER

NAME: .LINKER — Linker directive.

FORMAT: .LINKER <value> {,<value>}

PARAMETERS:

<value> = any legal expression. The expression can contain forward references, but cannot contain any external, relative or unresolved symbols. The expression must evaluate to an assemble-time constant.

DESCRIPTION: The .LINKER directive is used to pass information directly to the linker. Detailed knowledge of the object format is needed to used this directive. Extreme caution should be taken in using this directive. Refer to Appendix F for the effects of this directive on specific object files.

NOTES:

1.

.LIST .LIST

NAME: .LIST -- Listing control directive.

FORMAT: .LIST <keyword> {,<keyword>}

PARAMETERS:

<keyword> = any keyword listed in Table 7-8. Either the long form or short form of the keyword can be used.

DESCRIPTION: .LIST and its negative form, .NOLIST, specify listing control options in the source text of a program. .LIST causes certain types of lines to be included in the listing file.

Each keyword can be used alone or in combination with other keywords. If multiple keyword are specified, they must be separated by commas.

Table 7-8:

Long Form	Short Form	Default Condition	Description
PAGE	PG	List	Enable the use of the .PAGE directive.
CONDITIONS	CA	List	Not implemented.
DEFINITION	MD	List	Not implemented.
CALLS	MC	List	Not implemented.
EXPANSIONS	ME	List	Not implemented.
SOURCE	SL	List	List source lines.
TSTATES	TS	List	Not implemented.

.LIST (Continued) .LIST

Table 7-8:

Long Form	Short Form	Default Condition	Description
WICAT	WS	No List	Print the WICAT proprietary statement on the second line of each listing page. This statement will be printed on every listing page, regardless of where the .LIST directive is encountered.

NOTES:

1.

.LOCAL .LOCAL

NAME: .LOCAL — Local symbol declaration directive.

FORMAT: .LOCAL <symbol> {,<symbol>}

PARAMETERS:

<symbol> = any legal user-defined symbol.

DESCRIPTION: The .LOCAL directive indicates that specified symbols are declared as local symbols. If a symbol is not declared as a .GLOBAL or .EXTERN, then it is assumed to be local. If a symbol is declared, but never defined (e.g. as a label), an error is printed.

.LOCAL is the default for all symbols.

NOTES:

1. This directive must appear inside a module (see .MODULE).

.MODULE .MODULE

NAME: .MODULE -- Module definition directive.

FORMAT: .MODULE [<name>]

PARAMETERS:

<name> = any legal user-defined symbol. The default name is the null name (null string). This parameter is optional.

DESCRIPTION: The .MODULE directives is used to begin a new procedure or subroutine. Each module defines a new scoping symbol enviroment. Any previous defined section (see. SECT) is pushed onto the section stack. Modules may be nested.

NOTES:

1.

.NOLIST .NOLIST

NAME: .NOLIST -- Listing control directive.

FORMAT: .NOLIST <keyword> {,<keyword>}

PARAMETERS:

<keyword> = any keyword listed in Table 7-9. Either the long form or short form of the keyword can be used.

DESCRIPTION: .NOLIST specify listing control options in the source text of a program. .NOLIST causes certain types of lines to be excluded in the listing file.

Each keyword can be used alone or in combination with other keywords. If multiple keyword are specified, they must be separated by commas.

Table 7-9:

Long Form	Short Form	Default Condition	Description
PAGE	PG	List	Disable the use of .PAGE.
CONDITIONS	CA	List	Not implemented.
DEFINITION	MD	List	Not implemented.
CALLS	MC	List	Not implemented.
EXPANSIONS	ME	List	Not implemented.
SOURCE	SL	List	Do not list source lines.
TSTATES	TS	List	Not implemented.

.NOLIST	(Continued)	.NOLIST

Table 7-9:

Long Form	Short Form	Default Condition	Description
WICAT	WS	No List	Do not list the WICAT proprietary statement.

NOTES:

1.

.NONUSR .NONUSR

NAME: .NONUSR -- Non-user symbol declaration directive.

FORMAT: .NONUSR <symbol> {,<symbol>}

PARAMETERS:

<symbol> = any legal user-defined symbol.

DESCRIPTION: The .NONUSR directive indicates that specified symbols are declared as non-user defined symbols. This directive is useful for distinguishing user defined symbols form non-user defined symbols (e.g. compile symbols). By default, non-user symbols not included in object files.

If the USER keyword is disabled (see the description of .DISABL), all undeclared symbols will be assumed to be declared as non-user symbols (see .NONUSR). Thus, if the USER keyword is disnabled, the programmer need not specify symbols as non-user using the .NONUSR directive. However, if USER is enabled (the default), the programmer must explicitly use .NONUSR to declare any symbols that are not defined by the user symbols in the current module.

If a symbol is declared as non-user, but never defined (e.g. as a label), an error will be reported.

If multiple symbols are specified, they must be separated by commas.

NOTES:

1. This directive must appear inside a module (see .MODULE).

.PAGE .PAGE

NAME: .PAGE -- Page advance directive.

FORMAT: .PAGE

PARAMETERS: None.

DESCRIPTION: The .PAGE directive advances the listing file to the top of the next page. This is accomplished by writing a Form Feed (^L) character into the listing file. If the listing file is printed on a device that does not support the Form Feed character, the use of the .PAGE directive is ineffective.

.PRINT .PRINT

NAME: .PRINT -- Print message directive.

FORMAT: .PRINT "<message>"

PARAMETERS:

<message> = any legal character string. Null strings are allowed.

DESCRIPTION: .PRINT causes the assembler to display an informational message on standard error and in the listing file (if applicable). .PRINT can be used to display an informational message. The message produced by .PRINT is not considered an error or warning message.

User-generated messages have the form:

** <name>-Print <file_name> [User-generated]: <message>

Where:

<name> = to the assembler name.
<file_name> = to the source file that generated the message.
<message> = to message string to be printed

The '[User-generated]' distinguishes it from messages generated by the assembler. The double quotes (") do not appear as part of the printed message.

NOTES:

1. The line containing the .PRINT directive is not included in the listing file.

.PROCSS .PROCSS

NAME: .PROCSS — Processor definition directive.

FORMAT: .PROCSS <type>

PARAMETERS:

<type> = any processor type listed in Table 7-10.

DESCRIPTION: The .PROCSS directive causes the assembler to accepts only the instructions and addressing modes of the specified processor.

Table 7-10:

Long Form	Short Form	Description
M68000	M0	Accept the M68000 instructions and addressing modes.
M68020	M2	Not implemented.

NOTES:

1.

.SECT .SECT

NAME: .SECT — Section definition directive.

FORMAT:
```
.SECT   <number> {,<attribute>}
.SECT   <name>   {,<attribute>}
```

PARAMETERS:

<number> = any legal integer number. The section number must be in the range of 0 through 254. Negative numbers and expressions are not allowed.

<name> = any legal user-defined symbol.

<attribute> = any attribute listed in Table 7-11. Either the long form or short form of the attribute can be used.

DESCRIPTION: The directive .SECT defines a section and its attributes. When the <name> parameter is used, the name must be either a predefined section name (see Table 7-12), or have be defined previously with the NAME attribute. The <number> parameter specifies the section number.

Each section can be defined to have the attributes listed in Table 7-11. However, once a section is defined, conflicting attributes are not allowed.

Table 7-11:

Long Form	Short Form	Description
ABSOLUTE	AS	The linker assigns the section to be absolute. The contents of this section can be code or data. The default section type is relative (see RELATIVE).

.SECT (Continued) .SECT

Table 7-11:

Long Form	Short Form	Description
RELATIVE	RS	The linker assigns the section type to be relocatable. The contents of this section can be code or data. This is the default section type.
ADDRESS = <value>	AD	Set the beginning address of this section to <value>. The default beginning address is zero.
NAME = <symbol>	NM	Assign <symbol> as the name of this section.

Table 7-12: Predefined Section Names

Name	Attributes
TEXT	RELATIVE
DATA	RELATIVE
BSS	RELATIVE
CSTR	RELATIVE
PURE	RELATIVE
IMPURE	RELATIVE

NOTES:

1. This directive must appear inside a module (see .MODULE).

2. The ABSOLUTE and ADDRESS keywords are not supported by all object file formats. Refer to Appendix F for specific object file limitations.

.STRUCT .STRUCT

NAME: .STRUCT -- Structure definition directive.

FORMAT: .STRUCT <name>

PARAMETERS:

<name> = any legal user-defined symbol.

DESCRIPTION: Not supported in this version.

.USER .USER

NAME: .USER -- User symbol definition directive.

FORMAT: .USER <symbol> {,<symbol>}

PARAMETERS:

<symbol> = any legal user-defined symbol.

DESCRIPTION: The .USER directive indicates that specified symbols are declared as user defined symbols. This directive is useful for distinguishing user defined symbols form non-user defined symbols (e.g. compile symbols). By default, non-user symbols not included in object files.

If the USER keyword is disabled (see the description of .DISABL), all undeclared symbols will be assumed to be declared as non-user symbols (see .NONUSR). Thus, if the USER keyword is enabled, the programmer need not specify symbols as user using the .USER directive. However, if USER is disabled, the programmer must explicitly use .USER to declare any symbols that are defined as user symbols in the current module.

If a symbol is declared as user, but never defined (e.g. as a label), an error will be reported.

If multiple symbols are specified, they must be separated by commas.

NOTES:

1. This directive must appear inside a module (see .MODULE).

.WARN .WARN

NAME: .WARN -- Warning message print directive.

FORMAT: .WARN "<message>"

PARAMETERS:

<message> = any legal character string. Null strings are allowed.

DESCRIPTION: .WARN causes the assembler to display a warning message on standard error and in the listing file (if applicable). .WARN can be used to display a warning message when a macro call or conditional assembly contains a questionable set of conditions.

User-generated warning messages have the form:

** <name>-Warn <file_name> [User-generated]: <message>

Where:

<name> = to the assembler name.
<file_name> = to the source file that generated the warning message.
<message> = to message string to be printed

The '[User-generated]' distinguishes it from warning messages generated by the assembler. The double quotes (") do not appear as part of the printed message.

When the assembly finishes, the assembler displays the total number of warning encountered, this includes both assembler and user-generated warnings.

NOTES:

1. The line containing the .WARN directive is not included in the listing file.

.WEAK .WEAK

NAME: .WEAK — Weak symbol declaration directive.

FORMAT: .WEAK <symbol> {,<symbol>}

PARAMETERS:

<symbol> = any legal user-defined symbol.

DESCRIPTION: .WEAK specifies that references to the name symbol(s) may be allowed to be unresolved during the link editing process. If a .WEAK symbol is resolved, the .WEAK directive has the same effect as .EXTERN. If the symbol is unresolved during linking, references to it are set to the default value of -1 (NOTE: this value may be changed at link-time. See the linker documentation information on how to modify the default value.). The linker does NOT report an error for unresolved weak symbols.

When .WEAK specifies a symbol that is defined in the current visible scope, then that definition is used, and the symbol is considered defined. In this case, the .WEAK declaration has no meaning.

If a weak symbol is not referenced in the current scope, then an error is reported.

If multiple symbols are specified, they must be separated by commas.

NOTES:

1. This directive must appear inside a module (see .MODULE).

2. This directive is not supported by all object file formats. Refer to Appendix F for specific object file limitations.

CHAPTER 8

INSTRUCTION SET AND ADDRESSING MODES

[To be written later]

CHAPTER 9

WRITING POSITION INDEPENDENT CODE

[To be written later]

APPENDIX A

ASCII CHARACTER TABLE

CHR	OCT	DEC	HEX	CHR	OCT	DEC	HEX	CHR	OCT	DEC	HEX	CHR	OCT	DEC	HEX
NUL(^@)	000	0	00	SP	040	32	20	@	100	64	40	'	140	96	60
SOH(^A)	001	1	01	!	041	33	21	A	101	65	41	a	141	97	61
STX(^B)	002	2	02	"	042	34	22	B	102	66	42	b	142	98	62
ETX(^C)	003	3	03	#	043	35	23	C	103	67	43	c	143	99	63
EOT(^D)	004	4	04	$	044	36	24	D	104	68	44	d	144	100	64
ENQ(^E)	005	5	05	%	045	37	25	E	105	69	45	e	145	101	65
ACK(^F)	006	6	06	&	046	38	26	F	106	70	46	f	146	102	66
BEL(^G)	007	7	07	'	047	39	27	G	107	71	47	g	147	103	67
BS (^H)	010	8	08	(	050	40	28	H	110	72	48	h	150	104	68
HT (^I)	011	9	09	)	051	41	29	I	111	73	49	i	151	105	69
LF (^J)	012	10	0A	*	052	42	2A	J	112	74	4A	j	152	106	6A
VT (^K)	013	11	0B	+	053	43	2B	K	113	75	4B	k	153	107	6B
FF (^L)	014	12	0C	,	054	44	2C	L	114	76	4C	l	154	108	6C
CR (^M)	015	13	0D	-	055	45	2D	M	115	77	4D	m	155	109	6D
SO (^N)	016	14	0E	.	056	46	2E	N	116	78	4E	n	156	110	6E
SI (^O)	017	15	0F	/	057	47	2F	O	117	79	4F	o	157	111	6F
DLE(^P)	020	16	10	0	060	48	30	P	120	80	50	p	160	112	70
DC1(^Q)	021	17	11	1	061	49	31	Q	121	81	51	q	161	113	71
DC2(^R)	022	18	12	2	062	50	32	R	122	82	52	r	162	114	72
DC3(^S)	023	19	13	3	063	51	33	S	123	83	53	s	163	115	73
DC4(^T)	024	20	14	4	064	52	34	T	124	84	54	t	164	116	74
NAK(^U)	025	21	15	5	065	53	35	U	125	85	55	u	165	117	75
SYN(^V)	026	22	16	6	066	54	36	V	126	86	56	v	166	118	76
ETB(^W)	027	23	17	7	067	55	37	W	127	87	57	w	167	119	77
CAN(^X)	030	24	18	8	070	56	38	X	130	88	58	x	170	120	78
EM (^Y)	031	25	19	9	071	57	39	Y	131	89	59	y	171	121	79
SUB(^Z)	032	26	1A	:	072	58	30	Z	132	90	5A	z	172	122	7A
ESC(^[)	033	27	1B	;	073	59	3B	[	133	91	5B	{	173	123	7B
FS (^\)	034	28	1C	<	074	60	3C	\	134	92	5C	|	174	124	7C
GS (^])	035	29	1D	=	075	61	3D	]	135	93	5D	}	175	125	7D
RS (^^)	036	30	1E	>	076	62	3E	^	136	94	5E	~	176	126	7E
US (^_)	037	31	1F	?	077	63	3F	_	137	95	5F	RUB	177	127	7F

APPENDIX B

ASSEMBLER DIAGNOSTIC MESSAGES

[To be written later]

APPENDIX C

ASSEMBLER SYNTAX SUMMARY

This appendix describes the complete assembler syntax in modified Backus-Naur Form (BNF). The following symbols are meta-symbols belonging to the BNF formalism:

<>	- Denotes a syntactic unit. Read as: the name enclosed in the angle brackets.
::=	- Definition of a syntactic unit. Read as: "is defined to be".
|	- Choose between syntactic units. This symbol can appear as part of the assembler language. Read as: "or".
{}	- Denotes possible repetition of the enclosed syntactic unit(s) zero or more times. Read as: "zero or more occurences of".
[]	- Optional syntactic unit(s). Read as: "optionally".
<space>	- Concatenation of two syntactic units. Read as: "followed by"
..	- Terminal symbol range. Read as: "through" (implied "or" (|) between each element in the range.

All other characters are part of the assembler language.

GENERAL ASSEMBLER SYNTAX SUMMARY:

```
<file>                ::= { <line> }

<line>                ::= <label> <opcode> <operands> <comment>

<label>               ::= <symbol>: | <empty>

<opcode>              ::= <symbol>  | <empty>

<operands>            ::= <operand> | <operand> { , <operand> }

<operand>             ::= <PROCESSOR_DEPENDENT> | <empty>

<comment>             ::= ; { <character> }

<expression>          ::= <simple_expression> | <simple_expression>
                          <relation_operator> <simple_expression>

<simple_expression>   ::= <term> | <sign> <term> |
                          <simple_expression> <simple_operator> <term>

<term>                ::= <factor> | <term> <term_operator> <factor>

<factor>              ::= <symbol> | <integer> | ( <expression> ) |
                          ~ <factor> | ! <factor> | ? <symbol> | *

<relation_operator>   ::= < | <= | = | <> | != | >= | >

<simple_operator>     ::= + | - | ^ | | | || |

<term_operator>       ::= * | / | % | & | && | << | >>

<symbol>              ::= <symbol_start> <symbol_body>

<symbol_start>        ::= <letter> | . | _

<symbol_body>         ::= { <letter> | <decimal_digit> | . | _ | $ }

<integer>             ::= <character_literal> | <number>

<character_literal>   ::= ' { <character_unit> } '

<character_unit>      ::= <character> | \ <character> |
                          \ ^ <character> | \ <number>

<number>              ::= <binary_number>  | <octal_number>       |
                          <decimal_number> | <hexadecimal_number> |
                          <int-real_number>
```

```
<binary_number>      ::= `B <binary_digit> { <binary_digit> } |
                         `b <binary_digit> { <binary_digit> }

<octal_number>       ::= `O <octal_digit> { <octal_digit> } |
                         `o <octal_digit> { <octal_digit> }

<decimal_number>     ::= `D <decimal_digit> { <decimal_digit> } |
                         `d <decimal_digit> { <decimal_digit> } |
                            <decimal_digit> { <decimal_digit> }

<hexadecimal_number> ::= `H <hexadecimal_digit> { <hexadecimal_digit> } |
                         `h <hexadecimal_digit> { <hexadecimal_digit> } |
                          $ <hexadecimal_digit> { <hexadecimal_digit> }

<int-real_number>    ::= `R <real> | `r <real> |
                         `T <real> | `t <real> |
                         `F <real> | `f <real> |
                         `L <real> | `l <real> |
                         `U <real> | `u <real>

<real>               ::= <whole_part> . <fractional_part> <exponent>

<whole_part>         ::= <decimal_digit> { <decimal_digit> }

<fractional_part>    ::= <decimal_digit> { <decimal_digit> } |
                         <empty>
<exponent>           ::= E <sign> <decimal_digit> { <decimal_digit> } |
                         e <sign> <decimal_digit> { <decimal_digit> } |
                         <empty>

<sign>               ::= + | - | <empty>

<string>             ::= " { <character_unit> } "

<empty>              ::=

<character>          ::= ASCII character set

<letter>             ::= A..Z a..z

<binary_digit>       ::= 0 | 1

<octal_digit>        ::= 0..7

<decimal_digit>      ::= 0..9

<hexadecimal_digit>  ::= 0..9 | A..F | a..f
```

DIRECTIVE SYNTAX SUMMARY:

```
<label>  .ABSADR <expression> [,<expression>]
<label>  .ADDR   <expression>
         .ALIGN  <expression> [,<expression>]
         .ALIGN  <symbol> [,<expression>]
         .BEGIN  [<symbol>]
         .COMMON <symbol> [,<expression>]
         .CONFIG <string>
<label>  .CONST  <expression>
         .DC.B   <expression> {,<expression>}
         .DC.W   <expression> {,<expression>}
         .DC.L   <expression> {,<expression>}
         .DC.F   <real> {,<real>}
         .DC.D   <real> {,<real>}
         .DISABL <symbol>  {,<symbol>}
         .DS.B   <expression> [,<expression>]
         .DS.W   <expression> [,<expression>]
         .DS.L   <expression> [,<expression>]
         .DS.S   <expression> [,<expression>]
         .ELSE
         .ENABLE <symbol> {,<symbol>}
         .END    [<symbol>]
         .ENDC
         .ENDS   [<symbol>]
         .EOF
         .ERROR  <string>
         .EXTERN <symbol> {,<symbol>}
         .FATAL  <string>
         .FILE   <integer> [,<string> [,<integer>]]
         .GLOBAL <symbol> {,<symbol>}
         .HEADER <string>
         .IF     <expression>
         .INCLD  <string>
         .LINE   [+|-] [<integer>]
         .LINKER <expression> {,<expression>}
         .LIST   <symbol> {,<symbol>}
         .LOCAL  <symbol> {,<symbol>}
         .MODULE [<symbol>]
         .NOLIST <symbol> {,<symbol>}
         .NONUSR <symbol> {,<symbol>}
         .PAGE
         .PRINT  <string>
         .PROCSS <symbol>
         .SECT   <integer> {,<symbol>}
         .SECT   <symbol>  {,<symbol>}
         .STRUCT <symbol>
         .USER   <symbol> {,<symbol>}
         .WARN   <string>
         .WEAK   <symbol> {,<symbol>}
```

APPENDIX D

ASSEMBLER FLOATING POINT FORMAT

[To be written later]

APPENDIX E

68000 INSTRUCTION SET SUMMARY

This appendix provides a summary of the 68000 instruction set. For detailed information, refer to the Motorola MC68000 16-bit Microprocessor User's Manual.

68000 INSTRUCTION SET SUMMARY

**** PRELIMINARY - For internal use only ****

```
ABCD    Dn,Dn
        -(An),-(An)
ADD     <ea>,Dn
        Dn,<ea>
ADDA    <ea>,An
ADDI    #<data>,<ea>
ADDQ    #<data>,<ea>
ADDX    Dn,Dn
        -(An),-(An)
AND     <ea>,Dn
        Dn,<ea>
ANDI    #<data>,<ea>
        #<data>,CCR
        #<data>,SR
ASL     Dn,Dn
        #<data>,Dn
        <ea>
ASR     Dn,Dn
        #<data>,Dn
        <ea>
BCC     <label>
BCS     <label>
BEQ     <label>
BF      <label>
BGE     <label>
BGT     <label>
BHI     <label>
BHS     <label>
BLE     <label>
BLO     <label>
BLS     <label>
BLT     <label>
BMI     <label>
BNE     <label>
BPL     <label>
BT      <label>
BVC     <label>
BVS     <label>
BCHG    Dn,<ea>
        #<data>,<ea>
BCLR    Dn,<ea>
        #<data>,<ea>
BRA     <label>
BSET    Dn,<ea>
        #<data>,<ea>
BSR     <label>
BTST    Dn,<ea>
        #<data>,<ea>
CHK     <ea>,Dn
CLR     <ea>
CMP     <ea>,Dn
```

```
CMPA    <ea>,An
CMPI    #<data>,<ea>
CMPM    (An)+,(An)+
DBCC    Dn,<label>
DBCS    Dn,<label>
DBEQ    Dn,<label>
DBF     Dn,<label>
DBGE    Dn,<label>
DBGT    Dn,<label>
DBHI    Dn,<label>
DBHS    Dn,<label>
DBLE    Dn,<label>
DBLO    Dn,<label>
DBLS    Dn,<label>
DBLT    Dn,<label>
DBMI    Dn,<label>
DBNE    Dn,<label>
DBPL    Dn,<label>
DBRA    Dn,<label>
DBT     Dn,<label>
DBVC    Dn,<label>
DBVS    Dn,<label>
DIVS    <ea>,Dn
DIVU    <ea>,Dn
EOR     Dn,<ea>
EORI    #<data>,<ea>
        #<data>,CCR
        #<data>,SR
EXG     Rn,Rn
EXT     Dn
ILLEGAL
JMP     <ea>
JSR     <ea>
LEA     <ea>,An
LINK    An,#<data>
LSL     Dn,Dn
        #<data>,Dn
        <ea>
LSR     Dn,Dn
        #<data>,Dn
        <ea>
MOVE    <ea>,<ea>
        <ea>,CCR
        <ea>,SR
        SR,<ea>
        An,USP
        USP,An
MOVEA   <ea>,An
MOVEM   <register_list>,<ea>
        <ea>,<register_list>
MOVEP   Dn,d(An)
```

```
        d(An),Dn
MOVEQ   #<data>,Dn
MULS    <ea>,Dn
MULU    <ea>,Dn
NBCD    <ea>
NEG     <ea>
NEGX    <ea>
NOP
NOT     <ea>
OR      <ea>,Dn
        Dn,<ea>
ORI     #<data>,<ea>
        #<data>,CCR
        #<data>,SR
PEA     <ea>
RESET
ROL     Dn,Dn
        #<data>,Dn
        <ea>
ROR     Dn,Dn
        #<data>,Dn
        <ea>
ROXL    Dn,Dn
        #<data>,Dn
        <ea>
ROXR    Dn,Dn
        #<data>,Dn
        <ea>
RTE
RTR
RTS
SBCD    Dn,Dn
        -(An),-(An)
SCC     <ea>
SCS     <ea>
SEQ     <ea>
SF      <ea>
SGE     <ea>
SGT     <ea>
SHI     <ea>
SHS     <ea>
SLE     <ea>
SLO     <ea>
SLS     <ea>
SLT     <ea>
SMI     <ea>
SNE     <ea>
SPL     <ea>
SRA     <ea>
ST      <ea>
SVC     <ea>
```

```
SVS     <ea>
STOP    #<data>
SUB     <ea>,Dn
        Dn,<ea>
SUBA    <ea>,An
SUBI    #<data>,<ea>
SUBQ    #<data>,<ea>
SUBX    Dn,Dn
        -(An),-(An)
SWAP    Dn
TAS     <ea>
TRAP    #<vector>
TRAPV
TST     <ea>
UNLK    An
```

APPENDIX F

OBJECT FILE AND LINKER LIMITATIONS

[To be written later]