SAGE COMPUTER

Assembler

Printed in the United States of America.

Disclaimer

This document and the software it describes are subject to change without notice. No warranty expressed or implied covers their use. Neither the manufacturer nor the seller is responsible or liable for any consequences of their use.

PREFACE

This manual describes the p-System Assembler. The assemblers which accompany this manual enable you to produce assembly language code for any one of the following processors:

LSI-11/PDP-11
Z80
6502
6800
8080
9900
6809
Z8
68000
8086/8087/8088

The assembly language programming details for these processors isn't covered in this manual. You should use a manual which describes the processor you are programming for along with this manual. (See Chapter 2.)

You can use the p-System to develop assembly language programs to provide:

1. Assembly language procedures to run under control of a host program; or

2. Stand-alone assembly language programs to use outside of the operating system's environment.

Preface

The assemblers, in conjunction with the system linker and some support programs, give you these capabilities.

You should use this reference manual in conjunction with the processor software manual that supports your machine. For information concerning differences from the processor's standard software syntax, refer to Chapter 2.

This manual is organized as follows. Chapter 1, "The Assembler," presents detailed information which applies to the assembler in general. Chapter 2, "Processor-Specific Information," provides information that is specific to each processor with a section for each assembler.

Appendix A describes the linker which combines separately assembled code files and can also link a high-level host program with assembled routines.

Appendix B covers the Compress utility. This utility allows you to produce a relocatable or absolute assembled object code file, enabling it to be run outside of the p-System environment.

Appendix C contains some typical 8086 routines. These examples demonstrate how to interface with Pascal program from assembly language.

Appendices D through M lists the assembler syntax errors for each processor.

Appendix N shows the value of NIL used by each processor.

TABLE OF CONTENTS

Table of Contents

CHAPTER 1

THE ASSEMBLER

INTRODUCTION

This chapter describes the p-System Assembler. It covers assembler-related concepts, assembler directives, and assembler operations. Other topics covered here include:

- Linking assembled routines with host compilation units.
- Assembled listings.
- Error messages.
- Sharing PME Resources.

Assembly Language Definition

An assembly language consists of symbolic names that can represent machine instructions, memory addresses, or program data. The main advantage of assembly language programming over machine coding is that programs can be organized in a more readable fashion, making them easier to understand.

An assembler translates an assembly language program, called source code, into a sequence of machine instructions, called object code. Assemblers can create either relocatable or absolute object code. Relocatable code includes information that allows a loader to place it in any available area of memory, while absolute code must be loaded into a specific area of memory. Symbolic addresses in programs that are assembled to relocatable object code are called relocatable addresses.

Assembly Language Applications

Using the p-System, you can develop:

1. Assembly language procedures to be used under a host program; or

2. Stand-alone assembly language programs for use in a different operating system environment.

GENERAL INFORMATION

Object Code Format

Byte Organization

A byte consists of eight bits. These bits may represent eight binary values or a single character of data. The bits may also represent a one-byte machine instruction or a number that is interpreted as either a signed two's complement number in the range of -128 to 127 or an unsigned number in the range of 0 to 255.

Word Organization

A word consists of 16 bits or 2 adjacent bytes in memory. A word may contain a one-word machine instruction, any combination of byte quantities, or a number that may be interpreted as either a signed two's complement number in the range of -32,768 to 32,767 or an unsigned number in the range of 0 to 65,535.

Source Code Format

Character Set

Use the following characters to construct source code:

- Uppercase and lowercase alphabetic characters: A through Z, a through z
- Numerals: 0 through 9
- Special symbols: | @ # $ % ^ & * ( ) < > ~ [ ] . , / ; : " ' + -= ? _
- Space (' ') character and tab character

Identifiers

Identifiers consist of an alphabetic character followed by a series of alphanumeric characters and/or underscore characters. The underscore character isn't significant. Only the first 8 characters of an identifier are significant.

Use identifiers in:

- Label and constant definitions.
- Machine instructions, assembler directives, and macro identifiers.
- Label and constant references.

```
          FormArray
          FORM_ARRAY
          formarray
     ... all denote the same item.
```

Predefined Symbols and Identifiers

Predefined identifiers are reserved by the assembler as symbolic names for machine instructions and registers. Don't use them as names for labels, constants, or procedures. Also, the dollar sign, "$," is predefined to specify the location counter. When used in an expression, the dollar sign represents the current value of the location counter in the program.

Character Strings

Write a character string as a series of ASCII characters delimited by double quotes. A string may contain up to 80 characters, but can't cross source lines. You can embed a double quote in a character string by entering it twice; for example, "This contains ""embedded"" double quotes." The assembler directive .ASCII requires a character string for its operand.

Strings also have limited uses in expressions.

Constants

Binary Integer Constants

Write a binary integer constant as a series of bits or binary digits (0 through 1) followed by the letter 'T'. The range of values is 0 to 1111111111111111, or 0 to 11111111, if a byte constant.

```
0T
01000100T
11101T
```

Decimal Integer Constants

Write a decimal integer word constant as a series of numerals (0 through 9) followed by a period. Its range of values is -32768 to 32767 as a signed two's complement number. As a byte constant, its range of values is -128 to 127 as a signed two's complement number or 0 to 255 as an unsigned number.

```
001.
256.
-4096.
```

Hexadecimal Integer Constants

Write a hexadecimal integer word constant as a series of up to four significant hexadecimal numerals (0 through 9, A through F) followed by the letter 'H'. The leading numeral of a hexadecimal constant must be a numeric character. The range of values is 0 to FFFF. These are examples of valid hexadecimal constants:

```
0AH
100H
0FFFEH ; leading zero is required here
```

Byte constants possess similar syntax, but can have at most two significant hexadecimal numerals, with a range of 0 to FF.

Octal Integer Constants

Write an octal integer word constant as a series of up to six significant octal numerals (0 through 7) followed by the letter 'Q'. Its range of values is 0 to 177777. Byte constants can have at most three significant octal numerals, with a range of 0 to 377.

```
17Q
457Q
177776Q
```

Default Integer Constants

If you don't follow an integer constant with 'T', '.', 'H', or 'Q', the integer will, by default, be of a certain type. This type is processor dependent. (See Chapter 2.)

Character Constants

Character constants are special cases of character strings; you may use them in expressions. The maximum length is two characters for a word constant and one character for a byte constant. Character constants are delimited by double quotes.

```
"A"
"BC"
"YA"
```

Assembly Time Constants

Write an assembly time constant as an identifier that the .EQU directive has assigned a constant value. (Refer to the section on "Data and Constant Definitions," presented later in this chapter.) Its value is completely determined at assembly time from the expression following the directive. You must define assembly time constants before you refer to them.

Expressions

Use expressions as symbolic operands for machine instructions and assembler directives. An expression can be:

- A label, which might refer to a defined address or an address further down in the source code (implying that the label is presently undefined), an externally referenced address, or an absolute address.

- A constant.

- A series of labels or constants separated by arithmetic or logical operators.

- The null expression, which evaluates to a constant of value 0.

Relocatable and Absolute

An expression containing more than one label is valid, only if the number of relocatable labels added to the expression exceeds the number of relocatable labels subtracted from the expression by zero or one. The expression result is absolute if the difference is zero, and relocatable if the difference is one. Don't use subexpressions that evaluate to relocatable quantities as arguments to a multiplication, division, or logical operation. Also, don't apply unary operators to relocatable quantities.

In relocatable programs, don't use absolute expressions as operands of instructions that require location-counter-relative address modes.

Linking and Restrictions

An expression may contain no more than one externally defined label, and its value must be added to the expression. An expression containing an external reference may not contain a forward-referenced label, and the relocation sum of any other relocatable labels in the expression must be equal to zero.

An expression may contain no more than one forward-referenced identifier. A forward-referenced identifier is assumed to be a relocatable label defined further down in the source code; you must define any other identifiers before using them in an expression. Also, don't place an externally defined label in an expression containing a forward-referenced label.

Arithmetic & Logical Operators

You may use the following operators in expressions:

- Unary operations:

  '+'plus

  '-'minus (two's complement negation)

  '~'logical not (one's complement negatioı

- Binary operations:

  '+'plus

  '-'minus

  '^'exclusive or

  '*'multiplication

  '/'signed integer division (DIV)

  '//'unsigned integer division (DIV)

  '%'unsigned remainder division (MOD)

  '|'bitwise OR

  '&'bitwise AND

- Use the following operators only with conditional assembly directives:

  '='equal

  '<>'not equal

- Use the following symbols as alternatives to the single-character definitions presented above. Occurrences of these alternative definitions require at least single blank characters as delimiters:

  .OR = '|'

  .AND = '&'

  .NOT = '~'

  .XOR = '^'

  .MOD = '%'

The assembler evaluates expressions from left-to-right; there is no operator precedence. All operations are performed on word quantities. Limit unary operators to constants and absolute addresses; and enclose subexpressions that contain embedded unary operators with angle brackets.

Subexpression Grouping

You may use angle brackets ('<' and '>') in expressions to override the left-to-right evaluation of operands. Subexpressions enclosed in angle brackets are completely evaluated before including them in the rest of the expression. Angle brackets are used instead of parentheses to group expressions. Using parentheses to group expressions doesn't generate an error but causes the assembler to interpret the expression as indirect addressing mode.

Examples

In the following examples of valid expressions, the default radix is decimal:

```
MARK+4        ; The sum of the value of
              ; identifier MARK plus 4
BILL-2        ; The result of subtracting 2 from
              ; the value of identifier BILL.
2-BARRY       ; The result of subtracting the
              ; value of identifier BARRY from 2.
              ; BARRY must be absolute.
3*2+MACRO     ; The sum of the value of
              ; identifier MACRO plus the
              ; product of 3 times 2.
DAVID+3*2     ; 2 times the sum of the
              ; identifier DAVID and 3.
              ; David must be absolute.
650/2-RICH    ; The result of dividing 650 by 2
              ; and subtracting the value of
              ; identifier RICH from the
              ; quotient.  RICH must be absolute
              ; Null expression: constant 0
-4*12+<6/2>   ; evaluates to -45 (decimal)
85+2+<-5>     ; evaluates to 82 (decimal)
0|1&<~0>      ; evaluates to 1
0 .OR 1 .AND <.NOT 0>  ; is the same expression
                       ; (result is 1)
```

Source Statement Format

An assembly language source program consists of source statements that may contain machine instructions, assembler directives, comments, or nothing (a blank line). Each source statement is defined as one line of a text file.

Label Field

The assembler supports the use of both standard labels and local (that is, reusable) labels. Begin the label field in the left-most character position of each source line. Macro identifiers and machine instructions must not appear in the start of the label field, but assembler directives and comments may appear there.

Standard Label Usage

A standard label is an identifier placed in the label field of a source statement. You may terminate it with an optional colon character, which isn't used when referencing the label. Only the first eight characters of the label are significant; the assembler ignores the rest. The underscore character isn't significant.

```
BIOS
L3456:          ; referenced as 'L3456'
The__Kind
LONG__label     ; last character is ignored
```

A standard label is a symbolic name for a unique address or constant; declare it only once in a source program. A label is optional for machine instructions and for many of the assembler directives. A source statement consisting of only a label is a valid statement; it effectively assigns the current value of the location counter to the label. This is equivalent to placing the label in the label field of the next source statement that generates object code. Labels defined in the label field of the .EQU directive are assigned the value of the expression in the operand field. (See the "Data and Constant Definitions" section, presented later in this chapter.)

Local Label Usage

Local labels allow source statements to be labeled for other instructions to reference, without taking up storage space in the symbol table. They can contribute to the cleanliness of source program design by allowing nonmnemonic labels to be created for iterative and decision constructs to use, thus reserving the use of mnemonic label names for demarking conceptually more important sections of code.

In local labels, you must place "$" in the first character position; the remaining characters must be digits. As in regular labels, only the first eight digits are significant. The scope of a local label is limited to the lines of source statements between the declaration of consecutive standard labels; thus, the jump to label $4 in the following example is illegal:

```
LABEL1
        ADC     AX, SI
$3      MOV     MEM, AX
        JC      $3          ; legal
        NOP
        JNC     $4          ; illegal
LABEL2
        ADC     AX, SI
$4      MOV     MEM, AX
```

You may define up to 21 local labels between 2 occurrences of a standard label. On encountering a standard label, the assembler purges all existing local label definitions; hence, all local label names may be redefined after that point. Don't use local labels in the label field of the .EQU directive. (See the "Data and Constant Definition" section in this chapter.)

Opcode Field

Begin the opcode field with the first nonblank character following the label field; or with the first nonblank character following the left-most character position when the label is omitted. Terminate it with one or more blanks. The opcode field can contain identifiers of the following types:

- Machine instruction.
- Assembler directive.
- Macro call.

Operand Field

Begin the operand field with the first nonblank character following the opcode field; terminate it with zero or more blanks. It can contain zero or more expressions, depending on the requirements of the preceding opcode.

Comment Field

You can precede the comment field with zero or more blanks, begin it with a semicolon (';'), and extend it to the end of the current source line. The comment field may contain any printable ASCII characters. It is listed on assembled listings and has no other effect on the assembly process.

Source File Format

You should use the system editor to produce assembly source files and save them as text files. You can construct a source file from the following entities:

- Assembly routines (procedures and functions).
- Global declarations.

Assembly Routines

A source file may contain more than one assembly routine. In this case, a routine ends when a routine delimiting directive occurs (for example, the start of the following routine). Each routine in a source file is a separate entity. It contains its own relocation information; and, during linking, a host program may refer to it individually.

Begin assembly routines with a .PROC, .FUNC, .RELPROC, or .RELFUNC directive. Terminate the last routine in the source file with the .END directive.

At the end of each routine, the assembler's symbol table is cleared of all but predefined and globally declared symbols, and the location counter (LC) is reset to zero.

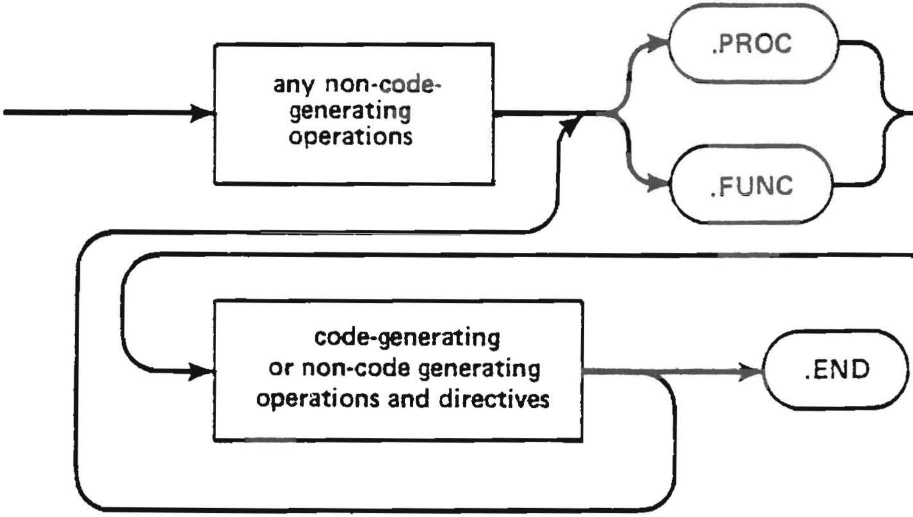

any non-code-generating operations
.PROC
.FUNC
code-generating or non-code generating operations and directives
.END

Figure 1-1. Structure of an Assembled Module

Global Declarations

An assembly routine may not directly access objects declared in another assembly routine, even if the routines are assembled in the same source file; however, sometimes it's desirable for a set of routines to share a common group of declarations. Therefore, the assembler allows global data declarations.

All subsequent assembly routines may reference any objects declared before a .PROC, .FUNC, .RELPROC, or .RELFUNC directive initially occurs in a source file. No code may be generated before the first procedure delimiting directive; hence, the "global" objects are limited to the noncode-generating directives (.EQU, .REF, .DEF, .MACRO, .LIST, etc.).

Absolute Sections

You'll often have to access absolute addresses in memory, regardless of where an assembly routine is loaded in memory. For instance, a program may need to access ROM routines. Absolute sections allow you to define labels and data space using the standard syntax and directives; this give you the added capability of specifying absolute (nonrelocatable) label addresses, starting at any location in memory.

You should initiate absolute sections with the directive .ASECT (for absolute section) and terminate them with the directive .PSECT (for program section, which is the default setting during assembly). When the .ASECT directive is encountered, the absolute section location counter (ALC) becomes the current location counter. Use the .ORG directive to set the ALC to any desired value. Label definitions are nonrelocatable and are assigned the current value of the ALC. The data directives .WORD, .BLOCK, and .BYTE cause the ALC—instead of the regular LC—to be incremented.

Data directives in an absolute section can't place initial values in the locations specified as they can when used in the program section. Thus, the absolute section serves as a tool for constructing a template of label-memory address assignments.

You may use the equate directive (.EQU) in an absolute section, but restrict the labels to being equated only to absolute expressions. The only other directives allowed to occur within an absolute section are .LIST, .NOLIST, .END, and the conditional assembly directives.

Absolute sections may appear as global objects.

The following is a simple example of an absolute section:

```
        .ASECT                  ; start absolute section
        .ORG ODFOOH             ; set ALC to DF00 hex
                                ; note - no data values assigned
                                ; label assignments below
DSKOUT  .BYTE                   ; DSKOUT = DF00
DSKSTAT .BYTE                   ; DSKSTAT = DF01
CONS    .WORD                   ; CONS = DF02
BLAGUE  .BLOCK 4                ; BLAGUE=DF04 (4 bytes)
REMOUT  .WORD                   ; REMOUT = DF08
OFFSET  .EQU          REMOUT+2  ; OFFSET = DF0A
        .PSECT
```

ASSEMBLER DIRECTIVES

Assembler directives (sometimes referred to as pseudo-ops) enable you to supply data to be included in the program and control the assembly process. Place assembler directives in the source code as predefined identifiers preceded by a period (.).

The following metasymbols are used in the syntax definitions for assembler directives:

- Special characters and items in capital letters must be entered as shown.
- Items within angle brackets (<>) are defined by you.
- Items within square brackets ([ ]) are optional.
- The word 'or' indicates a choice between two items.

- Items in lowercase letters are generic names for classes of items.

The following terms are names for classes of items:

b	The occurrence of one or more blanks.
comment	Any legal comment. (Refer to the "Comment Field" paragraph presented earlier in this chapter.)
expression	Any legal expression. (Refer to a prior paragraph entitled "Expressions.")
integer	Any legal integer constant as defined eariler in the section called "Constants."
label	Any legal label. (Refer to the "Label Field" paragraph earlier in this chapter.)
value	Any label, constant, or expression. Its default value is 0.
value list	A list of zero or more values delimited by commas.
identifier	A legal identifier as defined in a preceding paragraph entitled "Identifiers.")

idlist
A list of one or more identifiers delimited by commas.

id:integer list
A list of one or more identifier-integer pairs separated by a colon and delimited by a comma. The colon:integer part is optional; its default value is 1.

character string
Any legal character string. (See the paragraph "Character Strings," above.)

file identifier
Any legal name for a Pascal text file.

Example:

```
[<label>] [b] .ASCII b <character string> [<comment>]
```

This indicates that you may optionally include the label field, and that you must include a character string as an operand.

Small examples are included after each definition to supply you with a reference to the specific syntax of the directive.

Procedure-Delimiting Directives

Include at least one set of procedure-delimiting directives in every source program (including those intended for use as stand-alone code files). The assembler is used most frequently for assembling small routines intended to be linked with a host compilation unit. Use the directives .PROC and .FUNC to identify and delimit assembly language procedures; and .RELPROC and .RELFUNC to identify and delimit dynamically relocatable procedures. Dynamically relocatable procedures may reside in the code pool; they are subject to more of the system's memory management strategies. (For more detailed information about using these directives, refer to the section, "Program Linking and Relocation," presented later on in this chapter.)

.PROC Identifies the beginning of an assembly language procedure. The procedure is terminated when another delimiting directive occurs in the source file.

Form:

```
[b] .PROC b <identifier> [,<integer>] [<comment>]
```

<identifier> is the name associated with the assembly procedure.

<integer> indicates the number of parameter words passed to this routine. The default is 0.

Example:

```
 .PROC   DLDRIVE,2
```

.FUNC Identifies the beginning of an assembly language function. The host compilation unit expects a function to return a result on the top of the stack; otherwise, .FUNC is equivalent to the .PROC directive.

Form:

```
[b] .FUNC b <identifier>[,<integer>] [<comment>]
```

<identifier> is the name associated with the assembly procedure.

<integer> indicates the number of parameter words passed to this routine. The default is 0.

Example:

```
.FUNC  RANDOM
```

.RELPROC Identifies the beginning of a dynamically relocatable assembly language procedure. Such assembly procedures must be position-independent. (See the "Program Linking and Relocation" section in this chapter.) The procedure is terminated when another delimiting directive occurs in the source file.

Form:

```
[b] .RELPROC b <identifier> [,<integer>] [<comment>]
```

<identifier> is the name associated with the assembly procedure.

<integer> indicates the number of parameter words passed to this routine. The default is 0.

Example:

```
    .RELPROC   POOF,3
```

.RELFUNC

Identifies the beginning of a dynamically relocatable assembly language function. The host compilation unit expects this function to return a function result on top of the stack; otherwise, .RELFUNC is equivalent to the .RELPROC directive.

Form:

```
[b] .RELFUNC b <identifier>[,<integer>] [<comment>]
```

<identifier> is the name associated with the assembly function.

<integer> indicates the number of parameter words passed to this routine. The default is 0.

Example:

```
 .RELFUNC   POOOF
```

.END

Marks the end of an assembly source file.

Form:

```
 [<label>] [b] .END
```

Data and Constant Definitions

.ASCII

Converts character strings to a series of ASCII byte constants in memory. The bytes are allocated sequentially as they appear in the string. An identifier in the label field is assigned the location of the first character allocated in memory.

Form:

```
[<label>] [b] .ASCII b <character string> [<comment>]
```

<character string> is any string of printable ASCII characters delimited by double quotes.

Example:

```
.ASCII   "HELLO"
```

.BYTE

Allocates and initializes values in one or more bytes of memory. Values must be absolute byte quantities. The default value is zero. An identifier in the label field is assigned the location of the first byte allocated in memory.

Form:

```
[<label>] [b] .BYTE b [valuelist] [<comment>]
```

Example:

```
TEMP .BYTE 4; code would be 04 hex
TEMP1 .BYTE ; code would be 00 hex
```

.BLOCK

Allocates and initializes a block of consecutive bytes in memory. A byte value must be an absolute quantity. The default value is zero. An identifier in the label field is assigned the location of the first byte/word allocated.

Form:

```
[<label>] [b] .BLOCK b <length>[,<value>] [<comment>]
```

<length> is the the number of bytes to allocate with the initial value <value>.

Example:

```
   TEMP .BLOCK   4,6H
```

The output code would be:

```
06 06 06 06 ;four bytes with value 06 hex
```

.WORD

Allocates and initializes values in one or more consecutive words of memory. Values may be relocatable quantities. The default value is zero. An identifier in the label field is assigned the location of the first word allocated.

Form:

```
[<label>] [b] .WORD b <valuelist> [<comment>]
```

Example:

```
TEMP .WORD   0,2,,4
```

On a processor which has the least-significant byte first in a word, the output code would be:

```
0000
0200
0000   ; this is a default value.
0400
```

Example:

```
L1   .WORD  L2
```

The output code would be a word containing the address of the label L2.

.EQU

Associates a label with a particular value. Labels may be equated to an expression containing relocatable labels, externally referenced labels, and/or absolute constants. The general rule is that labels equated to values must be defined before use. The exception to this rule is for labels equated to expressions containing another label. Local labels may not appear in the label field of an equate statement.

Form:

```
<label> [b] .EQU b <value> [<comment>]
```

Example:

```
BASE        .EQU     R6
```

Location Counter Modification

These directives affect the value of the location counter (LC or ALC) and the location in memory of the code being generated.

.ORG

If used at the beginning of an absolute assembly program, .ORG initializes the location counter to <value>. Using .ORG anywhere else generates zero bytes until the value of the location counter equals <value>.

Form:

```
[b] .ORG b <value> [<comment>]
```

Example:

```
.ORG    1000H
```

.ALIGN Outputs sufficient zero bytes to set the location counter to a value that is a multiple of the operand value.

Form:

```
[b] .ALIGN b <value> [<comment>]
```

Example:

```
 .ALIGN   2
```

This aligns the LC to a word boundary.

Listing Control Directives

Use these directives to control the format of the assembled listing file generated by the assembler. These directives don't generate code, and their source lines don't appear on assembled listings. (For a more detailed description of an assembled listing, refer to the "Assembler Output" paragraph, presented later in this chapter.)

.TITLE

Changes the title printed on the top of each page of the assembled listing. The title may be up to 80-characters long. The assembler changes the title to 'SYMBOLTABLE DUMP' when printing a symbol table; the title reverts back to its former value after the symbol table is printed. The default value for the title is ' '.

Form:

```
[b] .TITLE b <character string> [<comment>]
```

Example:

```
.TITLE "MACROS"
```

.ASCIILIST

Prints all bytes the .ASCII directive generates in the code field of the list file, creating multiple lines in the list file if necessary. Assembly begins with an implicit .ASCIILIST directive.

Form:

```
[b] .ASCIILIST  [<comment>]
```

Example:

```
.ASCIILIST
```

.NOASCIILIST Limits the printing of data the .ASCII directive generates to as many bytes as will fit in the code field of one line in the list file.

Form:

```
[b] .NOASCIILIST  [<comment>]
```

Example:

```
.NOASCIILIST
```

.CONDLIST Lists source code contained in the unassembled sections of conditional assembly directives.

Form:

```
[b] .CONDLIST  [<comment>]
```

Example:

```
.CONDLIST
```

.NOCONDLIST Suppresses the listing of source code contained in the unassembled sections of conditional assembly directives. Assembly begins with an implicit .NOCONDLIST directive.

Form:

```
[b] .NOCONDLIST  [<comment>]
```

Example:

```
.NOCONDLIST
```

.NOSYMTABLE Suppresses the printing of a symbol table after each assembly routine in an assembled listing.

Form:

```
[b] .NOSYMTABLE  [<comment>]
```

Example:

```
.NOSYMTABLE
```

.PAGEHEIGHT Controls the number of lines printed in an assembled listing between page breaks. Assembly begins with an implicit .PAGEHEIGHT 59 directive.

Form:

```
[b] .PAGEHEIGHT <integer>  [<comment>]
```

Example:

```
.PAGEHEIGHT    40.
```

.NARROWPAGE Limits the width of an assembled listing to 80 columns. The symbol table is printed in a narrow format, source lines are truncated to a maximum of 49 characters, and title lines on the page headers are truncated to a maximum of 40 characters.

Form:

```
[b] .NARROWPAGE  [<comment>]
```

Example:

```
.NARROWPAGE
```

.PAGE

Continues the assembled listing on the next page by sending an ASCII form feed character to the assembled listing.

Form:

```
[b] .PAGE
```

Example:

```
.PAGE
```

.LIST

Enables output to the list file, if a listing isn't already being generated. You can use .LIST and .NOLIST to examine certain sections of source and object code without creating an assembled listing of the entire program. Assembly begins with an implicit .LIST directive.

Form:

```
[b] .LIST
```

Example:

```
.LIST
```

.NOLIST

Suppresses output to the list file, if it isn't already off.

Form:

```
[b] .NOLIST
```

Example:

```
.NOLIST
```

.MACROLIST

Specifies that all subsequent macro definitions have their macro bodies printed when they are called in the source program. Assembly begins with an implicit .MACROLIST directive. The section called "Macro Language," presented later in this chapter, gives a detailed description of macro language.

Form:

```
[b] .MACROLIST
```

Example:

```
.MACROLIST
```

.NOMACROLIST Specifies that all subsequent macro definitions won't have their macro bodies printed when they are called in the source program. Only the macro identified and parameter list are included in the listing.

Form:

```
[b] .NOMACROLIST
```

Example:

```
.NOMACROLIST
```

.PATCHLIST Lists occurrences of all back patches of forward-referenced labels in the list file. Assembly begins with an implicit .PATCHLIST directive. For a detailed description of back patches, refer to the paragraph, "Forward References," in the section called, "Assembler Output," presented later in this chapter.

Form:

```
[b] .PATCHLIST
```

Example:

```
.PATCHLIST
```

.NOPATCHLIST Suppresses the listing of back patches of forward references.

Form:

```
[b] .NOPATCHLIST
```

Example:

```
.NOPATCHLIST
```

Program Linkage Directives

Linking directives enable communication between separately assembled and/or compiled programs. Later in this chapter, the section called "Program Linking and Relocation" has a detailed description of program linking.

.CONST Allows the assembly procedure to access globally declared constants in the host compilation unit.

Form:

```
[b] .CONST b <idlist> [<comment>]
```

Each <ID> is the name of a global constant declared in the host.

Example:

```
.CONST   LENGTH
```

.PUBLIC

Allows an assembly language routine to reference variables declared in the global data segment of the host compilation unit.

Form:

```
[b] .PUBLIC b <idlist> [<comment>]
```

Each <ID> is the name of a global variable declared in the host.

Example:

```
.PUBLIC  I,J,LENGTH
```

.PRIVATE Allows an assembly language routine to store variables, which only the assembly language routine can access, in the global data segment of the host compilation unit.

Form:

```
[b] .PRIVATE b <id:integer list> [<comment>]
```

Each <ID> is treated as a label defined in the source code. <integer> determines the number of words of space allocated for <ID>.

Example:

```
.PRIVATE   PRINT,BARRAY:9
```

.INTERP Allows an assembly language procedure to access code or data in the p-code PME. .INTERP is a predefined symbol for a processor-dependent location in the resident PME code; you may use offsets from this base location to access any code in the PME. To use this feature correctly, you must know the PME's jump vector for this location. .INTERP is generally restricted to systems applications.

Form:

```
valid when used in <expression>
```

Example:

```
ERR .EQU 12                ; hypothetical
                           ; routine offset
BOMB .EQU .INTERP+ERR
       JMP   BOMB
```

.REF Provides access to one or more labels defined in other assembly language routines.

Form:

```
[b] .REF <idlist> [<comment>]
```

Example:

```
.REF    SCHLUMP
```

.DEF Makes one or more labels, to be defined in the current routine, available for other assembly language routines to reference.

Form:

```
[b] .DEF <idlist> [<comment>]
```

Example:

```
.DEF    FOON,YEEN
```

Conditional Assembly Directives

A detailed description of conditional assembly features is presented later in this chapter in a section called, "Conditional Assembly."

.IF Marks the start of a conditional section of source statements.

Form:

```
[b] .IF b <expression> [ = or <> <expression>] [<comment>]
```

Example:

```
.IF  DEBUG
```

.ENDC Marks the end of a conditional section of source statements.

Form:

```
[b] .ENDC  [<comment>]
```

Example:

```
.ENDC
```

.ELSE Marks the start of an alternative section of source statements.

Form:

```
[b] .ELSE  [<comment>]
```

Example:

```
.ELSE
```

Macro Definition Directives

A detailed description of macro language is presented later in this chapter in the section, "Macro Language."

.MACRO Indicates the start of a macro definition.

Form:

```
[b] .MACRO b <identifier>  [<comment>]
```

<identifier> calls the macro being defined.

Example:

```
.MACRO  ADDWORDS
```

.ENDM Marks the end of a macro definition.

Form:

```
[b] .ENDM [<comment>]
```

Example:

```
.ENDM
```

Miscellaneous Directives

.INCLUDE Causes the assembler to start assembling the file named as an argument of the directive; when the end of this file is reached, assembling resumes with the source code that follows the directive in the original file. This feature is useful for including a file of macro definitions or for splitting up a source program too large to be edited as a single text file. You can't use .INCLUDE in: (1) an included source file (that is, nested use of the directive); and (2) in a macro definition.

Form:

```
[b] .INCLUDE b <file identifier> [<comment>]
```

At least one blank character must separate the comment field of the .INCLUDE directive from the file identifier.

Example:

```
.INCLUDE  MYDISK:MACROS
```

.ABSOLUTE Causes the following assembly routine to be assembled without relocation information. Labels become absolute addresses and label arithmetic is allowed in expressions. .ABSOLUTE is valid only before the first procedure delimiting directive occurs. Don't use .ABSOLUTE when the assembled routine is to be called from a high-level host. (Refer to the "Program Linking and Relocation" section, presented later in this chapter, for a detailed description of abolute code files.)

Form:

```
[b] .ABSOLUTE [<comment>]
```

Example:

```
.ABSOLUTE
```

.ASECT Specifies the start of an absolute section. For a detailed description of ".ASECT," refer to the paragraph called "Absolute Sections," presented earlier in this chapter.

Form:

```
[b] .ASECT [<comment>]
```

Example:

```
.ASECT
```

.PSECT Specifies the start of a program section and terminates an absolute section. (Refer to the "Absolute Sections" paragraphs, presented earlier.)

Form:

```
[b] .PSECT [<comment>]
```

Example:

```
.PSECT
```

.RADIX Sets the current default radix to the value of the operand. Allowable operands are: 2 (binary), 8 (octal), 10 (decimal), and 16 (hexadecimal). The default radix of an integer constant is processor-specific. (See Chapter 2.)

Form:

```
[b] .RADIX <integer> [<comment>]
```

Example:

```
.RADIX 10     ; decimal
              ; default radix
```

CONDITIONAL ASSEMBLY

Use conditional assembly directives to selectively exclude or include sections of source code at assembly time. Initiate conditional sections with the .IF directive and terminate them with the .ENDC directive. They may contain the .ELSE directive. Use conditional expressions to control inclusion of conditional sections. Conditional sections may contain other conditional sections.

When the assembler encounters an .IF directive, it evaluates the associated expression to determine the condition value. If the condition value is false, the source statements following the directive are discarded until a matching .ENDC or .ELSE is reached. If you use the .ELSE directive in a conditional section, source code before the .ELSE is assembled if the condition is true; and source code after the .ELSE is assembled if the condition is false.

Overall syntax for a conditional section (using the meta language described earlier in the "Assemblers Directives" paragraph) is as follows:

```
.IF   <conditional expression>
      <source statements>
[.ELSE
      <source statements>]
.ENDC
```

Conditional Expressions

A conditional expression can take one of two forms: a single expression or comparison of two character strings or expressions. The first form is considered false if it evaluates to zero; otherwise, it's considered true. The second form of conditional expression compares for equality or inequality (indicated by the symbols '=' and '<>', respectively).

Example:

```
.IF LABEL1-LABEL2 ; arithmetic expression
                  ; This code is assembled only if
                  ; difference is not zero
  .IF %1="STUFF"  ; comparison expression
                  ; This code is assembled only if
                  ; outer condition is true and
                  ; text of first macro parameter
                  ; is equal to "STUFF".
  .ENDC           ; terminate nested section
                  ; This code is assembled if outer
                  ; condition is true
.ELSE
                  ; This code is assembled if first
                  ; condition is false
.ENDC             ; terminate outer section
```

MACRO LANGUAGE

The assembler allows you to use a macro language in source programs. This enables you to associate a set of source statements with an identifying symbol. When the assembler encounters this symbol (known as a macro identifier) in the source code, it substitutes the corresponding set of source statements (known as the macro body) for the macro identifier, and assembles the macro body as if it had been included directly in the source program. You can use carefully designed set of macro definitions in all source programs to simplify developing assembly language routines.

In addition, you can enhance the macro language by including a mechanism for passing parameters (known as macro parameters) to the macro body while it is being expanded. This allows a single macro definition to be used for an entire class of subtasks.

Here is a simple example:

```
                ; macro definition...
.MACRO  STRING  ; macro identifier is
                ;   STRING
                ; Macro Body:
                ; %1 and %2 are
                ;   parameter
                ;   declarations
.BYTE   %2      ; 2nd parameter is
                ;   length byte
.ASCII  %1      ; 1st parameter is
                ;   argument
.ENDM           ; end macro definition
```

Further down in the source code...

```
STRING  "WRITE",5.         ; 1st macro call
                           ; parameters are
                           ;   '"WRITE"'
                           ;   and '5.'
STRING  "TYPE SPACE",10.   ; 2nd macro call
                           ; parameters are
                           ; '"TYPE SPACE"'
                           ;   and '10.'
```

This is what gets assembled...

```
.BYTE   5. ; data string declarations
.ASCII  "WRITE"
.BYTE   10.
.ASCII  "TYPE SPACE"
```

Macro Definitions

You may place macro definitions anywhere in a source program and delimit them with the directives .MACRO and .ENDM. The macro identifier must be unique to the source program, except when you redefine a predefined machine instruction name as a macro identifier. You shouldn't include a macro definition within another macro definition. However, you may include macro calls. You may nest macro calls to a maximum depth of five levels. A macro definition must occur before any calls to that macro are assembled, but macro calls may be forward-referenced within the bodies of other macro definitions.

Macro Calls

You can place macro calls anywhere in a source program that code may be generated. A macro call consists of a macro identifier followed by a list of parameters. Delimit the parameters with commas and terminate them with a carriage return or semicolon. Upon encountering a macro call, source code is read from the text of the corresponding macro body. Macro parameters within the macro body are substituted with the text of the matching parameter listed after the macro identifier that initiated the call.

Parameter Passing

You may reference macro parameters in a macro body by using the symbol '%n' in an expression, where 'n' is a single nonzero decimal digit. Upon scanning this symbol, the assembler replaces it with the text of the n'th macro parameter. Note that macro parameters are <u>not</u> expanded within the quotes of an ASCII data string.

Three cases are possible:

1. The parameter exists—the substitution is made.

2. The n'th parameter doesn't exist in the parameter list being checked (less than n parameters were passed); a null string is substituted.

3. Another symbol of the form '%m' is encountered in the parameter list. If nested macro calls exist, the text of the m'th parameter at the next higher level of macro nesting is substituted; otherwise, the symbol itself is assembled.

You must pass parameters without leading and trailing blanks. You may pass all assembly symbols, except macro calls, as parameters.

The following is an example of parameter passing in macros:

```
.MACRO DOS
UNO     %2,UN
SAR     %1
.ENDM
.MACRO  UNO
MOV     %1,%2
SAL     %2
.ENDM
```

In a program, the macro call...

```
DOS   TROIS,DEUX
```

assembles as...

```
MOV   DEUX,UN ; UNO got UN directly,
              ; but had to use DOS's
              ; 2nd param
SAL   UN
SAR   TROIS   ; DOS used its own 1st
              ; param
```

Scope of Labels in Macros

A problem arises in using macro language when the definition of a macro body requires you to use branch instructions and, thus, have labels. Declaring a regular label in a macro body is incorrect if the macro is called more than once, because the label would be substituted twice into the source program and flagged by the assembler as a previously defined label. You can use location-counter-relative addressing, but this is prone to errors in nontrivial applications. The best solution is to generate labels that are local to the macro body; the assembler's local labels can do this.

Local label names you declare in a macro body are local to that macro; thus, a section of code that contains a local label $1 and a macro call whose body also has the local label $1, assembles without errors. (Contrast this with what happens when two occurrences of $1 fall between two regular labels.) This feature allows you to use local labels freely in macros without conflicting with the rest of the program.

NOTE: Remember that a maximum of 21 local labels can be active at any instant.

Local Labels as Macro Parameters

Passing local labels as parameters has a special property. Unlike other macro parameters, local labels aren't passed as uninterpreted text. The scope of a local label passed in a macro call doesn't change as it is passed through increasing levels of macro nesting, regardless of naming conflicts along the way. One use of this property is passing an address to a macro that simulates a conditional branch instruction.

The following is an example of passing local labels as macro parameters:

```
        .MACRO EIN
        JE     $1
        JNE    %1
$1
        .ENDM
```

In a program, the code...

```
TWIE
    SUB   ICHI,NI
    EIN   $1
    RET
$1
    JMP   SAN
```

assembles as...

```
TWIE
    SUB   ICHI,NI
    JE    $1 ; this references macro
             ;   local label
    JNE   $1 ; this references
             ;   outside $1
$1           ; macro local label
    RET
$1           ; outside $1
    JMP   SAN
```

PROGRAM LINKING & RELOCATION

The assembler produces either absolute or relocatable object code that you may link, as required, to create executable programs from separately assembled or compiled modules. (The linker is described in Appendix A.)

Program linking directives generate information the system linker requires to link modules. Some of the advantages of linking are:

- You can divide long programs into separately assembled modules to avoid a long assembly, reduce the symbol table size, and encourage modular programming techniques.
- You can enable other linked modules to share modules.
- You can add utility modules to the system library for a large number of programs to use as external procedures.
- Programs can call assembly language procedures directly.

The assembler generates linker information in both relocatable and absolute code files. The system linker accesses this information during linking and removes it from the linked code file.

Relocatable code includes information that allows a loader program to place it anywhere in memory, while absolute (also called core image) code files must be loaded into a specific area of memory to execute properly. Assembly procedures running in the p-System environment must always be relocatable; the system PME performs loading and relocation at a load address the state of the system determines.

Absolute code won't run under the p-System environment (under which high-level programs must run). However, relocatable code *can* run under the p-System. Code segments containing statically relocatable code remain in main memory throughout the lifetime of their host program (or unit) and are position-locked for that duration. Thus, relocatable code may maintain and reference its own internal data space (or spaces). In addition, statically relocatable code saves some space because its relocation information doesn't have to remain present throughout the life of the program.

The directives .PROC and .FUNC designate statically relocatable routines; .RFLPROC and .RELFUNC designate dynamically relocatable routines. Code segments that contain dynamically relocatable code don't necessarily occupy the same location in memory throughout their host's lifetime, but are maintained in the code pool along with other dynamic segments (mostly p-code); they may be swapped in and out of main memory while the host program (or unit) is running. Thus, dynamically relocatable code shouldn't maintain internal data spaces if that data must last across calls to the assembled routine. Data that is meant to last across different calls to the assembly routine must be kept in your host data segments by using .PRIVATEs and .PUBLICs.

1. Data space is embedded in the code, but the code doesn't move:

```
.PROC  FOON
.WORD  SPACE
  ...
.END
```

2. The code moves, but data space is allocated in the host compilation unit's global data segment:

```
.RELPROC  FOON
.PRIVATE  SPACE
  ...
.END
```

3. **Caution**: The code may move and since the data is embedded in the code, the data may be destroyed between calls to the routine:

```
.RELPROC  FOON
.WORD     SPACE
  ...
.END
```

Code pool management is described in the Internal Architecture Reference Manual.

Program Linking Directives

This section describes the overall use of linking directives. All linking of assembly procedures involves word quantities; it isn't possible to externally define and reference data bytes or assembly time constants. Arguments of these directives must match the corresponding name in the target module (a lowercase Pascal identifier will match an uppercase assembly name, and vice versa) and must not have been used before their appearance in the directive. The assembler treats all subsequent references to the arguments as special cases of labels. The linker and/or PME resolves these external references by adding the link-time and run-time offsets to the existing value of the word quantity in question. Thus, any initial offsets generated by including of external references and constants in expressions are preserved.

Host Communication Directives

Use the directives .CONST, .PUBLIC, and .PRIVATE to allow constants and data to be shared between an assembly procedure and its host compilation unit. For examples, see the "Program Linkage Directives" paragraph in the "Assembler Directives" section, presented previously in this chapter.

.CONST Allows an assembly procedure to access globally declared constants in the host compilation unit. The linker patches all references to arguments of .CONST with a word containing the value of the host's compile-time constant.

.PUBLIC Allows an assembly procedure to access globally declared variables in the host compilation unit. Note: You can use this directive to set up pointers to the start of multi-word variables in host programs; it isn't limited to single word variables.

.PRIVATE Allows an assembly procedure to declare variables in the global data segment of the host compilation unit that the host can't access. The optional length attribute of the arguments allows multi-word data spaces to be allocated; the default data space is one word.

External Reference Directives

Use the directives .REF and .DEF to allow separately assembled modules to share data space and subroutines. (For examples, refer ahead, in this chapter, to the paragraph, "Example of Linking to Pascal.")

.DEF Declares a label to be defined in the current program as accessible to other modules. One restriction is imposed on its use—you can't .DEF a label that has been equated to a constant expression or used in an expression containing an external reference.

.REF Declares a label existing and .DEF'ed in another module to be accessible to the current program.

Program Identifier Directives

Use the directives .PROC, .FUNC, .RELPROC, .RELFUNC, and .END as delimiters for source programs. You must include at least one pair of delimiting directives in every source program (relocatable or absolute).

The identifier argument of the .PROC or .RELPROC directive serves two functions: the linker can reference it when linking an assembly procedure to its corresponding host, and other modules can reference it as an externally declared label. Specifically, the declaration:

```
 .PROC FOON    ; procedure heading
```

in a source program—is functionally equivalent in the assembly environment to the following statements:

```
  .DEF FOON    ; FOON may be externally
               ;   referenced
FOON           ; declare FOON as a label
```

This feature allows an assembly module to call other (external and eventually linked in) assembly modules by name. Use the .FUNC and .RELFUNC directives when linking an assembly function directly to a host program; they aren't intended for uses that involve linking with other assembly modules.

The linker references the optional integer argument after the procedure identifier. It does this to determine if the number of parameter words passed by the host's external procedure declaration matches the number specified by the assembly procedure declaration. It isn't relevant when linking with other assembly modules.

Linking Program Modules

For information on linking with the p-System's other high-level languages, refer to the documentation on that particular language.

Linking with a Pascal Program

External procedures and functions are assembly language routines declared in Pascal programs. To run Pascal programs with external declarations, you must compile the Pascal program, assemble the external procedure or function, and link the two code files.

A host program declares a procedure to be external in a syntactically similar manner to a forward declaration. The procedure heading is given (with parameter list, if any), followed by the keyword 'EXTERNAL'. Calls to the external procedure use standard Pascal syntax. The compiler checks that calls to the external procedure agree in type and number of parameters with the external declaration. All parameters are pushed on the stack in the order of their appearance in the parameter list of the declaration; thus, the right-most parameter in the declaration will be on the top of the stack. (For a detailed description of parameter passing conventions, refer to the next section, called "Parameter Passing Conventions.")

You must make sure that the assembly language routine maintains the integrity of the stack. This includes removing all parameters passed from the host, preserving the SS and SP registers, and making a clean return to the Pascal run-time environment using the return address originally passed to it. If you don't do this, a potentially fatal system crash can occur, as assembly routines are outside the scope of the Pascal environment's run-time error facilities. (For a detailed description of Pascal/assembly language protocols, refer ahead, in this chapter, to the section, "Sharing PME Resources.")

An external function is similar to a procedure, but has some differences that affect the way that parameters are passed to and from the Pascal run-time environment. First, the external function call pushes one, two, or four words on the stack before any parameters have been pushed. Two or four words are pushed for a function of type real, depending upon the real size that you are using. One word is pushed for all other types of functions. The words are part of the p-machine's function calling mechanism and are irrelevant to assembly language functions; the assembly routine must throw these away before returning the function's result. Second, the assembly routine must push the proper number of words (2 or 4 for type real; 1, otherwise) containing the function result onto the stack before passing control back to the host. A subsequent section, "Sharing PME Resources," describes a clean way to do all of this without ever using an actual POP or PUSH operation.

Parameter Passing Conventions

The ability of external procedures to pass any variables as parameters gives you complete freedom to access the machine-dependent representations of machine-independent host data structures. However, with this freedom comes the responsibility of respecting the integrity of the p-machine run-time environment. To give you a better understanding of the host/assembly language interface, this section enumerates the p-machine's parameter passing conventions for all data types; it doesn't actually describe data representations. For examples of parameter passing between Pascal and external procedures, see Appendix C.

You may pass parameters by either value or by reference (variable parameters). To manipulate assembly language, variable parameters are handled in a more straightforward fashion than value parameters.

The word "tos" is used in the following sections as an abbreviation for "top of stack."

Variable Parameters

You should reference variable parameters through a one-word pointer passed to the procedure. Thus, the procedure declaration:

```
procedure pass_by_name (var i,j : integer;
          var q : some_type); external;
```

would pass three one-word pointers on the stack; tos would be a pointer to q, followed by pointers to j and i.

A Pascal external procedure declaration is allowed to contain variable parameters lacking the usual type declaration; this enables you to pass variables of different Pascal types through a single parameter to an assembly routine. Untyped parameters aren't allowed in normal Pascal procedure declarations.

The procedure declaration:

```
procedure untyped_var (var i; var q:
          some_type); external;
```

contains the untyped parameter 'i'.

Value Parameters

Value parameters are handled according to their data type. Pass the following types by pushing copies of their current values directly on the stack: boolean, char, integer, real, subrange, scalar, pointer, set, and long integer. Other sections of this manual describe the number of words per data type and the internal data format. For instance, the declaration:

```
procedure pass_by_value (i : integer; r : real);
                          external;
```

would pass two words or four words on "tos" containing the value of the real variable 'r' followed by one-word containing the value of the integer variable 'i'.

Pass variables of type record and array by value in the same manner as variable parameters; pointers to the actual variable are pushed onto the stack. Pass variables of type PACKED ARRAY OF CHAR and STRING by value with a segment pointer (described in next section).

Value parameters which are passed using pointers should be copied into a local data space for processing. The original copy of a value parameter should never be altered.

String and Byte Array Parameters

When a string or byte array is passed as a value parameter to an assembly language routine, a "segment pointer" is passed on the stack. A segment pointer consists of two words. The first word (tos) contains either NIL or a pointer to a segment environment record. (This is determined by whether the parameter is a constant or variable.)

If the first word is NIL, then the second word (at tos-1) points to the parameter.

If the first word isn't NIL, then to find the parameter it is necessary to chain through some records. The first word (tos) is a pointer and the second word (tos-1) is an offset. The first word points to a segment environment record (EREC). The third word of that record contains a pointer to a SIB (Segment Information Block). If the first word of the SIB is NIL, then the second word is a pointer to the base of the segment where the parameter resides. If the first word of the SIB isn't NIL, then it points to a Pool Descriptor. The contents of the first two words of the Pool Descriptor plus the contents of the second word of the SIB is a pointer to the base of the segment where the parameter resides. (Note that the first word of the Pool Descriptor contains the 16 most-significant bits, and the second word contains the 16 least-significant bits.

Each word, however, is in the natural byte sex of the host processor. On processors that address the least-significant byte first, this means that the bytes are in this order: second most-significant, first most-significant, fourth most-significant, third most-significant.)

The exact location of the parameter is given by the segment base plus the contents of the second word on the stack (tos-1), which is an offset into the code segment.

The following figure illustrates this accessing scheme. Note that cases 1 and 2 produce a 16-bit address which is relative to the base of the p-System Stack/Heap area. Case 3, however, produces a 32-bit absolute physical address. (For a full description of these mechanisms, refer to the _Internal Architecture Reference Manual_.)

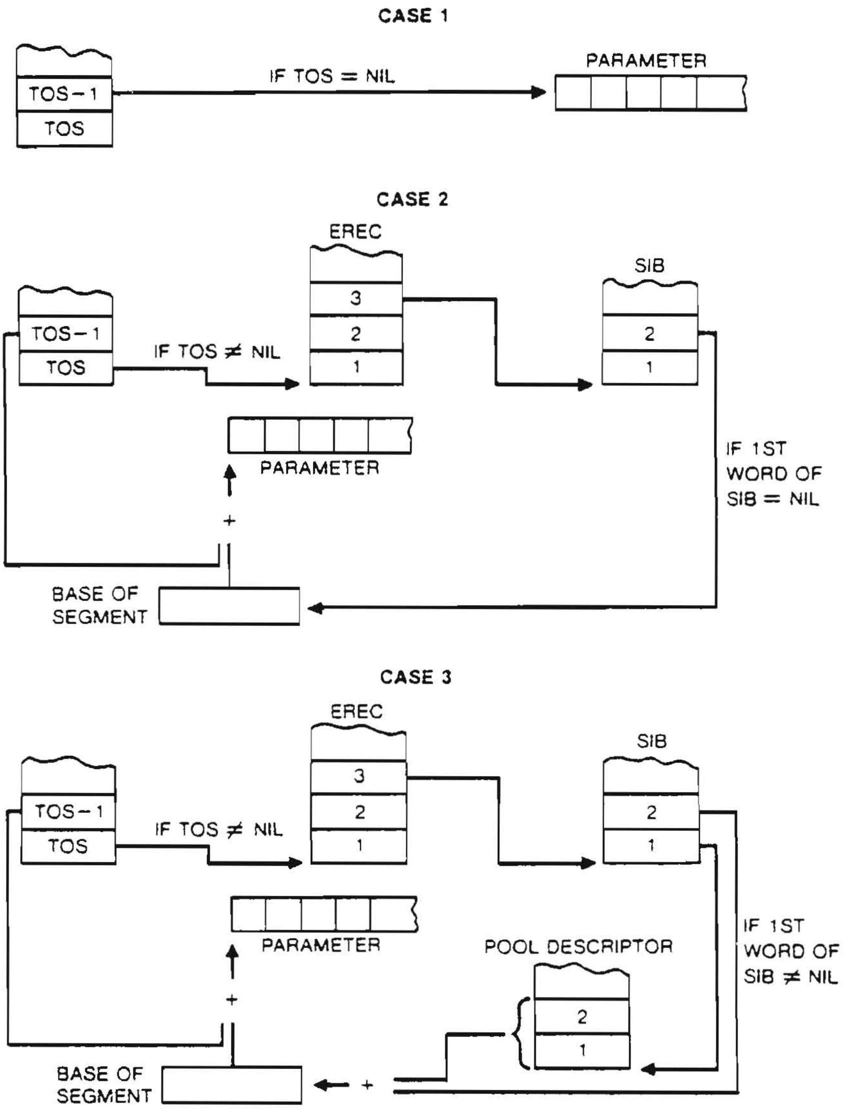


Figure 1-2. String and Byte
Array Parameters

Example of Linking to Pascal

Note that in the following example the host program passes control to the beginning of an assembly procedure whether or not machine instructions are there. Therefore, all data sections you allocate in the procedure must either: (1) occur after the end of the machine instructions; or (2) have a jump instruction branch around them.

```
PROGRAM EXAMPLE; { Pascal host program }
const size = 80;
var   i,j,k:  integer;
      lst1:   array [[0..9] of char;
      { PRT and LST2 get allocated here }
  procedure do_nothing; external;
  function null_func(xxyxx,z:integer)
                    :integer; external;
begin
  k := 45;
  do_nothing;
  j := null_func(k,size);
end.
.PROC      DONOTHING      ; underscores are not
                          ; significant in Pasal
.CONST     SIZE           ; can get at size
                          ; constant in host
.PUBLIC    I,LST1         ; and also these two
                          ; global vars
.DEF       TEMP1          ; this allows NULLFUNC
                          ; to get at temp1
                          ; code starts here...
POP        RETURN1        ; return addr pushed on
POP        RETURN2        ; stack
                          ; does nothing
PUSH       RETURN2        ; set up stack for
PUSH       RETURN1        ; return

RETL
                          ; data area
RETADR   .EQU  TEMP1
TEMP1    .WORD
RETURN1  .WORD
RETURN2  .WORD
                          ; end of procedure
                          ; DONOTHING
.FUNC      NULLFUNC,2
.PRIVATE   PRT,LST2:9     ; 10 words of
                          ; private data
.REF       TEMP1          ; references data temp
                          ; in DONOTHING
                          ; code starts here
```

```
POP       RETURN1       ; save return address
POP       RETURN2
POP       PRT           ; get parameter 'z'
POP       LST2+4        ; get parameter 'xxyxx'
POP       TEMP1         ; toss 1 word of junk
                        ; (funtion return area)

                        ; performs null action
PUSH      LST2+4        ; return xxyxx as
                        ; result
PUSH      RETURN2       ; restore subr link
PUSH      RETURN1
RETL                    ; return to calling
                        ; program
                        ; data starts here
RETURN1  .WORD
RETURN2  .WORD
                        ; end of assembly
  .END
```

Stand-Alone Applications

The p-System assembler can produce absolute (core image) code files for use outside of the p-System's run-time environment.

The p-System doesn't include a linking loader or an assembly language debugger, as the p-machine architecture isn't conducive to running programs (whether high or low level) that must reside in a dedicated area of memory. You are responsible for loading and executing the object code file; do this by using the p-System, with the understanding that the existing run-time environment may be jeopardized in the process. (For some ideas on how to create a Pascal loader program, refer ahead, in this chapter, to the paragraph, "Executing Absolute Code Files.")

Use Compress utility for a much easier and more versatile way of doing this task. It allows you to relocate and compact code. Refer to Appendix B.

Assembling

Use the .ABSOLUTE and .ORG directives to create an object code file suitable for use as an absolute core image. .ABSOLUTE causes the creation of nonrelocatable object code, and .ORG can initialize the location counter to any starting value. Limit a source file headed by .ABSOLUTE to no more than one assembly routine; sequential absolute routines don't produce continuous object code and can't be successfully linked with one another to produce a core image.

The code file format consists of a one-block code file header followed by the absolute code. It is terminated by one block of linker information; thus, stripping off the first and last block of the code file leaves a core image file. You should use .ABSOLUTE in only one routine; though linker information is generated, it's difficult to link absolute code files to produce a correct core image file.

Executing Absolute Code Files

The following section describes one method of using the p-System to load and execute absolute code files. The program outlined isn't the only solution. You can also use the system intrinsics to read and/or move the code file into the desired memory location; however, this requires a knowledge of where the p-machine emulator, operating system, and your program reside in order to prevent system crashes by accidentally overwriting them. The program outlined below allows you the most freedom in loading core images; the only constraint is that the assembly code itself isn't overwritten while being moved to its final location. You can detect this possibility before proceeding with loading.

NOTE: In most cases, loading object code into arbitary memory locations, while a p-System is resident, adversely affects the system; the absolute assembly language program is then on its own, and rebooting may be necessary to revive the p-System.

The loader program consists of:

1. A host program that calls two external procedures.

2. One or more linkable absolute code files to be loaded. (.RELPROCs aren't allowed.)

3. A small assembly procedure, MOVE_AND_GO, that moves the above object code files from their system load address to their proper locations and then transfers control to them.

4. A small assembly language procedure, LOAD_ADDRESS, that returns the system load addresses of the assembly code to the host program.

The absolute code files are assembled to run at their desired locations, and MOVE_AND_GO contains the desired load addresses of each core image. Both LOAD_ADDRESS and MOVE_AND_GO have external references to the core images; these are used to calculate the system load address and code size of each image file. The whole collection is linked and executed. The host performs the following actions:

1. Print the result of calling LOAD_ADDRESS to determine whether the area of memory in which the p-System loaded the assembly code overlays the known final load address of the core images.

   Issuing a prompt to continue, so that the program can be aborted if a conflict arises.

2. Calls MOVE_AND_GO.

OPERATION OF THE ASSEMBLER

You call the system assembler by pressing 'A' with the operating system Command menu displayed. This command executes the file named SYSTEM.ASSMBLER. (Note the missing 'E' in the file name; this is required to conform to the file system's restrictions on file name lengths.) If this isn't the name of the desired assembler version, be sure to save the existing file 'SYSTEM.ASSMBLER' under a different name before changing the desired assembler's name to 'SYSTEM.ASSMBLER'. Assemblers that aren't in use are usually saved with a file name such as 'ASM8086.CODE'.

Support Files

The p-System Assembler has two associated support files: an opcodes file and an error file. Always store these along with the assembler code file.

In order for the assembler to run correctly, the proper opcodes file must be present on some on-line disk. The opcodes file has a name such as Z80.OPCODES, 9900.OPCODES, and so forth. The opcodes file contains all predefined symbols (instruction and register names) and their corresponding values for the associated assembly language. If the opcodes file isn't on-line, the assembler writes '<opfilename> not on any vol' and aborts the assembly. The 8086 assembler uses an additional opcodes file called 8087.FOPS. This is only necessary when you are programming for the 8087 floating point processor.

The assembler also has an error file that contains a list of processor-specific error messages. The error file has a name such as 8080.ERRORS, 68K.ERRORS, and so forth. The error file need not be present to run the assembler, but it can aid greatly in eliminating syntax errors from a newly written program.

Setting Up Input And Output Files

When you first call the assembler from the Command menu, it attempts to open the work file as its input file; if a work file exists, the first prompt will be the listing prompt described in the next paragraph, "Responses to Listing Prompt," and the generated code file will be named 'SYSTEM.WRK.CODE'. If not, this prompt appears:

```
Assemble what text?
```

Enter the file name of the input file; then press <return>. Pressing only <return> aborts the assembly; otherwise, the next prompt appears:

```
To what codefile?
```

Enter the desired name of the output code file, followed by pressing <return>.

Pressing only <return> here causes the assembler to name the output '*SYSTEM.WRK.CODE', but pressing '$' causes the code file to be created with the same file name prefix as the source file. The assembler then displays its standard listing prompt.

Responses to Listing Prompt

Before assembling begins, the following prompt appears on the console:

```
8086 Assembler [version]
Output file for assembled listing: (<CR> for none)
```

At this point, you may respond with one of the following:

1. The <esc> key followed by <return>; this aborts the assembly and returns you to the Command menu.

2. 'CONSOLE:' or '#1:'; this sends an assembled listing of the source program to the screen during assembly.

3. 'PRINTER:' or '#6:'; which sends an assembled listing to the printer unit.

4. 'REMOUT:' or '#8:'; which sends an assembled listing to the REMOTE unit.

5. A carriage return; which causes the assembler to suppress generation of an assembled listing and ignore all listing directives.

6. All other responses cause the assembler to write the assembled listing to a text file of that name; any existing text file of that name is removed in the process. For instance, the following responses cause a list file named 'LISTING.TEXT' to be created on disk unit 5:

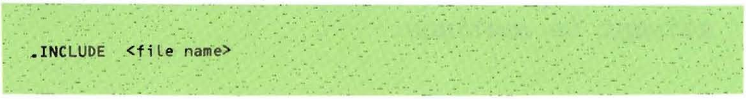
```
#5:Listing.text
#5:Listing
```

In all cases, it's your responsibility to ensure that the specified unit is on-line; the assembler will print an error message and abort if it is requested to open an off-line I/O unit.

Output Modes

If you send an assembled listing to the console, then that listing is displayed on the screen during the assembly process; however, if you send the listing to some other unit or if no listing is generated, the assembler writes a running account of the assembly process to the screen for your benefit. One dot is written to the screen for every line assembled; on every 50th line, the number of lines currently assembled is written on the left side of the screen (delimited by angle brackets).

When the assembler processes an include file directive, the console displays the current source statement:



```
.INCLUDE  <file name>
```

This allows you to keep track of which include file is currently being assembled.

At the end of the assembly, the console displays the total number of lines assembled in the source program and the total number of errors flagged in the source program.

Responses to Error Prompt

When the assembler uncovers an error, it prints the error number and the current source statement. (If applicable to the error; this doesn't apply to undefined labels and system errors.) The assembler then attempts to retrieve and print an error message from the errors file. If the errors file can't be opened—the file doesn't exist or there isn't enough memory—no message appears. This is followed by the menu:

```
<sp>(continue), <esc>(terminate), E(dit
```

Pressing 'E' calls the editor, pressing <space> continues the assembly, and pressing <esc> aborts the assembly. The following restrictions exist when you call the editor or attempt to continue:

1. In most cases, pressing <space> restarts the assembly process with no problems; since assembly language source statements are independent of one another with respect to syntax, it's not difficult for the assembler to continue generating a code file. Thus, a code file will exist at the end of an assembly if you press <space> for every (nonfatal) error prompt that appears; of course, the code produced may not be a correct translation of your source program. The assembler considers certain system errors fatal; these errors abort the assembly regardless of how you respond to the preceding menu.

2. If you press 'E', the system automatically calls the editor. Unless you are using a work file, the editor prompts you for a file name. You should indicate the file currently being assembled. The editor positions the cursor at the location where the error occurred.

Miscellany

At the end of an assembly, an error message is printed for each undefined label. In some cases, you can ignore occurrences of undefined labels if these labels are semantically irrelevant to the desired execution of the code file. The resulting code file will be perfectly valid, but the references to the nonexistent labels won't be completely resolved.

In addition to generating a code file, the assembler makes use of a scratch file, which is always removed from the disk upon normal termination of the assembly. Occasionally though, a system error may occur that prevents the assembler from removing this file; if this happens, a new file named 'LINKER.INFO' may appear. You can easily remove it since it's entirely useless outside of the assembler. This should occur rarely if at all.

ASSEMBLER OUTPUT

The assembler can generate two varieties of output files. It always produces a code file, but you can control whether or not it generates an assembled listing of the source file.

An assembled listing displays each line of the source program, the machine code generated by that line, and the current value of the location counter. The listing may display the expanded form of all macro calls in the source program. Any errors that occur during assembly contain messages printed in the listing file, usually immediately preceding the line of source code that caused the error. A symbol table is printed at the end of the listing; it's the directory for locating all labels declared in the source program.

An assembled listing of a source program printed on hard copy is one of the most effective debugging aids available for assembly language programs; it's equally useful for off-line, 'mental' debugging and for use with system debuggers.

A description of the code file format is beyond the scope of this document. See the Internal Architecture Reference Manual.

Source Listing

When you respond to the assembler's listing prompt with a list file name, a paginated assembled listing is produced. The default listing is 132-characters wide and 55 lines per page. Each line of a source program is included in the assembled listing, except for source lines that contain list directives. Source statements that contain the equate directive .EQU have the resulting value of the associated expression listed to the left of the source line.

Macro calls are always listed, including the list of macro parameters and the comment field, if any. The macro is expanded by listing the body (with all formal parameters replaced by their passed values) if the macro list option was enabled when the macro was defined. Macro expansion text is marked in the assembled listing by the character '#' just to the left of the source listing. Comment fields in the definition of the macro body aren't listed in macro expansions.

Source lines with conditional assembly directives are listed; however, source statements in an unassembled part of a conditional section aren't listed unless the .CONDLIST directive has been used.

Error Messages

Error messages in assembled listings have the same format as the error messages sent to the console, except that the prompt isn't included. (Refer back to the section, "Operation of the Assembler.")

Code Listing

The code field lies to the left of the source program listing. It always contains the current value of the location counter, along with either code generated by the matching source statement or the value of an expression occurring in a statement that includes the equate directive .EQU. All are printed in the default list radix of the assembler version being used in either hexadecimal or octal. (Refer ahead in this chapter to the section, "Example Assembled Listing.") Spaces delimit separately emitted bytes and words of code on the same line.

Forward References

When the assembler is forced to emit a byte or word quantity that is the result of evaluating an expression that includes an undefined label, it lists a '*' for each digit of the quantity printed (for example, an unresolved hexadecimal byte is listed as '**', while an unresolved octal word appears as '******'). If you use the .PATCHLIST directive, the assembler lists patch messages every time it encounters a label declaration that enables it to resolve all occurrences of a forward reference to that label. The messages (one for every backpatch performed) appear before the source statement that contains the label in question; they look like this:

```
<location in codefile patched>* <patch value>
```

With this feature, the listing describes the contents of each byte or word of emitted code. If you want the assembled listing to be especially clean and neat, use the .NOPATCHLIST directive to suppress the patch messages.

External References

When the assembler emits a word quantity that results from evaluating an expression that contains an externally referenced label, the value of that label (which can't be determined until link time) is taken as zero. Therefore, the emitted value reflects only the result of any assembly time constants that were present in the expression.

Multiple Code Lines

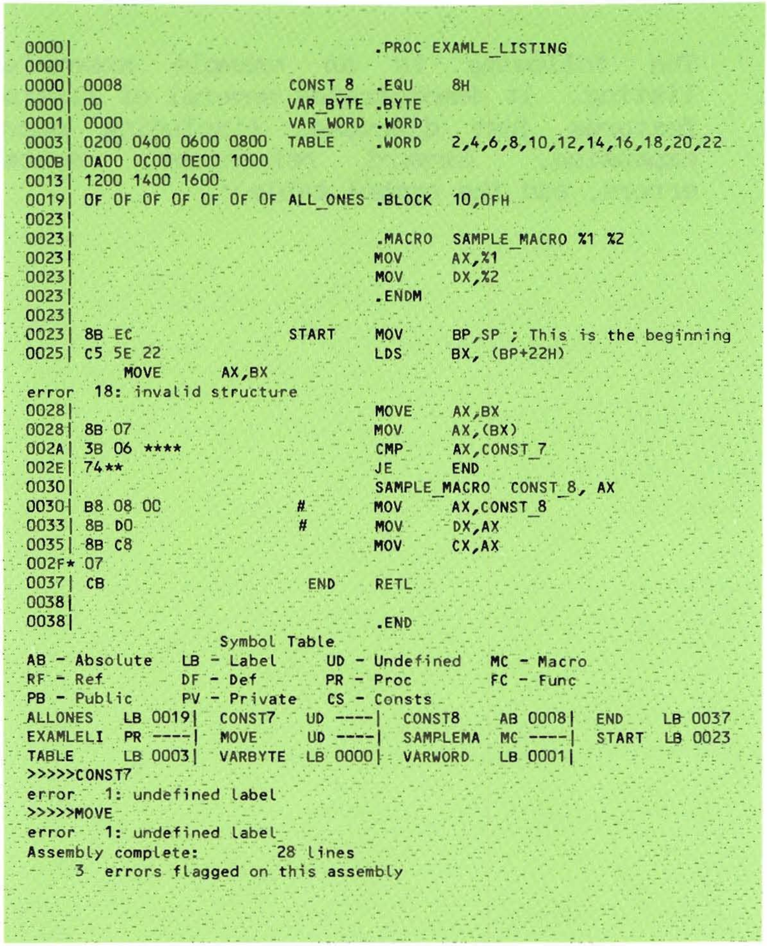
Sometimes, one source statement can generate more code than can fit in the code field. In most cases, the code is listed on successive lines of the code field, with corresponding blank source listing fields. Three exceptions are the .ORG, .ALIGN, and .BLOCK directives; the code field for these arguments is limited to as many bytes as will fit in the code field of one line. This is because most uses of these directives generate large numbers of uninteresting byte values.

Symbol Table

The symbol table is an alphabetically sorted table of entries for all symbols declared in the source program. Fach entry consists of three fields; the symbol identifier, the symbol type, and the value assigned to that symbol. The symbol identifiers are defined in a dictionary printed at the top of the symbol table. Symbols equated to constants have their constant values in the third field, while program labels are matched with their location counter offsets; all other symbols have dashes in their value field, as they possess no values relevant to the listing.

Example Assembled Listing

The following is an example assembled listing. It demonstrates several of listing features just discussed (including macro expansion, forward references, syntax errors, and the symbol table):



```
0000|                                .PROC EXAMLE_LISTING
0000|
0000| 0008                 CONST_8  .EQU     8H
0000| 00                   VAR_BYTE .BYTE
0001| 0000                 VAR_WORD .WORD
0003| 0200 0400 0600 0800  TABLE    .WORD    2,4,6,8,10,12,14,16,18,20,22
000B| 0A00 0C00 0E00 1000
0013| 1200 1400 1600
0019| 0F 0F 0F 0F 0F 0F 0F ALL_ONES .BLOCK   10,0FH
0023|
0023|                               .MACRO   SAMPLE_MACRO %1 %2
0023|                               MOV      AX,%1
0023|                               MOV      DX,%2
0023|                               .ENDM
0023|
0023| 8B EC                START    MOV      BP,SP ; This is the beginning
0025| C5 5E 22                      LDS      BX, (BP+22H)
          MOVE      AX,BX
error  18: invalid structure
0028|                               MOVE     AX,BX
0028| 8B 07                         MOV      AX,(BX)
002A| 3B 06 ****                    CMP      AX,CONST_7
002E| 74**                          JE       END
0030|                               SAMPLE_MACRO  CONST_8, AX
0030| B8 08 00              #       MOV      AX,CONST_8
0033| 8B D0                 #       MOV      DX,AX
0035| 8B C8                         MOV      CX,AX
002F* 07
0037| CB                     END    RETL
0038|
0038|                               .END
                    Symbol Table
AB - Absolute   LB - Label     UD - Undefined   MC - Macro
RF - Ref        DF - Def       PR - Proc        FC - Func
PB - Public     PV - Private   CS - Consts
ALLONES   LB 0019|  CONST7   UD ----|  CONST8    AB 0008|  END    LB 0037
EXAMLELI  PR ----|  MOVE     UD ----|  SAMPLEMA  MC ----|  START  LB 0023
TABLE     LB 0003|  VARBYTE  LB 0000|  VARWORD   LB 0001|
>>>>>CONST7
error   1: undefined label
>>>>>MOVE
error   1: undefined label
Assembly complete:        28 lines
     3  errors flagged on this assembly
```

CHAPTER 2

PROCESSOR-SPECIFIC INFORMATION

INTRODUCTION

This chapter is intended to be used in conjunction with processor manuals distributed by the manufacturers of the various processors. These manuals provide syntax conventions for the instruction sets and address modes used by the corresponding assembler versions. The company chosen as a base for syntax conventions is listed for each version, along with a list of deviations from that company's syntax conventions.

LSI-11/PDP-11 ASSEMBLER

Syntax Conventions

The 11 assembler adheres to DEC standard syntax for opcode fields, register names, and address modes. The location counter symbol is an asterisk '*'.

Sharing PME Resources

The return address to the system is passed on the stack. Registers 0 and 1 are available to the assembly routine; other registers must be saved on entry and restored on exit.

Memory Organization

The 11 processor is byte-addressed and word-oriented; machine instructions and data words must be aligned to start on an even byte boundary. The byte sex is least-significant-byte-first.

Default Constant and List Radices

The default constant radix and default list radix are octal.

Z80 ASSEMBLER

Syntax Conventions

The Z80 assembler adheres to Zilog standard syntax for opcode fields, register names, and address modes. The following conventions may deviate from this standard:

- The syntax for exchanging the register pair AF and the alternate register pair 'AF' is the following:

```
      EX   AF
```

  The location counter symbol is a dollar sign '$'.

Sharing PME Resources

The return address to the system is passed on the stack. All registers are available for use in the assembly routine.

Memory Organization

The Z80 processor is byte-addressed and byte-oriented. The byte sex is least-significant-byte-first.

Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

6502 ASSEMBLER

Syntax Conventions

The 6502 assembler adheres to Rockwell standard syntax for opcode fields and register names. The following conventions may deviate from this standard:

- Immediate operands are specified by using a preceding pound sign '#' character:

```
LABEL   .EQU   5
 LDA    #LABEL      ; immediate
```

- Zero-page addressing is achieved only by using absolute operands (that is, assembly time constants) with values between 0 and 255:

```
LABEL   .EQU   5
 LDA    LABEL       ; zero-page
```

- Indirect addressing has the following form:

```
LDA    @LABEL,X     ; indexed-indirect (preindexing)
LDA    @LABEL,Y     ; indirect-indexed (postindexing)
JMP    @LABEL       ; indirect jump
```

The location counter symbol is an asterisk '*'.

Sharing PME Resources

The return address to the system is passed on the stack. All registers are available for use in the assembly routine.

Memory Organization

The 6502 processor is byte-addressed and byte-oriented. The byte sex is least-significant-byte-first.

Default Constant and List Radices

The default constant radix and default list radix are hexadecimal.

6800 ASSEMBLER

Syntax Conventions

The 6800 assembler adheres to Motorola standard syntax for opcode fields and register names. The following conventions may deviate from this standard:

- All instructions which can specify the A and B registers have the register name separated from the opcode field:

```
LDA   A,LABEL
LDA   A,0,X       (instead of LDA  A,X)
LDX   0,X         (instead of LDA  X)
STA   A,14,X
PUL   A
ASL   B
```

- Immediate operands are specified by using a preceding pound sign '#' character:

```
LABEL   .EQU   5
 LDA    A,#LABEL    ; immediate
```

- Zero-page addressing is achieved only by using absolute operands (that is, assembly time constants) with values between 0 and 255:

```
LABEL   .EQU   5
 LDA    B,LABEL     ; zero-page
```

- Numbers in hex must always contain four digits (yes, even for bytes):

  ```
      .BYTE   0002H,00A9H   specifies the quantity 02A9 base 16
  ```

  The location counter symbol is an asterisk '*'.

Sharing PME Resources

The return address to the system is passed on the stack. All registers are available for use in the assembly routine.

Memory Organization

The 6800 processor is byte-addressed and byte-oriented. The byte sex is most-significant-byte-first.

Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

8080 ASSEMBLER

Syntax Conventions

The 8080 assembler adheres to Intel standard syntax for opcode fields, register names, and address modes. The location counter symbol is a dollar sign '$'.

Sharing PME Resources

The return address to the system is passed on the stack. All registers are available for use in the assembly routine.

Memory Organization

The 8080 processor is byte-addressed and byte-oriented. The byte sex is least-significant-byte-first.

Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

9900 ASSEMBLER

Syntax Conventions

The 9900 assembler adheres to TI standard syntax for opcode fields, register names, and address modes. The following conventions may deviate from this standard:

- In operand fields, the lack of an address mode character (for example, a '@' or '*' preceding the operand) defaults to '@'. The location counter symbol is a dollar sign '$'.

Sharing PME Resources

The return address to the system is passed in register 11. Registers 0 thru 5 are available to the assembly routine; other registers must be saved on entry and restored on exit.

Memory Organization

The 9900 processor is byte-addressed and word-oriented; machine instructions and data words must be aligned to start on an even byte boundary. The byte sex is most-significant-byte-first.

Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

6809 ASSEMBLER

Syntax Conventions

The 6809 Assembler adheres to Motorola standard syntax for opcode fields and register names. The following conventions may deviate from this standard:

- Immediate operands are specified by using a preceding '#':

```
ANDCC       #01
```

- Indirect addressing is specified by a single leading at sign ('@') instead of square brackets ('[ ]'):

```
LDX         @THERE,PCR
```

- Zero-page addressing is achieved only by using operands that are absolute (for example, not labels) and less than 256:

```
ZEROPAGE  .EQU   15
 LDB     ZEROPAGE
```

Sharing PME Resources

The return address to the system is passed on the stack. Registers Y and U must be saved and restored if they are to be used. All other registers are available for use.

Memory Organization

The 6809 processor is byte-addressed and byte-oriented. The byte sex is most-significant-byte first.

Default Constant and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

Z8 ASSEMBLER

Syntax Conventions

Symbols

The Z8 Adaptable Assembler adheres to Zilog standard syntax (refer to the Z8 PLZ/ASM Assembly Language Programming Manual) for opcode fields, register names, and addressing modes.

Numeric Constants

The Z8 Assembler follows the constant conventions of other adaptable assemblers, except that octal constants are indicated by a radix switch character of 'O' rather than 'Q', and binary constants are indicated by a radix switch character of 'B' rather than 'T'.

```
011101B    0B    14670    111100
```

Predefined Constants

There are no predefined constants in the Z8 Assembler. Specifically, the constants '%L', '%T', '%R', '%P', '%%', and '%Q' in Zilog syntax are NOT allowed.

Sharing PME Resources

No PME is currently available for the Z8.

Memory Organization

The Z8 processor is byte-addressed and byte-oriented. The byte sex is least-significant-byte-first.

Default and List Radices

The default constant radix is decimal and the default list radix is hexadecimal.

8086/8088/8087 ASSEMBLER

Syntax Conventions

The p-System 8086/88/87 Assembler differs in some respects from the standard Intel assembler. This section lists these differences.

Assembler Directives. None of the Intel assembler directives are implemented. Instead, the assembler directives described in Chapter 1 of this manual are available.

Parenthesis. Enclose index or base register references in a memory operand in parentheses, not square brackets; for example, FIRST(BX) rather than FIRST[BX]. Group expressions with angle brackets rather than parentheses.

Immediate Byte. Code ADD immediate byte to memory operand as:

```
ADDBIM  memop,immedbyte
```

to distinguish it from the ADD memop, immedword, which is the default. Similarly, MOVBIM, ADCBIM, SUBBIM, SBBBIM, CMPBIM, ANDBIM, ORBIM, XORBIM, and TESTBIM are added to the vocabulary.

Memory Byte. Code INC memory byte as:

```
INCMB    memop
```

to distinguish it from INC memory word, which is the default. Similarly, DECMB, MULMB, IMULMB, DIVMB, IDIVMB, NOTMB, NEGMB, ROLMB, RORMB, RCLMB, RCRMB, SALMB, SHLMB, SHRMB, SARMB are added to the vocabulary to specify memory byte operands.

Direct Addressing Mode. Code MOV with direct addressing as:

```
MOVM     AX,02DEFH
MOVM     02DEFH,AX
```

to distinguish it from MOV immediate value which is the default. Similarly, ADCM, ADDM, ANDM, CMPM, ORM, SBBM, TESTM, and XORM are added to the vocabulary for use with direct addressing.

MUL and DIV Byte. In MUL, IMUL, DIV, IDIV the single memory operand form,

```
MUL     memop
```

implies a word operation. To specify a byte operation, you may use either MULMB memop, or the form

```
MUL     AL,memop
```

The same holds true for IMUL, DIV, IDIV. (Note that DIV AL,memop is rather misleading, as the actual operation would be AX/memory-byte.)

MOV Substitute for LEA. For LEA reg,label or LEA reg,label+const the assembler substitutes MOV reg,immedval where immedval = label or label+const. This saves four clock times (4 versus 8).

IN and OUT. The normal form of IN and OUT is IN ac,port or IN ac,DX and OUT port,ac or OUT DX,ac where ac=AL denotes an 8-bit data path and ac=AX denotes a 16-bit path. Since the accumulator is the only possible register source/destination (DX specifies port=address in DX), single operand forms are also provided: INB and OUTB for byte data, and INW and OUTW for 16-bit data. The syntax is INB port or INB DX.

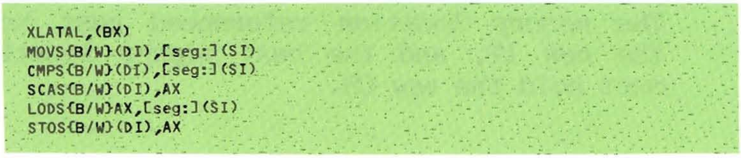
In the two-operand forms of IN and OUT, the order of the operands isn't important; thus OUT ac,DX or OUT ac,port will be acceptable.

String Operations. The mnemonics for the string operations are suffixed with B or W to denote byte or word operations; thus, MOVSB and MOVSW, CMPSB and CMPSW, SCASB and SCASW, LODSB and LODSW, and STOSB and STOSW are in the vocabulary, but MOVS—STOS aren't.

Segment Override. XLAT and the string instructions (9) have implied memory operands and nothing is required to be coded in the operand field. However, to permit you to specify a segment override prefix in the case of XLAT, MOVSB/MOVSW, CMPSB/CMPSW, and LODSB/LODSW, the assembler permits operand expressions for these instructions.

NOTE: That only the default segment for SI, namely DS, can be overridden. The segment for DI is ES and can't be overridden. A segment override prefix of DS applied to SI doesn't generate a segment override prefix.

If you were to write these operations with operands, they would have this syntax:



```
XLATAL,(BX)
MOVS{B/W}(DI),[seg:](SI)
CMPS{B/W}(DI),[seg:](SI)
SCAS{B/W}(DI),AX
LODS{B/W}AX,[seg:](SI)
STOS{B/W}(DI),AX
```

You may prefix the string instructions with a REP (repeat) instruction of some type. The assembler flags an error if you specify both REP and a segment override.

In addition to the forms DS:memop, and so on, you may write a separate mnemonic SEG followed by a segment register name in a statement preceding the instruction mnemonic. For example:

```
MOV  AX,ES:AVALUE
```

is equivalent to:

```
SEG  ES  MOV  AX,AVALUE
```

Long Jumps, Calls, and Returns. Implement intersegment CALL, RET, and JMP as follows:

1. The mnemonics CALLL, RETL, and JMPL specifically designate intersegment operations.

2. An indirect address (for example, (reg) or (label)) is assembled in standard fashion with a "mod op r/m" effective address byte possibly followed by displacement bytes. The memory location referenced must hold the new IP, and the next higher location must hold the new CS.

3. The direct address form must have two absolute operands:

```
CALLL    expr1,expr2
```

   where expr1 is the new IP and expr2 becomes the new CS. Constants or external symbols (for example, .REF definitions) qualify as absolute operands.

8087 Mnemonics. Mnemonics for the 8087 floating point operations are standard except for some of the memory reference operations, where a letter suffix is appended to denote the operand size:

D	short real or short integer (double word)
Q	long real or long integer (quad word)
W	integer word
T	temporary real (ten byte)

The 'D' and 'Q' suffixes apply to the following real ops:

FADD, FCOM, FCOMP, FDIV, FDIVR, FMUL,
FST, FSUB, FSUBR, FLD, FSTP

For example, FADDD, FADDQ, and so.

The 'T' suffix applies only to FLD and FSTP.

The 'W' and 'D' suffixes apply to the following integer ops:

FIADD, FICOM, FICOMP, FIDIV, FIDIVR, FIMUL, FIST, FISUB, FISUBR, FILD, FISTP

The 'Q' suffix for long integers applies only to FILD and FISTP.

Sharing PME Resources

Calling and Returning

The p-machine emulator (PME) calls an assembly routine using the call long (CALLL) operator. Thus, the top of the stack contains a two-word return address upon entering into the routine. In order to return from an assembly routine, use the return long (RETL) operator. (Alternatively, the return address can be popped and a jump long (JMPL) operation used.)

Accessing Parameters

The 8086/88 Processor contains instructions that facilitate accessing parameters passed to an assembly routine. By moving the value of SP (which points to the p-machine stack) into BP, you can access the parameters by adding an offset of 4 bytes (to account for the two-word return address). The first parameter, located four bytes above the top of the stack, is actually the last declared parameter in the host routine (the parameters are pushed in the order that they are declared).

If a .FUNC assembly routine is to return a function value, you should place it just above the last parameter (which is just before the first declared parameter) using the same accessing scheme. The size of the returned function value is either one, two, or four words as described in a previous paragraph called, "Linking with a Pascal Program."

You may give the RETL operator an operand that indicates how many bytes to cut the stack back after popping its two-word return address. Use the size of the data space occupied by the parameters. Thus, parameters may be accessed, and a clean return made, without ever using a specific POP or PUSH instruction.

The following is an example of this scheme of accessing parameters and returning:

```
MOV   BP,SP
MOV   AX,(BP+4)     ;Last Param
MOV   BX,(BP+6)     ;Middle Param
MOV   CX,(BP+8)     ;First Param
             .
             .
             .
MOV   (BP+10),RSLT  ;Function return val
                    ; (if .FUNC)
RETL  6             ;Remove 3 params
```

Register Usage

All of the 8086/88 registers are available for use by your assembly routines (the PME saves and restores the register values that it needs).

However, you must preserve SS and SP. (You may create and use a private stack if a minimum of 40 words are left available for stack expansion during interrupts. This is a very dangerous procedure, however, and is not recommended.)

NOTE: You must maintain the integrity of the p-machine stack. If you don't, the results can't be predicted.

Upon entering into the assembly routine, SS points to the base of the p-machine stack and data area. Also, DS, ES, and CS are all equal to the base of the p-System code segment.

Parameters that are passed as Pascal VAR variables are p-System pointers to actual data. These pointers are relative to SS. For example:

```
MOVBX, (BP+4)    ; pick up parameter (pointer)
MOVAX, SS;(BX)   ; pick up VAR parameter value
```

.PRIVATE and .PUBLIC variables are also SS relative. For example:

```
.PRIVATE        COUNTER
MOV             AX,SS:COUNTER
```

.BYTE quantities, .WORD quantities, and .REF'ed labels are relative to CS, DS, or ES.

Memory Organization

The 8086 processor is byte-addressed and byte-oriented. The byte sex is least-significant-byte-first.

Default Constant and List Radices

The default constant radix is decimal. The default list radix is hexadecimal.

68000 ASSEMBLER

Syntax Conventions

The 6800 Assembler follows Motorola standard syntax for opcode fields, register names and address modes. The following list points out some restrictions.

- Only the absolute short address mode is available. The absolute long address can't be generated by the assembler.
- Labels may not be accessed with the absolute address mode.
- References to labels with a .PROC or .FUNC generate the PC-relative address mode.
- An external label may only be accessed as a displacement from an address register.
- Immediates above FFFFH can't be generated.
- Opcodes which have an optional suffix of A, I, M, Q or X must contain that suffix explicitly.
- Length qualifiers (.B, .W or .L) must be specified explicitly in those instructions which have a choice of length. All other instructions must not contain a length qualifier.

The following instuctions must contain a length qualifier:

ADD, ADDA, ADDI, ADDQ, ADDX, AND, ANDI, ASL (register), ASR (register), CLR, CMP, CMPA, CMPI, CMPM, EOR, EORI, EXT, LSL (register), LSR (register), MOVE (except special forms), MOVEA, MOVEM, MOVEP, NEG, NEGX, NOT, OR, ORI, ROL (register), ROR (register), ROXL (register), ROXR (register), SUB, SUBA, SUBI, SUBQ, SUBX, TST

The following instructions must not contain a length qualifier:

ABCD, ASL (memory), ASR (memory), BCHG, BCLR, BSET, BTST, CHK, DBcc, DIVS, DIVU, EXG, JMP, JSR, LEA, LINK, LSL (memory), LSR (memory), MOVE to CCR, MOVE to SR, MOVE from SR, MOVE USP, MOVEQ, MULS, MULU, NBCD, NOP, PEA, RESET, ROL (memory), ROR (memory), ROXL (memory), ROXR (memory), RTE, RSR, RTS, SBCD, Scc, STOP, SWAP, TAS, TRAP, TRAPV, UNLK

The following instructions may contain an optional length qualifier of .S (generate short forward branch):

Bcc, BRA, BSR

Sharing PME Resources

An assembly language procedure is called via a JSR instruction, so it should expect a double word return address on the stack. It is usual to return via an RTS instruction.

Registers A0-A2 and D0-D7 are available for use. Register A3-A7 must be restored to the values at call-time if they are used.

Since pointers within the p-machine are byte offsets from a base register (A6), .PUBLIC references to Pascal variables will generate an offset, not the actual address, of the variable. In order to access an external variable, it is necessary to use this offset as a displacement from A6. For example:

```
    ADDQ.W     #1,ABC(A6)
```

will increment the Pascal variable ABC.

A variable parameter is a p-machine pointer to the parameter, so it is also accessed as above. For example, a variable parameter may be accessed as follows:

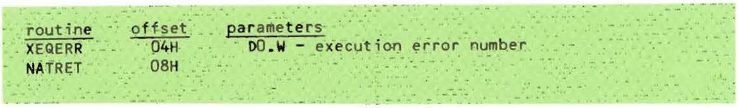
```
MOVEQ     #0,D7          ; clear the upper half of D7
MOVE.W    4(SP),D7       ; load the pointer (parameter)
ADDQ.W    #1,0(A6,D7.L)  ; increment the variable
```

References to variables in other assembly language procedures (via a .REF) may be accessed as above using (A2), provided the segment the procedures are in is located in the data area (for example, it isn't a RELPROC).

Here is a list of the register values available to the assembly language procedure on entry:

```
A2 - base of current segment
A3 - base of PME
A4 - p-machine program counter
A6 - pointer to data area
A7 - stack pointer
```

The .INTERP directive (used to access items in the PME) is ignored. Instead, accesses should be made relative to A3 (the base of the PME). The following entry-points are available to the assembly language programmer:



routine	offset	parameters
XEQERR	04H	D0.W - execution error number
NATRET	08H	

XEQERR may be used to cause an execution error to be recognized from assembly language. XEQERR should be jumped to, not called. Before jumping to XEQERR, the stack should be clear of all parameters (including the return word), and all registers should be restored. This routine is normally used for system work.

NATRET is the entry-point used by automatically generated native-code to return to the p-System. It shouldn't be used for any other purpose.

Memory Organization

The 68000 processor is byte-addressed and word-oriented. The byte sex is most-significant-byte first.

Default Constant and List Radices

The default constant radix is decimal, and the default list radix is hexadecimal.

APPENDICES

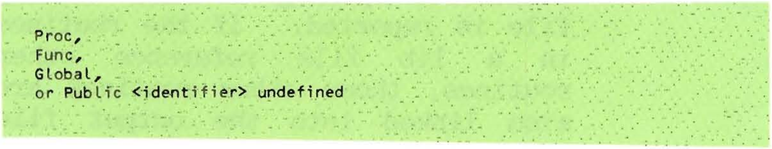
APPENDIX A
THE LINKER

The linker is an item on the Command menu which allows assembled code to be linked into a host program. The linker may also be used to link together separately assembled pieces of a single assembly program.

The linker is a program of the sort called a "link editor." It stitches code together by installing the internal linkages that allow various pieces to functon as a unified whole.

When a program that must be linked is R(un, the linker is automatically called and searches *SYSTEM.LIBRARY for the necessary external routines. If you use X(ecute, instead of R(un, or the assembled routines aren't in SYSTEM.LIBRARY, you are responsible for manually linking the code before executing it.

When the linker is called automatically and can't find the needed code in *SYSTEM.LIBRARY, it responds with the following error message.



```
Proc,
Func,
Global,
or Public <identifier> undefined
```

In order to manually use the linker, select L(ink from the Command menu.

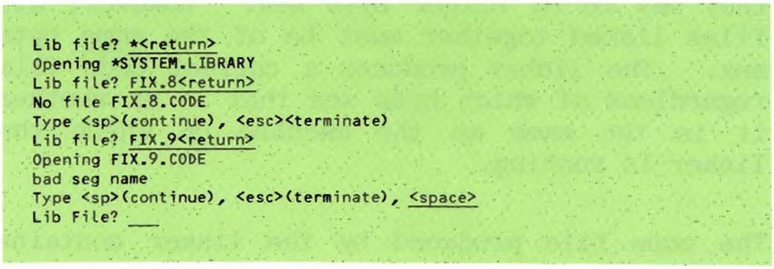
Using the Linker

The linker displays prompts asking for several file names. It reads and links code together, and displays the names of the routines it is linking. The following paragraphs list those prompts and explain the use or responses.

Host file? The host file should contain the code for the high-level program which references external routines. Alternatively, the host file may contain an assembled routine which references other assembled routines. The ".CODE" suffix is automatically appended to the file name that you specify (unless you terminate that name with a period). If you respond with <return>, the linker attempts to open the code work file as the host file.

Lib file? Any number of library files may be specified. The prompt will keep reappearing until you press the <return>. Responding '*<return>' opens *SYSTEM.LIBRARY. The successful opening of each library file is reported. If the routines in a lib file reference other routines, those other routines are also linked into the output file (assuming that they are found in one of the lib files).

Example (underlined portions are your input):



```
Lib file? *<return>
Opening *SYSTEM.LIBRARY
Lib file? FIX.8<return>
No file FIX.8.CODE
Type <sp>(continue), <esc><terminate)
Lib file? FIX.9<return>
Opening FIX.9.CODE
bad seg name
Type <sp>(continue), <esc>(terminate), <space>
Lib File? ___
```

When the names of all library files have been entered, the linker reads all the necessary routines from the designated code files. It then asks for a destination for the linked code output:

Output file? Respond with a code file name (often the same as the host file). The .CODE suffix must be included. If you press <return>, (*SYSTEM.WRK.CODE) becomes the output file.

After this last prompt, the linker commences actual linking. During linking, the linker displays the names of all routines being linked. A missing or undefined routine causes the linker to abort with the '<identifier> undefined' message described above.

NOTE: Since the files may be assembled files, they may be of either byte sex. However, all files linked together must be of the same byte sex. The linker produces a correct code file regardless of which byte sex that is or whether it is the same as the machine on which the linker is running.

The code file produced by the linker contains routines in the order in which they were given in the library files. This is important to note if the program is an assembly language file. The code file contains first routines from the host file and then library file routines, all in their original order.

APPENDIX B
THE COMPRESS UTILITY

The Compress utility program takes an input code file consisting of one or more linked assembly procedures. It produces an object file suitable for execution outside the p-System run-time environment.

Compress can produce either relocatable or absolute object files. Absolute code files are relocated to the base address specified by you and contain pure machine code. Relocatable code files include a simplified form of relocation information (a description of its format is in this appendix). Both kinds of output files are stripped of all file information normally used by the system and must be loaded into memory by your program in order to execute properly.

Preparing Code Files

The assembly routines must be created with the assembler, and linked with the linker. Code files containing anything other than one segment of linked assembly code will cause Compress to abort. Routines to be compressed shouldn't contain any of the following assembler directives.

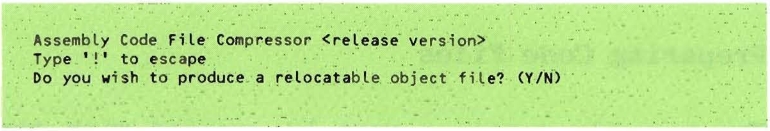
```
.ORG          .ABSOLUTE
.PUBLIC       .PRIVATE
.CONST        .INTERP
```

The .ORG and .ABSOLUTE directives produce absolute code files directly from the assembler. Code files that contain the .ABSOLUTE directive can be compressed, but the resulting code will be incorrect.

The .PUBLIC, .PRIVATE, .CONST and .INTERP directives are used to communicate between assembly procedures and a host compilation unit (whether Pascal or some other language). These have no use outside of the system's run-time environment. Their inclusion in an assembly program generates relocation information in formats that will cause Compress to abort.

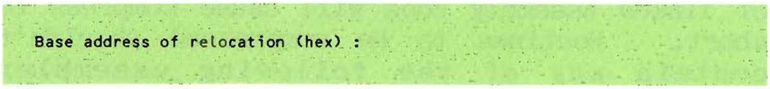
Running Compress

In order to run Compress, you should X(ecute COMPRESS.CODE. This utility displays the following prompt:



```
Assembly Code File Compressor <release version>
Type '!' to escape
Do you wish to produce a relocatable object file? (Y/N)
```

If you press 'N', the following prompt appears:



```
Base address of relocation (hex) :
```

This is the starting address of the absolute code file to be produced. It should be entered as a sequence of 1 to 4 hexadecimal digits followed by <return>. The prompt will reappear if an invalid number is entered.

The following prompt always appears:

```
File to compress :
```

Enter the name of the file to be compressed. It isn't necessary to enter the '.CODE' suffix. If the file can't be found, the prompt reappears.

```
Output file (<ret> for same) :
```

Enter the name of the output file, which can be any legal file name (Compress doesn't append a .CODE suffix). Pressing <return> causes the output file to have the same name as the input file, thus eliminating the original input file. If the file can't be opened, Compress will print an error message and abort.

In all the previous prompts, pressing the character '!' causes Compress to abort.

After receiving this information from you, Compress reads the entire source file, compresses the procedures, and writes out the entire destination file. Large code files may cause Compress to abort, if the system doesn't have sufficient memory space.

While running, Compress displays for each procedure the starting and ending addresses (in hexadecimal) and the length in bytes. After finishing, the total number of bytes in the output file is displayed. If an absolute code file is produced, the system displays the highest memory address to be occupied by the loaded code file.

Compress produces a file of pure code, which must be loaded and executed directly by your software.

Action and Output Specification

Compress removes the following information from input files:

- The segment dictionary (block 0 of code file).
- Relocation list and procedure dictionary pointers.
- Symbolic segment name and code sex word.
- Embedded procedure DATASIZE and EXITIC words.
- Procedure dictionary and number of procs word.
- Standard relocation list.

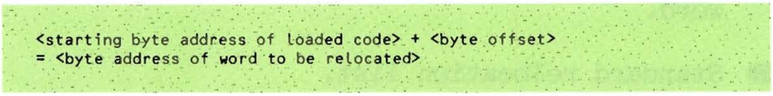
Procedure code in the output file is contiguous, except for padding bytes, which are emitted (when necessary) to preserve the word alignment of all procedures. Code files always contain an integral number of blocks of data and space between the end of the executable code. The end of the code file is zero-filled.

Relocatable object files must be formatted in the following way. The relocatable code is immediately followed by relocation information. The last word in the last block of the code file contains the code-relative word offset of the relocation list header. The following lines are an example.

```
<starting byte address of loaded code> + <word offset * 2>
= <byte address of relocation list header word>
```

The list header word contains the decimal value 256. The next-lower-addressed word contains the number of entries in the relocation list. This word is followed (from higher addresses to lower addresses) by the list of relocation entries.

Beneath the last relocation entry is a zero-filled word, which marks the end of the relocation information. Each relocation entry is a word quantity containing a code-relative byte offset into the loaded code. The following lines are an example:



```
<starting byte address of loaded code> + <byte offset>
= <byte address of word to be relocated>
```

Each byte address pointed to by a relocation entry is a word quantity that is relocated by adding the byte address of the front of the loaded code.

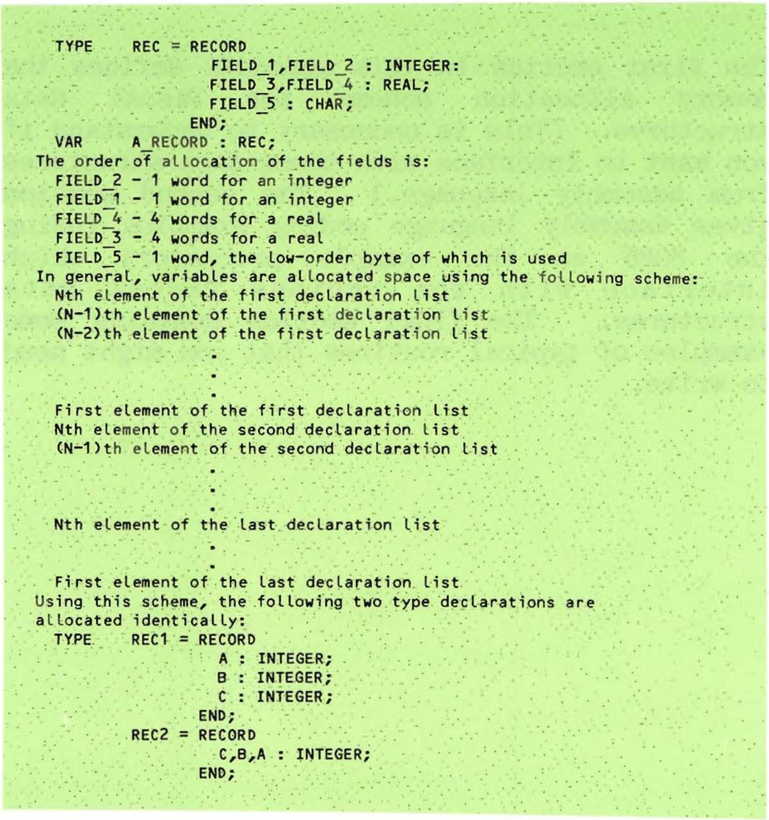
NOTE: If you relocate a file towards the high end of the 16-bit address space, you must ensure that the relocated file won't wrap around into low memory (that is, <relocation base address> + <code file size> must be less than or equal to FFFF(hexadecimal)). Compress performs no internal checking for this case.

APPENDIX C
CODING EXAMPLES

The first section in this appendix defines the memory allocation scheme for Pascal data structures. (This is necessary to understand if you want to interface with these data structures from assembly language.) The second section gives assembly language coding examples (using the 8086 as the example processor) which interface with the various Pascal data structures. The final section contains some examples of typical routines that you might need to write.

PASCAL DATA STRUCTURES

Given the following Pascal declaration:

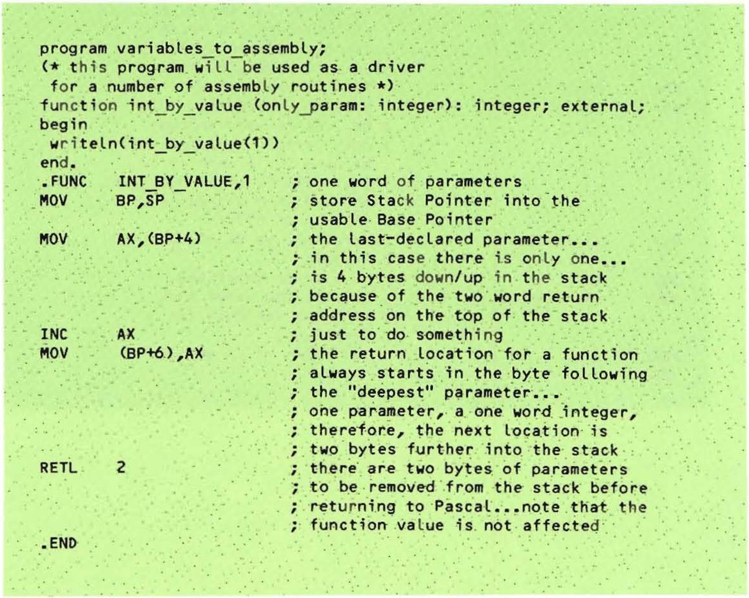


```
  TYPE    REC = RECORD
                  FIELD_1,FIELD_2 : INTEGER:
                  FIELD_3,FIELD_4 : REAL;
                  FIELD_5 : CHAR;
                END;
  VAR     A_RECORD : REC;
The order of allocation of the fields is:
  FIELD_2 - 1 word for an integer
  FIELD_1 - 1 word for an integer
  FIELD_4 - 4 words for a real
  FIELD_3 - 4 words for a real
  FIELD_5 - 1 word, the low-order byte of which is used
In general, variables are allocated space using the following scheme:
  Nth element of the first declaration list
  (N-1)th element of the first declaration list
  (N-2)th element of the first declaration list
                .
                .
                .
  First element of the first declaration list
  Nth element of the second declaration list
  (N-1)th element of the second declaration list
                .
                .
                .
  Nth element of the last declaration list
                .
                .
  First element of the last declaration list
Using this scheme, the following two type declarations are
allocated identically:
  TYPE    REC1 = RECORD
                   A : INTEGER;
                   B : INTEGER;
                   C : INTEGER;
                 END;
          REC2 = RECORD
                   C,B,A : INTEGER;
                 END;
```

INTERFACING WITH PASCAL

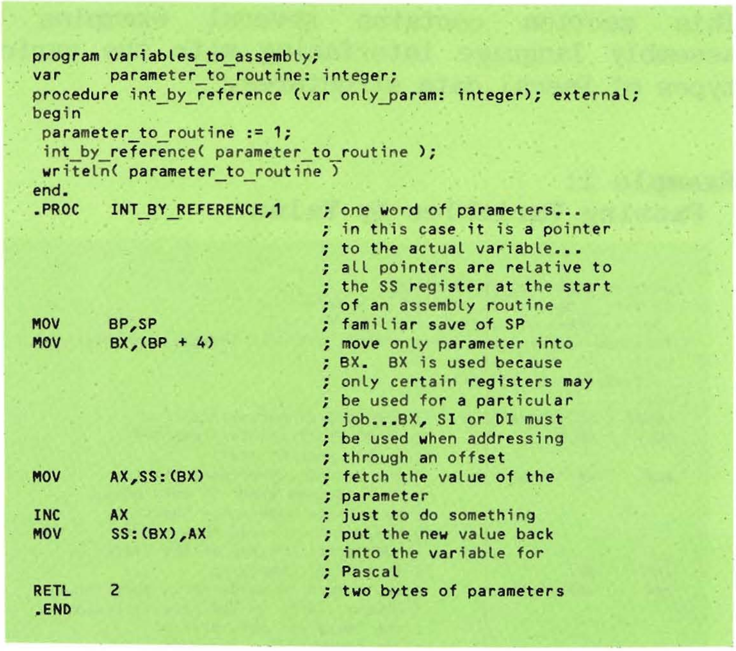
This section contains several examples of assembly language interfacing with the various types of Pascal data structures.

Example 1:
Passing Variables by Value



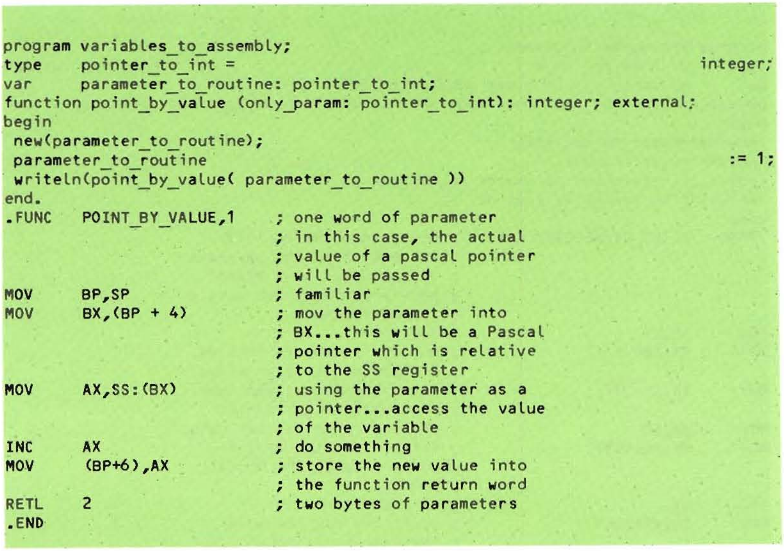
```
program variables_to_assembly;
(* this program will be used as a driver
 for a number of assembly routines *)
function int_by_value (only_param: integer): integer; external;
begin
 writeln(int_by_value(1))
end.
.FUNC   INT_BY_VALUE,1    ; one word of parameters
MOV     BP,SP             ; store Stack Pointer into the
                          ; usable Base Pointer
MOV     AX,(BP+4)         ; the last-declared parameter...
                          ; in this case there is only one...
                          ; is 4 bytes down/up in the stack
                          ; because of the two word return
                          ; address on the top of the stack
INC     AX                ; just to do something
MOV     (BP+6),AX         ; the return location for a function
                          ; always starts in the byte following
                          ; the "deepest" parameter...
                          ; one parameter, a one word integer,
                          ; therefore, the next location is
                          ; two bytes further into the stack
RETL    2                 ; there are two bytes of parameters
                          ; to be removed from the stack before
                          ; returning to Pascal...note that the
                          ; function value is not affected
.END
```

Example 2:
Passing Variables by Reference



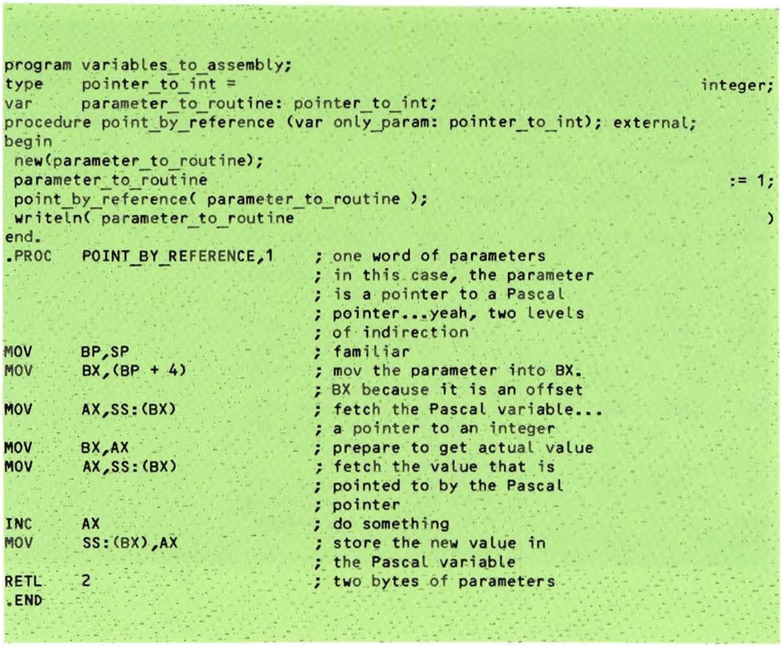
```
program variables_to_assembly;
var     parameter_to_routine: integer;
procedure int_by_reference (var only_param: integer); external;
begin
 parameter_to_routine := 1;
 int_by_reference( parameter_to_routine );
 writeln( parameter_to_routine )
end.
.PROC   INT_BY_REFERENCE,1               ; one word of parameters...
                                         ; in this case it is a pointer
                                         ; to the actual variable...
                                         ; all pointers are relative to
                                         ; the SS register at the start
                                         ; of an assembly routine
MOV     BP,SP                            ; familiar save of SP
MOV     BX,(BP + 4)                      ; move only parameter into
                                         ; BX.  BX is used because
                                         ; only certain registers may
                                         ; be used for a particular
                                         ; job...BX, SI or DI must
                                         ; be used when addressing
                                         ; through an offset
MOV     AX,SS:(BX)                       ; fetch the value of the
                                         ; parameter
INC     AX                               ; just to do something
MOV     SS:(BX),AX                       ; put the new value back
                                         ; into the variable for
                                         ; Pascal
RETL    2                                ; two bytes of parameters
.END
```

Example 3:
Passing Pointers By Value



```
program variables_to_assembly;
type     pointer_to_int =                                                  integer;
var      parameter_to_routine: pointer_to_int;
function point_by_value (only_param: pointer_to_int): integer; external;
begin
 new(parameter_to_routine);
 parameter_to_routine                                                        := 1;
 writeln(point_by_value( parameter_to_routine ))
end.
.FUNC    POINT_BY_VALUE,1     ; one word of parameter
                              ; in this case, the actual
                              ; value of a pascal pointer
                              ; will be passed
MOV      BP,SP                ; familiar
MOV      BX,(BP + 4)          ; mov the parameter into
                              ; BX...this will be a Pascal
                              ; pointer which is relative
                              ; to the SS register
MOV      AX,SS:(BX)           ; using the parameter as a
                              ; pointer...access the value
                              ; of the variable
INC      AX                   ; do something
MOV      (BP+6),AX            ; store the new value into
                              ; the function return word
RETL     2                    ; two bytes of parameters
.END
```

Example 4: Passing Pointers By Reference



```
program variables_to_assembly;
type    pointer_to_int =                                            integer;
var     parameter_to_routine: pointer_to_int;
procedure point_by_reference (var only_param: pointer_to_int); external;
begin
 new(parameter_to_routine);
 parameter_to_routine                                                  := 1;
 point_by_reference( parameter_to_routine );
 writeln( parameter_to_routine                                            )
end.
.PROC   POINT_BY_REFERENCE,1     ; one word of parameters
                                 ; in this case, the parameter
                                 ; is a pointer to a Pascal
                                 ; pointer...yeah, two levels
                                 ; of indirection
MOV     BP,SP                    ; familiar
MOV     BX,(BP + 4)              ; mov the parameter into BX.
                                 ; BX because it is an offset
MOV     AX,SS:(BX)               ; fetch the Pascal variable...
                                 ; a pointer to an integer
MOV     BX,AX                    ; prepare to get actual value
MOV     AX,SS:(BX)               ; fetch the value that is
                                 ; pointed to by the Pascal
                                 ; pointer
INC     AX                       ; do something
MOV     SS:(BX),AX               ; store the new value in
                                 ; the Pascal variable
RETL    2                        ; two bytes of parameters
.END
```

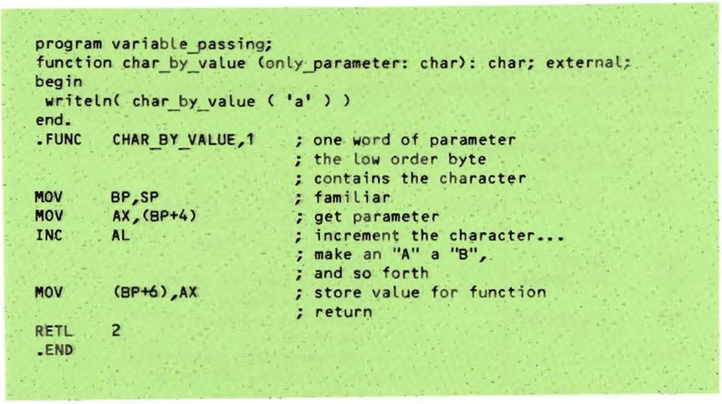
Example 5: Passing Reals By Value

```
program variable_passing;
function real_by_value (only_parameter: real): real; external;
begin
 writeln(real_by_value(10.0):4:1)
end.
.FUNC   REAL_BY_VALUE,4    ; 4 words of parameters
                           ; because reals are stored
                           ; as four-word numbers
MOV     BP,SP              ; familiar
MOV     AX,(BP+4)          ; last word of parameter...
                           ; the low-order bytes of
                           ; the mantissa
MOV     BX,6.
MOV     NUMBER(BX),AX      ; store the value
MOV     AX,(BP+6)          ; next word of parameter
MOV     BX,4.
MOV     NUMBER(BX),AX      ; store the value
MOV     AX,(BP+8)          ; next word of parameter
MOV     BX,2.
MOV     NUMBER(BX),AX      ; store the value
MOV     AX,(BP+10)         ; first word of parameter...
                           ; contains high-order byte
                           ; of mantissa and the exponent
MOV     NUMBER,AX          ; store the value
{ do something with the number, in this case multiply it by ten...
  for example, increment the exponent by one }
MOV     AX,NUMBER
INC     AH                 ; exponent is high-order byte
MOV     NUMBER,AX
{ the next section stores the new values into the stack for
  the function return to Pascal }
MOV     BX,6.
MOV     AX,NUMBER(BX)
MOV     (BP+12),AX
MOV     BX,4.
MOV     AX,NUMBER(BX)
MOV     (BP+14),AX
MOV     BX,2.
MOV      AX,NUMBER(BX)
MOV      (BP+16),AX
MOV      AX,NUMBER
MOV      (BP+18),AX
RETL     8
NUMBER   .BLOCK  8
.END
```

Example 6:
Passing Reals By Reference

```
program variable_passing;
var      param: real ;
procedure real_by_reference (var only_parameter: real); external;
begin
 param := 10.0;
 real_by_reference(param);
 writeln(param:4:1)
end.
.PROC    REAL_BY_REFERENCE,1      ; one word of parameter
                                  ; a pointer to the real
                                  ; variable
MOV      BP,SP                    ; familiar
MOV      BX,(BP+4)                ; mov the address of the
                                  ; variable into the
                                  ; "address" register
MOV      AX,SS:(BX+6)             ; fetch the last word of the
                                  ; variable ( a four word
                                  ; real, last word is six
                                  ; bytes offset )
INC      AH                       ; increment the exponent...
                                  ; stored in the high_order
                                  ; byte
MOV      SS:(BX+6),AX             ; store the new value
RETL     2
.END
```

Example 7:
Passing Characters By Value



```
program variable_passing;
function char_by_value (only_parameter: char): char; external;
begin
 writeln( char_by_value ( 'a' ) )
end.
.FUNC    CHAR_BY_VALUE,1     ; one word of parameter
                             ; the low order byte
                             ; contains the character
MOV      BP,SP               ; familiar
MOV      AX,(BP+4)           ; get parameter
INC      AL                  ; increment the character...
                             ; make an "A" a "B",
                             ; and so forth
MOV      (BP+6),AX           ; store value for function
                             ; return
RETL     2
.END
```

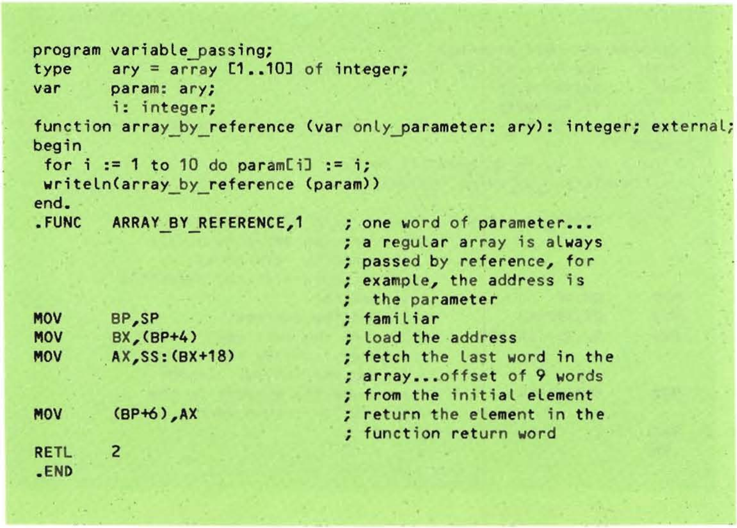
Example 8: Passing Characters By Reference

```
program variable_passing;
var     param: char;
procedure char_by_reference (var only_parameter: char); external;
begin
 param := 'a';
 char_by_reference (param);
 writeln(param)
end.
.PROC   CHAR_BY_REFERENCE,1     ; one word of parameter
                                ; is a pointer to a
                                ; character variable
MOV     BP,SP                   ; familiar
MOV     BX,(BP+4)               ; get the address of the
                                ; actual variable
MOV     AX,SS:(BX)              ; fetch the value of the
                                ; variable
INC     AL                      ; increment the character...
                                ; for example, "A" to "B"
MOV     SS:(BX),AX              ; restore variable
RETL    2
.END
```

Example 9:
Passing Arrays By Value

```
program variable_passing;
type    ary = array [1..10] of integer;
var     param: ary;
        i: integer;
function array_by_value (only_parameter: ary): integer; external;
begin
 for i := 1 to 10 do param[i] := i;
 writeln(array_by_value (param))
end.
.FUNC   ARRAY_BY_VALUE,1       ; one word of parameter...
                               ; a regular array is always
                               ; passed by reference, ie.
                               ; the address is the parameter
MOV     BP,SP                  ; familiar
MOV     BX,(BP+4)              ; load the address
MOV     AX,SS:(BX+18)          ; fetch the last word in the
                               ; array...offset of 9 words
                               ; from the initial element
MOV     (BP+6),AX              ; return the element in the
                               ; function return word
RETL    2
.END
```

Example 10:
Passing Arrays By Reference

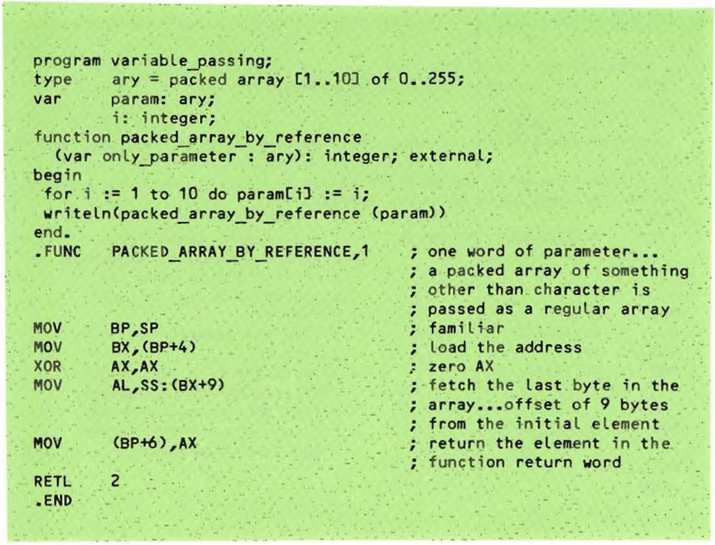


```
program variable_passing;
type    ary = array [1..10] of integer;
var     param: ary;
        i: integer;
function array_by_reference (var only_parameter: ary): integer; external;
begin
 for i := 1 to 10 do param[i] := i;
 writeln(array_by_reference (param))
end.
.FUNC   ARRAY_BY_REFERENCE,1     ; one word of parameter...
                                 ; a regular array is always
                                 ; passed by reference, for
                                 ; example, the address is
                                 ;  the parameter
MOV     BP,SP                    ; familiar
MOV     BX,(BP+4)                ; load the address
MOV     AX,SS:(BX+18)            ; fetch the last word in the
                                 ; array...offset of 9 words
                                 ; from the initial element
MOV     (BP+6),AX                ; return the element in the
                                 ; function return word
RETL    2
.END
```

Example 11: Passing Packed Arrays By Value

```
program variable_passing;
type     ary = packed array [1..10] of 0..255;
var      param: ary;
         i: integer;
function packed_array_by_value (only_parameter: ary): integer; external;
begin
 for i := 1 to 10 do param[i] := i;
 writeln(packed_array_by_value (param))
end.
.FUNC    PACKED_ARRAY_BY_VALUE,1     ; one word of parameter...
                                     ; a packed array of something
                                     ; other than character is
                                     ; passed as a regular array
MOV      BP,SP                       ; familiar
MOV      BX,(BP+4)                   ; load the address
XOR      AX,AX                       ; zero AX
MOV      AL,SS:(BX+9)                ; fetch the last byte in the
                                     ; array...offset of 9 bytes
                                     ; from the initial element
MOV      (BP+6),AX                   ; return the element in the
                                     ; function return word
RETL     2
.END
```

Example 12:
Passing Packed Arrays By Reference

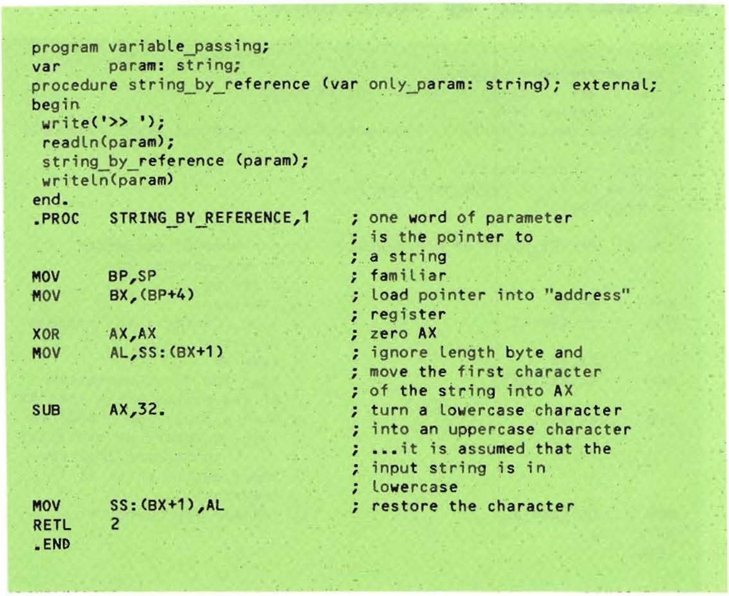


```
program variable_passing;
type    ary = packed array [1..10] of 0..255;
var     param: ary;
        i: integer;
function packed_array_by_reference
  (var only_parameter : ary): integer; external;
begin
 for i := 1 to 10 do param[i] := i;
 writeln(packed_array_by_reference (param))
end.
.FUNC   PACKED_ARRAY_BY_REFERENCE,1     ; one word of parameter...
                                        ; a packed array of something
                                        ; other than character is
                                        ; passed as a regular array
MOV     BP,SP                           ; familiar
MOV     BX,(BP+4)                       ; load the address
XOR     AX,AX                           ; zero AX
MOV     AL,SS:(BX+9)                    ; fetch the last byte in the
                                        ; array...offset of 9 bytes
                                        ; from the initial element
MOV     (BP+6),AX                       ; return the element in the
                                        ; function return word
RETL    2
.END
```

Example 13: Passing Strings or Packed Arrays of Character By Value

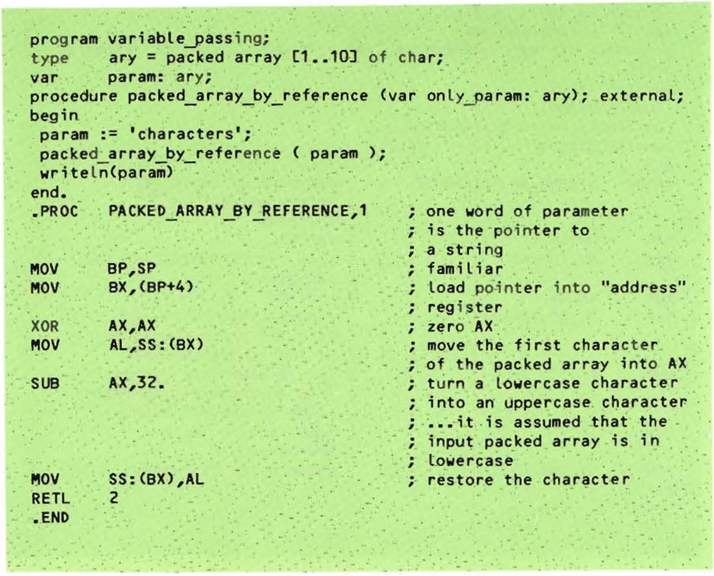
```
program variable_passing;
function string_by_value (only_param: string): char; external;
begin
 writeln( string_by_value ( 'something' ) )
end.
     .FUNC    STRING_BY_VALUE,2       ; Identical to Packed
                                      ; Array of Char by Value
                                      ; two words of parameters
                                      ; are a segment pointer
                                      ; to the string parameter
     MOV      BP,SP                   ; familiar
     MOV      AX,(BP+4)               ; TOS...if NIL, for
                                      ; example, = 0, next word
                                      ; is a pointer, if not
                                      ; NIL, for example, <> 0,
                                      ; strange things...
     { NIL is an implementation dependent value...here it is
       assumed to be equal to 0...this may not necessarily
       be the case }
     TEST     AX,0.
     JE       EASY
HARD
     { not NIL...therefore, is a pointer to a Segment
       Environment Record, the third word of which is a
       pointer to the SIB, hence the 4 in the next
       statement.  The second word of the SIB is the
       pointer to the actual segment that contains the
       string. }
     MOV      BX,AX                   ; load "address" register
     MOV      DI,SS:(BX + 4)          ; get address of SIB
     MOV      BX,SS:(DI + 2)          ; get address of base of
                                      ; actual segment
     MOV      AX,(BP+6)               ; get next word of parameter...
                                      ; this is the offset into
                                      ; the actual segment for
                                      ; the string
     ADD      BX,AX                   ; compute pointer to string...
                                      ; <pointer to segment> plus
                                      ; <offset>
     JMP      FOUND                   ; we now have the address of
                                      ; the string in BX...jump
                                      ; to do the work
EASY
     { is NIL...therefore the second word on
       the stack is the pointer to the string }
     MOV      BX,(BP+6)
FOUND
     { we now have the address of the string in BX }
     XOR      AX,AX                   ; zero AX
     MOV      AL,SS:(BX+1)            ; fetch the first character...
                                      ; ignore the length byte
     MOV      (BP+8),AX               ; put the character into the
                                      ; function return word
                                      ; on the stack
     RETL     4
   .END
```

Example 14: Passing Strings By Reference



```
program variable_passing;
var     param: string;
procedure string_by_reference (var only_param: string); external;
begin
 write('>> ');
 readln(param);
 string_by_reference (param);
 writeln(param)
end.
.PROC   STRING_BY_REFERENCE,1      ; one word of parameter
                                   ; is the pointer to
                                   ; a string
MOV     BP,SP                      ; familiar
MOV     BX,(BP+4)                  ; load pointer into "address"
                                   ; register
XOR     AX,AX                      ; zero AX
MOV     AL,SS:(BX+1)               ; ignore length byte and
                                   ; move the first character
                                   ; of the string into AX
SUB     AX,32.                     ; turn a lowercase character
                                   ; into an uppercase character
                                   ; ...it is assumed that the
                                   ; input string is in
                                   ; lowercase
MOV     SS:(BX+1),AL               ; restore the character
RETL    2
.END
```

Example 15: Passing Packed Arrays of Character by Reference



```
program variable_passing;
type     ary = packed array [1..10] of char;
var      param: ary;
procedure packed_array_by_reference (var only_param: ary); external;
begin
 param := 'characters';
 packed_array_by_reference ( param );
 writeln(param)
end.
.PROC    PACKED_ARRAY_BY_REFERENCE,1       ; one word of parameter
                                           ; is the pointer to
                                           ; a string
MOV      BP,SP                             ; familiar
MOV      BX,(BP+4)                         ; load pointer into "address"
                                           ; register
XOR      AX,AX                             ; zero AX
MOV      AL,SS:(BX)                        ; move the first character
                                           ; of the packed array into AX
SUB      AX,32.                            ; turn a lowercase character
                                           ; into an uppercase character
                                           ; ...it is assumed that the
                                           ; input packed array is in
                                           ; lowercase
MOV      SS:(BX),AL                        ; restore the character
RETL     2
.END
```

Example 16:
Passing Records By Value

```
program variable_passing;
 type    rec = record
                 i_am_2nd,i_am_1st: integer;
                 i_am_4th,i_am_3rd: char;
               end;
 var     param: rec;
 function record_by_value (only_param: rec): char; external;
 begin
  with param do
   begin
    i_am_2nd := 1;
    i_am_1st := 2;
    i_am_4th := 'a';
    i_am_3rd := 'b';
   end;
  writeln( record_by_value ( param ) )
 end.
 .FUNC    RECORD_BY_VALUE,1       ; one word of parameter... a
                                  ; record is passed exactly
                                  ; the same whether it is a
                                  ; value or a reference
                                  ; parameter...a pointer to the
                                  ; structure is on the top
                                  ; of the stack
 MOV      BP,SP                   ; familiar
 MOV      BX,(BP+4)               ; access the pointer
 MOV      AX,SS:(BX)              ; access the first word
                                  ; of the record...the last
                                  ; variable in the first
                                  ; field_declaration_list,
                                  ; in this case an integer,
                                  ; done as an example
 { the following is an example of accessing another field
   in the record, in this case, the third word of the record
   contains a char (the last variable in the second
   declaration list).  As a char is stored in the
   low-order byte of the word, the offset should be even
   address of the word. }
 XOR      DX,DX                   ; zero DX
 MOV      DL,SS:(BX+4)            ; access the character and
                                  ; store it in the low-order
                                  ; byte of DX
 MOV      (BP+6),DX               ; place the character in the
                                  ; function return word
 RETL     2
 .END
```

Example 17: Passing Records By Reference

```
program variable_passing;
type    rec = record
                i_am_2nd,i_am_1st: integer;
                i_am_4th,i_am_3rd: char;
              end;
var     param : rec;
procedure record_by_reference (var only_param: rec); external;
begin
 with param do
   begin
     i_am_2nd := 1;
     i_am_1st := 2;
     i_am_4th := 'a';
     i_am_3rd := 'b';
    end;
  writeln('before call');
  with param do
    begin
      writeln('i_am_1st ',i_am_1st);
      writeln('i_am_2nd ',i_am_2nd);
      writeln('i_am_3rd ',i_am_3rd);
      writeln('i_am_4th ',i_am_4th)
    end;
  record_by_reference (param);
  writeln('after call');
   with param do
    begin
      writeln('i_am_1st ',i_am_1st);
      writeln('i_am_2nd ',i_am_2nd);
      writeln('i_am_3rd ',i_am_3rd);
      writeln('i_am_4th ',i_am_4th)
    end
 end.
.PROC   RECORD_BY_REFERENCE,1    ; one word of parameter
                                 ; is a pointer to a structure
MOV     BP,SP                    ; familiar
MOV     BX,(BP+4)                ; access the parameter
{ this routine switches the values of the variables
  in the record...the first and second are both integers
  and the third and the fourth are characters }
MOV     AX,SS:(BX)               ; get first word of record
MOV     DX,SS:(BX+2)             ; get second word of record
MOV     SS:(BX),DX               ; restore
MOV     SS:(BX+2),AX             ; variables
XOR     AX,AX                    ; zero AX
XOR     DX,DX                    ; zero DX
MOV     AL,SS:(BX+4)             ; get low-order byte of
                                 ; the third word
MOV     DL,SS:(BX+6)             ; get low-order byte of
                                 ; the fourth word
MOV     SS:(BX+4),DL             ; restore
MOV      SS:(BX+6),AL              ; variables
RETL     2
.END
```

Example 18:
Multiple Parameter Passing

```
program strange_params;
type     rec = record
                field1: array[1..10] of integer;
                field2,field3: char;
               end;
var      param1,param2: rec;
         i: integer;
procedure multi_params
   (value_rec: rec; var reference_rec: rec); external;
begin
 with param1 do
   begin
     for i := 1 to 10 do field1[i] := i;
     field2 := 'a';
     field3 := 'b';
   end;
 multi_params ( param1,param2 );
 with param2 do
   begin
    for i := 1 to 10 do writeln('element',i,' ',field1[i]);
    writeln('field2 ',field2);
    writeln('field3 ',field3)
   end;
end.
     .PROC    MULTI_PARAMS,2          ; two words of parameters
                                      ; TOS is a pointer to a
                                      ; record passed as a reference
                                      ; parameter...TOS-1 is
                                      ; a pointer to a record
                                      ; passed as a value parameter
     MOV      BP,SP
     MOV      BX,(BP+6)               ; access TOS-1 for the address
                                      ; of the value parameter
     MOV      DI,(BP+4)               ; access TOS for the address
                                      ; of the reference parameter
     ADD      DI,18.                  ; the first field of the record
                                      ; is a ten element array of
                                      ; integers, therefore the
                                      ; offset of the last element
                                      ; is 9 words or 18 bytes...
     MOV      CX,10.                  ; there are 10 elements in the
                                      ; array
    { the following loop reads the 10 elements of the array
      in the value parameter and stores them in reverse order
      in the array in the reference parameter...that is why
      the offset of the last element is needed (see above). }
START   MOV     AX,SS:(BX)          ; load next element
        MOV     SS:(DI),AX          ; store it
        INC     BX                  ; the next element is
        INC     BX                  ; 2 bytes offset
        DEC     DI                  ; back up to previous
        DEC     DI                  ; element...2 bytes
        LOOPNZ  START               ; decrement CX, if not
                                    ; 0 then loop to START
        ADD     DI,22.              ; access next element past
                                    ; the array in the reference
                                    ; parameter
        MOV     AX,SS:(BX)          ; load the next field from
                                    ; the value parameter
        MOV     SS:(DI+2),AX        ; store it in the last field
                                    ; of the reference parameter
        MOV     AX,SS:(BX+2)        ; load the last field from
                                    ; the value parameter
        MOV     SS:(DI),AX          ; store it in the next-to-last
                                    ; field of the reference param
        RETL    4
    .END
```

Example 19: Program to Determine NIL

NIL is a machine-dependent value. If you want to determine what NIL is for your system, you can use the following Pascal program. Note that the value of NIL for each processor is listed in Appendix N.

```
program find_nil;
type   trix = record
                case boolean of
                true: (x: integer);
                false: (y:                                          integer);
              end;
var   p: trix;
begin
  p.y := nil;
  writeln (p.x);
end.
```

USEFUL ROUTINES

This section contains some example routines that might be found generally useful.

```
function readport (port: integer): integer; external;
procedure writport (port, value: integer); external;
procedure readmemory
   (segmnt, offset: integer; var result: integer); external;
function lookup (entry: integer): integer; external;
```

The first routine, below, reads a byte from an I/O port. The second routine writes a byte to an I/O port. The third routine reads an arbitrary byte from memory. The last two routines work together to quickly look up an item in a table.

```
        .FUNC    READPORT,1          ; read byte I/O port
PORT    .EQU     4                   ; port number to read from
RESULT  .EQU     6                   ; result of function
ENTRY    MOV     BP,SP               ; point to parameters
         MOV     DX,(BP+PORT)        ; fetch port number
         IN      AL,DX               ; read byte from port
         XOR     AH,AH               ; put zero to extend to word
         MOV     (BP+RESULT),AX      ; set returned result
         RETL    2                   ; cut stack by 2 bytes for parameter
        .PROC    WRITPORT,2          ; write byte I/O port
VALUE   .EQU     4                   ; value to write
PORT    .EQU     6
         MOV     BP,SP
         MOV     DX,(BP+PORT)
         MOV     AL,(BP+VALUE)       ; fetch value to write
         OUT     DX,AL               ; byte output value
         RETL    4                   ; cut back two parameters words
        .RELPROC READMEMORY,3        ; read word of memory
VARPTR  .EQU     4                   ; pointer to variable
OFFSET  .EQU     6                   ; pointer to memory
SEGMENT .EQU     8                   ; segment of memory
         MOV     BP,SP               ; point to parameters
         LDS     BX,(BP+OFFSET)      ; fetch extended pointer
         MOV     AX,(BX)             ; memory word
         MOV     DI,(BP+VARPTR)      ; pointer to variable
         MOV     SS:(DI),AX          ; store in variable in stack segment
         RETL    6                   ; pop three parameters
        .RELPROC PRIMES
        .DEF     TABLE
TABLE   .WORD    1,2,3,5,7,11,13,17,23
        .RELFUNC LOOKUP,1
        .REF     TABLE
LAST    .EQU     8
ENTRY   .EQU     4
RESULT  .EQU     6
         MOV     BP,SP
         MOV     BX,(BP+ENTRY)       ; fetch index
         CMP     BX,LAST             ; check range
         JA      $01                 ; do nothing if too high
         MOV     SI,BX               ; copy to index register
         MOV     AX,TABLE(BX)(SI)    ; tricky word index
         MOV     (BP+RESULT),AX      ; store result
$01      RETL    2
        .END
```

APPENDIX D
6502 SYNTAX ERRORS

1: undefined label
2: operand out of range
3: must have procedure name
4: number of parameters expected
5: extra symbols on source line
6: input line over 80 characters
7: unmatched conditional assembly directive
8: must be declared in .ASECT before used
9: identifier previously declared
10: improper format
11: illegal character in text
12: must .EQU before use if not to a label
13: macro identifier expected
14: code file too large
15: backwards .ORG not allowed
16: identifier expected
17: constant expected
18: invalid structure
19: extra special symbol
20: branch too far
21: LC-relative to externals not allowed
22: illegal macro parameter index
23: illegal macro parameter
24: operand not absolute
25: illegal use of special symbols
26: ill-formed expression
27: not enough operands
28: LC-relative to absolutes unrelocatable
29: constant overflow
30: illegal decimal constant
31: illegal octal constant
32: illegal binary constant
33: invalid key word
34: unmatched macro definition directive
35: include files may not be nested

36: unexpected end of input
37: .INCLUDE not allowed in macros
38: label expected
39: expected local label
40: local label stack overflow
41: string constants must be on single line
42: string constant exceeds 80 characters
43: cannot handle this relocate count
44: no local labels in .ASECT
45: expected key word
46: string expected
47: I/O - bad block, parity error (CRC)
48: I/O - illegal unit number
49: I/O - illegal operation on unit
50: I/O - undefined hardware error
51: I/O - unit no longer on-line
52: I/O - file no longer in directory
53: I/O - illegal file name
54: I/O - no room on disk
55: I/O - no such unit on-line
56: I/O - no such file on volume
57: I/O - duplicate file
58: I/O - attempted open of open file
59: I/O - attempted access of closed file
60: I/O - bad format in real or integer
61: I/O - ring buffer overflow
62: I/O - write to write-protected disk
63: I/O - illegal block number
64: I/O - illegal buffer address
65: nested macro definitions not allowed
66: '=' or '<>' expected
67: may not equate to undefined labels
68: .ABSOLUTE must appear before 1st proc
69: .PROC or .FUNC expected
70: too many procedures
71: only absolute expressions in .ASECT
72: must be label expression
73: no operands allowed in .ASECT
74: offset not word-aligned

75: LC not word-aligned
76: index register required
77: 'X' or 'Y' expected
78: zero-page address required
79: illegal use of register
80: index register expected
81: ill-formed operand
82: 'X' expected for indexed addressing
83: must use 'X' index register
84: must use 'Y' index register

APPENDIX E
6800 SYNTAX ERRORS

1: undefined label
2: operand out of range
3: must have procedure name
4: number of parameters expected
5: extra symbols on source line
6: input line over 80 characters
7: unmatched conditional assembly directive
8: must be declared in .ASECT before used
9: identifier previously declared
10: improper format
11: illegal character in text
12: must .EQU before use if not to a label
13: macro identifier expected
14: code file too large
15: backwards .ORG not allowed
16: identifier expected
17: constant expected
18: invalid structure
19: extra special symbol
20: branch too far
21: LC-relative to externals not allowed
22: illegal macro parameter index
23: illegal macro parameter
24: operand not absolute
25: illegal use of special symbols
26: ill-formed expression
27: not enough operands
28: LC-relative to absolutes unrelocatable
29: constant overflow
30: illegal decimal constant
31: illegal octal constant
32: illegal binary constant
33: invalid key word
34: unmatched macro definition directive
35: include files may not be nested

36: unexpected end of input

37: .INCLUDE not allowed in macros

38: label expected

39: expected local label

40: local label stack overflow

41: string constants must be on single line

42: string constant exceeds 80 characters

43: cannot handle this relocate count

44: no local labels in .ASECT

45: expected key word

46: string expected

47: I/O - bad block, parity error (CRC)

48: I/O - illegal unit number

49: I/O - illegal operation on unit

50: I/O - undefined hardware error

51: I/O - unit no longer on-line

52: I/O - file no longer in directory

53: I/O - illegal file name

54: I/O - no room on disk

55: I/O - no such unit on-line

56: I/O - no such file on volume

57: I/O - duplicate file

58: I/O - attempted open of open file

59: I/O - attempted access of closed file

60: I/O - bad format in real or integer

61: I/O - ring buffer overflow

62: I/O - write to write-protected disk

63: I/O - illegal block number

64: I/O - illegal buffer address

65: nested macro definitions not allowed

66: `=' or `<>` expected

67: may not equate to undefined labels

68: .ABSOLUTE must appear before 1st proc

69: .PROC or .FUNC expected

70: too many procedures

71: only absolute expressions in .ASECT

72: must be label expression

73: no operands allowed in .ASECT

74: offset not word-aligned

75: LC not word-aligned
76: 'X' expected for indexed addressing
77: 'A' or 'B' expected
78: invalid operand
79: comma expected

APPENDIX F
6809 SYNTAX ERRORS

1: undefined label
2: operand out of range
3: must have procedure name
4: number of parameters expected
5: extra symbols on source line
6: input line over 80 characters
7: unmatched conditional assembly directive
8: must be declared in .ASECT before used
9: identifier previously declared
10: improper format
11: illegal character in text
12: must .EQU before use if not to a label
13: macro identifier expected
14: code file too large
15: backwards .ORG not allowed
16: identifier expected
17: constant expected
18: invalid structure
19: extra special symbol
20: branch too far
21: LC-relative to externals not allowed
22: illegal macro parameter index
23: illegal macro parameter
24: operand not absolute
25: illegal use of special symbols
26: ill-formed expression
27: not enough operands
28: LC-relative to absolutes unrelocatable
29: constant overflow
30: illegal decimal constant
31: illegal octal constant
32: illegal binary constant
33: invalid key word
34: unmatched macro definition directive
35: include files may not be nested

36: unexpected end of input
37: .INCLUDE not allowed in macros
38: label expected
39: expected local label
40: local label stack overflow
41: string constants must be on single line
42: string constant exceeds 80 characters
43: cannot handle this relocate count
44: no local labels in .ASECT
45: expected key word
46: string expected
47: I/O - bad block, parity error (CRC)
48: I/O - illegal unit number
49: I/O - illegal operation on unit
50: I/O - undefined hardware error
51: I/O - unit no longer on-line
52: I/O - file no longer in directory
53: I/O - illegal file name
54: I/O - no room on disk
55: I/O - no such unit on-line
56: I/O - no such file on volume
57: I/O - duplicate file
58: I/O - attempted open of open file
59: I/O - attempted access of closed file
60: I/O - bad format in real or integer
61: I/O - ring buffer overflow
62: I/O - write to write-protected disk
63: I/O - illegal block number
64: I/O - illegal buffer address
65: nested macro definitions not allowed
66: '=' or '<>' expected
67: may not equate to undefined labels
68: .ABSOLUTE must appear before 1st proc
69: .PROC or .FUNC expected
70: too many procedures
71: only absolute expressions in .ASECT
72: must be label expression
73: no operands allowed in .ASECT
74: offset not word-aligned

75: LC not word-aligned
76: immediate operand expected
77: invalid register list entry
78: operand must be indexed
79: invalid index register
80: no offset allowed
81: indirect not allowed
82: invalid offset register
83: invalid offset
84: immediate not allowed
85: registers are incompatible

APPENDIX G
8080 SYNTAX ERRORS

1: undefined label
2: operand out of range
3: must have procedure name
4: number of parameters expected
5: extra symbols on source line
6: input line over 80 characters
7: unmatched conditional assembly directive
8: must be declared in .ASECT before used
9: identifier previously declared
10: improper format
11: illegal character in text
12: must .EQU before use if not to a label
13: macro identifier expected
14: code file too large
15: backwards .ORG not allowed
16: identifier expected
17: constant expected
18: invalid structure
19: extra special symbol
20: branch too far
21: LC-relative to externals not allowed
22: illegal macro parameter index
23: illegal macro parameter
24: operand not absolute
25: illegal use of special symbols
26: ill-formed expression
27: not enough operands
28: LC-relative to absolutes unrelocatable
29: constant overflow
30: illegal decimal constant
31: illegal octal constant
32: illegal binary constant
33: invalid key word
34: unmatched macro definition directive
35: include files may not be nested

36: unexpected end of input
37: .INCLUDE not allowed in macros
38: label expected
39: expected local label
40: local label stack overflow
41: string constants must be on single line
42: string constant exceeds 80 characters
43: cannot handle this relocate count
44: no local labels in .ASECT
45: expected key word
46: string expected
47: I/O - bad block, parity error (CRC)
48: I/O - illegal unit number
49: I/O - illegal operation on unit
50: I/O - undefined hardware error
51: I/O - unit no longer on-line
52: I/O - file no longer in directory
53: I/O - illegal file name
54: I/O - no room on disk
55: I/O - no such unit on-line
56: I/O - no such file on volume
57: I/O - duplicate file
58: I/O - attempted open of open file
59: I/O - attempted access of closed file
60: I/O - bad format in real or integer
61: I/O - ring buffer overflow
62: I/O - write to write-protected disk
63: I/O - illegal block number
64: I/O - illegal buffer address
65: nested macro definitions not allowed
66: '=' or '<>' expected
67: may not equate to undefined labels
68: .ABSOLUTE must appear before 1st proc
69: .PROC or .FUNC expected
70: too many procedures
71: only absolute expressions in .ASECT
72: must be label expression
73: no operands allowed in .ASECT
74: offset not word-aligned

75: LC not word-aligned
76: invalid operand
77: comma expected

APPENDIX H
9900 SYNTAX ERRORS

1: undefined label
2: operand out of range
3: must have procedure name
4: number of parameters expected
5: extra symbols on source line
6: input line over 80 characters
7: unmatched conditional assembly directive
8: must be declared in .ASECT before used
9: identifier previously declared
10: improper format
11: illegal character in text
12: must .EQU before use if not to a label
13: macro identifier expected
14: code file too large
15: backwards .ORG not allowed
16: identifier expected
17: constant expected
18: invalid structure
19: extra special symbol
20: branch too far
21: LC-relative to externals not allowed
22: illegal macro parameter index
23: illegal macro parameter
24: operand not absolute
25: illegal use of special symbols
26: ill-formed expression
27: not enough operands
28: LC-relative to absolutes unrelocatable
29: constant overflow
30: illegal decimal constant
31: illegal octal constant
32: illegal binary constant
33: invalid key word
34: unmatched macro definition directive
35: include files may not be nested

36: unexpected end of input
37: .INCLUDE not allowed in macros
38: label expected
39: expected local label
40: local label stack overflow
41: string constants must be on single line
42: string constant exceeds 80 characters
43: cannot handle this relocate count
44: no local labels in .ASECT
45: expected key word
46: string expected
47: I/O - bad block, parity error (CRC)
48: I/O - illegal unit number
49: I/O - illegal operation on unit
50: I/O - undefined hardware error
51: I/O - unit no longer on-line
52: I/O - file no longer in directory
53: I/O - illegal file name
54: I/O - no room on disk
55: I/O - no such unit on-line
56: I/O - no such file on volume
57: I/O - duplicate file
58: I/O - attempted open of open file
59: I/O - attempted access of closed file
60: I/O - bad format in real or integer
61: I/O - ring buffer overflow
62: I/O - write to write-protected disk
63: I/O - illegal block number
64: I/O - illegal buffer address
65: nested macro definitions not allowed
66: '=' or '<>' expected
67: may not equate to undefined labels
68: .ABSOLUTE must appear before 1st proc
69: .PROC or .FUNC expected
70: too many procedures
71: only absolute expressions in .ASECT
72: must be label expression
73: no operands allowed in .ASECT
74: offset not word-aligned

75: LC not word-aligned

76: illegal immediate operand

77: index must be WR

78: close paren ')' expected

79: indirect & autoincr must be WR

80: autoincr must be WR indirect

81: comma ',' expected

82: no operand allowed

83: illegal map file

84: WR expected

APPENDIX I
LSI-11/PDP-11 SYNTAX ERRORS

1: undefined label
2: operand out of range
3: must have procedure name
4: number of parameters expected
5: extra symbols on source line
6: input line over 80 characters
7: unmatched conditional assembly directive
8: must be declared in .ASECT before used
9: identifier previously declared
10: improper format
11: illegal character in text
12: must .EQU before use if not to a label
13: macro identifier expected
14: code file too large
15: backwards .ORG not allowed
16: identifier expected
17: constant expected
18: invalid structure
19: extra special symbol
20: branch too far
21: LC-relative to externals not allowed
22: illegal macro parameter index
23: illegal macro parameter
24: operand not absolute
25: illegal use of special symbols
26: ill-formed expression
27: not enough operands
28: LC-relative to absolutes unrelocatable
29: constant overflow
30: illegal decimal constant
31: illegal octal constant
32: illegal binary constant
33: invalid key word
34: unmatched macro definition directive
35: include files may not be nested

36: unexpected end of input
37: .INCLUDE not allowed in macros
38: label expected
39: expected local label
40: local label stack overflow
41: string constants must be on single line
42: string constant exceeds 80 characters
43: cannot handle this relocate count
44: no local labels in .ASECT
45: expected key word
46: string expected
47: I/O - bad block, parity error (CRC)
48: I/O - illegal unit number
49: I/O - illegal operation on unit
50: I/O - undefined hardware error
51: I/O - unit no longer on-line
52: I/O - file no longer in directory
53: I/O - illegal file name
54: I/O - no room on disk
55: I/O - no such unit on-line
56: I/O - no such file on volume
57: I/O - duplicate file
58: I/O - attempted open of open file
59: I/O - attempted access of closed file
60: I/O - bad format in real or integer
61: I/O - ring buffer overflow
62: I/O - write to write-protected disk
63: I/O - illegal block number
64: I/O - illegal buffer address
65: nested macro definitions not allowed
66: '=' or '<>' expected
67: may not equate to undefined labels
68: .ABSOLUTE must appear before 1st proc
69: .PROC or .FUNC expected
70: too many procedures
71: only absolute expressions in .ASECT
72: must be label expression
73: no operands allowed in .ASECT
74: offset not word-aligned

75: LC not word-aligned
76: close paren ')' expected
77: register expected
78: too many special symbols
79: unrecognizable operand
80: register reference only
81: first operand must be register
82: comma ',' expected
83: unimplemented instruction
84: must branch backwards to label

APPENDIX J
Z8 SYNTAX ERRORS

1: undefined label
2: operand out of range
3: must have procedure name
4: number of parameters expected
5: extra symbols on source line
6: input line over 80 characters
7: unmatched conditional assembly directive
8: must be declared in .ASECT before used
9: identifier previously declared
10: improper format
11: invalid radix
12: must .EQU before use if not to a label
13: macro identifier expected
14: code file too large
15: backwards .ORG not allowed
16: identifier expected
17: constant expected
18: invalid structure
19: extra special symbol
20: branch too far
21: LC-relative to externals not allowed
22: illegal macro parameter index
23: illegal macro parameter
24: operand not absolute
25: illegal use of special symbols
26: ill-formed expression
27: not enough operands
28: LC-relative to absolutes unrelocatable
29: constant overflow
30: illegal decimal constant
31: illegal octal constant
32: illegal binary constant
33: invalid key word
34: unmatched macro definition directive
35: include files may not be nested

```
36: unexpected end of input
37: .INCLUDE not allowed in macros
38: label expected
39: expected local label
40: local label stack overflow
41: string constants must be on single line
42: string constant exceeds 80 characters
43: cannot handle this relocate count
44: no local labels in .ASECT
45: expected key word
46: string expected
47: I/O - bad block, parity error (CRC)
48: I/O - illegal unit number
49: I/O - illegal operation on unit
50: I/O - undefined hardware error
51: I/O - unit no longer on-line
52: I/O - file no longer in directory
53: I/O - illegal file name
54: I/O - no room on disk
55: I/O - no such unit on-line
56: I/O - no such file on volume
57: I/O - duplicate file
58: I/O - attempted open of open file
59: I/O - attempted access of closed file
60: I/O - bad format in real or integer
61: I/O - ring buffer overflow
62: I/O - write to write-protected disk
63: I/O - illegal block number
64: I/O - illegal buffer address
65: nested macro definitions not allowed
66: `'=' or ` <> ` expected
67: may not equate to undefined labels
68: .ABSOLUTE must appear before 1st proc
69: .PROC or .FUNC expected
70: too many procedures
71: only absolute expressions in .ASECT
72: only labels equated to .DEFs
73: no operands allowed in .ASECT
74: offset not word-aligned
```

75: LC not word-aligned
76: too many symbols
77: operand expected
78: bad data value
79: ")" expected
80: bad operand type
81: odd register
82: unknown instruction
83: working register expected
84: indirect or register expected
85: condition code expected

APPENDIX K
Z80 SYNTAX ERRORS

1: undefined label
2: operand out of range
3: must have procedure name
4: number of parameters expected
5: extra symbols on source line
6: input line over 80 characters
7: unmatched conditional assembly directive
8: must be declared in .ASECT before used
9: identifier previously declared
10: improper format
11: illegal character in text
12: must .EQU before use if not to a label
13: macro identifier expected
14: code file too large
15: backwards .ORG not allowed
16: identifier expected
17: constant expected
18: invalid structure
19: extra special symbol
20: branch too far
21: LC-relative to externals not allowed
22: illegal macro parameter index
23: illegal macro parameter
24: operand not absolute
25: illegal use of special symbols
26: bill-formed expression
27: not enough operands
28: LC-relative to absolutes unrelocatable
29: constant overflow
30: illegal decimal constant
31: illegal octal constant
32: illegal binary constant
33: invalid key word
34: unmatched macro definition directive
35: include files may not be nested

36: unexpected end of input
37: .INCLUDE not allowed in macros
38: label expected
39: expected local label
40: local label stack overflow
41: string constants must be on single line
42: string constant exceeds 80 characters
43: cannot handle this relocate count
44: no local labels in .ASECT
45: expected key word
46: string expected
47: I/O - bad block, parity error (CRC)
48: I/O - illegal unit number
49: I/O - illegal operation on unit
50: I/O - undefined hardware error
51: I/O - unit no longer on-line
52: I/O - file no longer in directory
53: I/O - illegal file name
54: I/O - no room on disk
55: I/O - no such unit on-line
56: I/O - no such file on volume
57: I/O - duplicate file
58: I/O - attempted open of open file
59: I/O - attempted access of closed file
60: I/O - bad format in real or integer
61: I/O - ring buffer overflow
62: I/O - write to write-protected disk
63: I/O - illegal block number
64: I/O - illegal buffer address
65: nested macro definitions not allowed
66: `=' or `<>` expected
67: may not equate to undefined labels
68: .ABSOLUTE must appear before 1st proc
69: .PROC or .FUNC expected
70: too many procedures
71: only absolute expressions in .ASECT
72: must be label expression
73: no operands allowed in .ASECT
74: offset not word-aligned

75: LC not word-aligned
76: incorrect operand format
77: close paren ')' expected
78: comma ',' expected
79: plus '+' expected
80: open paren '(' expected
81: stack pointer 'SP' expected
82: 'HL' expected
83: illegal 'cc' condition code
84: register 'C' expected
85: register expected 'r'
86: register 'A' expected

APPENDIX L
8086/88/87 SYNTAX ERRORS

1: undefined label
2: operand out of range
3: must have procedure name
4: number of parameters expected
5: extra symbols on source line
6: input line over 80 characters
7: unmatched conditional assembly
   directive
8: must be declared in .ASECT before
   used
9: identifier previously declared
10: improper format
11: illegal character in text
12: must .EQU before use if not to a label
13: macro identifier expected
14: code file too large
15: backwards .ORG not allowed
16: identifier expected
17: constant expected
18: invalid structure
19: extra special symbol
20: branch too far
21: LC-relative to externals not allowed
22: illegal macro parameter index
23: illegal macro parameter
24: operand not absolute
25: illegal use of special symbols
26: ill-formed expression
27: not enough operands
28: LC-realtive to absolutes
   unrelocatable
29: constant overflow
30: illegal decimal constant
31: illegal octal constant
32: illegal binary constant

33: invalid key word
34: unmatched macro definition directive
35: include files may not be nested
36: unexpected end of input
37: .INCLUDE not allowed in macros
38: label expected
39: expected local label
40: local label stack overflow
41: string constants must be on single line
42: string constants exceeds 80 characters
43: cannot handle this relocate count
44: no local labels in .ASECT
45: expected key word
46: string expected
47: I/O – bad block, parity error (CRC)
48: I/O – illegal unit number
49: I/O – illegal operation on unit
50: I/O – undefined hardware error
51: I/O – unit no longer on-line
52: I/O – file no longer in directory
53: I/O – illegal file name
54: I/O – no room on disk
55: I/O – no such unit on-line
56: I/O – no such file on volume
57: I/O – duplicate file
58: I/O – attempted open of open file
59: I/O – attempted access of closed file
60: I/O – bad format in real or integer
61: I/O – ring buffer overflow
62: I/O – write to write-protected disk

63: I/O — illegal block number
64: I/O — illegal buffer address
65: nested macro definitions not allowed
66: '=' or '<>' expected
67: may not equate to undefined labels
68: .ABSOLUTE must appear before first proc
69: .PROC or .FUNC expected
70: to many procedures (more than 10)
71: only absolute expressions in .ASECT
72: must be label expression
73: no operands allowed in .ASECT
74: offset not word-aligned
75: LC not word-aligned
76: had label, open parenthesis then illegality
77: expected absolute expression
78: both operands cannot be a seg register
79: illegal pair of index registers
80: have to use BX, BP, SI or DI
81: illegal constant as first operand
82: the first operand is needed
83: the second operand is needed
84: expected comma before second operand
85: registers stand-alone except in indirect
86: only two registers per operand
87: expected label or absolute
89: close parenthesis expected
90: cannot POP CS
91: cannot have xchg r8 with r16
92: segment registers not allowed
93: ESC external operand on left must be constant<64
94: only one of operands can have segment override
95: right operand must be a memory location

96: left operand must be a 16 bit register
97: left operand must be memory or register alone
98: operand cannot be a segment or immediate
99: count must be 1 or in CL
100: a byte constant operand is required
101: operand must use ( ) or be a label
102: LOCK followed by something illegal
103: REP precedes only string operations
104: not implemented
105: expected a label
106:
107: open parenthesis expected
108: expected register alone as right operand
109: segovpre then regalone, that's illegal
110: only one operand allowed
111: operands are AL,op2 for byte MUL, etc.
112: SP can only be used with the SS segment
113: MOVBIM only for immediate to memory
114: BIMs must be immediate bytes to memory
115: MOV immediate to Segment Register not allowed
116: Segment Register expected
117: (8087) invalid two-operand format
118: (8087) invalid single operand format
119: (8087) inproper operand field

120: (8087) instruction has no operands
121: no override of ES on string
     destination
122: intersegment jump or call needs 2
     constant or external operands
123: I/O port must be immediate byte
     or DX
124: I/O source-destination register
     must be AL or AX
125: prefix must be on same line as code
126: register expected as first token
     after '('

APPENDIX M
68000 SYNTAX ERRORS

1: undefined label
2: operand out of range
3: must have procedure name
4: number of parameters expected
5: extra symbols on source line
6: input line over 80 characters
7: unmatched conditional assembly directive
8: must be declared in .ASECT before used
9: identifier previously declared
10: improper format
11: illegal character in text
12: must .EQU before use if not to a label
13: macro identifier expected
14: code file too large
15: backwards .ORG not allowed
16: identifier expected
17: constant expected
18: invalid structure
19: extra special symbol
20: branch too far
21: LC-relative to externals not allowed
22: illegal macro parameter index
23: illegal macro parameter
24: operand not absolute
25: illegal use of special symbols
26: ill-formed expression
27: not enough operands
28: LC-relative to absolutes unrelocatable
29: constant overflow
30: illegal decimal constant
31: illegal octal constant
32: illegal binary constant
33: invalid key word
34: unmatched macro definition directive
35: include files may not be nested

36: unexpected end of input
37: .INCLUDE not allowed in macros
38: label expected
39: expected local label
40: local label stack overflow
41: string constants must be on single line
42: string constant exceeds 80 characters
43: cannot handle this relocate count
44: no local labels in .ASECT
45: expected key word
46: string expected
47: I/O - bad block, parity error (CRC)
48: I/O - illegal unit number
49: I/O - illegal operation on unit
50: I/O - undefined hardware error
51: I/O - unit no longer on-line
52: I/O - file no longer in directory
53: I/O - illegal file name
54: I/O - no room on disk
55: I/O - no such unit on-line
56: I/O - no such file on volume
57: I/O - duplicate file
58: I/O - attempted open of open file
59: I/O - attempted access of closed file
60: I/O - bad format in real or integer
61: I/O - ring buffer overflow
62: I/O - write to write-protected disk
63: I/O - illegal block number
64: I/O - illegal buffer address
65: nested macro definitions not allowed
66: '=' or '<>' expected
67: may not equate to undefined labels
68: .ABSOLUTE must appear before 1st proc
69: .PROC or .FUNC expected
70: too many procedures
71: only absolute expressions in .ASECT
72: must be label expression
73: no operands allowed in .ASECT
74: offset not word-aligned

75: LC not word-aligned

76: unrecognizable address mode

77: address register expected

78: close paren ')' expected

79: displacement out of range

80: index register expected

81: illegal length qualifier

82: illegal source address mode

83: illegal destination address mode

84: comma ',' expected

85: length qualifier required

86: length qualifier not allowed

87: data register expected

88: label expected

89: illegal register list

90: immediate operand expected

APPENDIX N
NIL POINTER VALUES

The following table lists the value designated as NIL for each processor. A NIL pointer (a pointer variable which is assigned the value NIL) is uninitialized or points to nothing.

Z80	0001
8080	0001
6502	0000
6809	0000
68000	0000
HP-87	0000
PDP-11	F001
9900	0000
8086	0000

INDEX