UCSD p-System™ Assembler

Software Library Manual

Texas Instruments Professional Computer

UCSD p-System Assembler
Part No. 2232402-0001
Original Issue: 15 April 1983

Preface

This manual describes the UCSD p-System™ 8086/88/87 Assembler. It also describes the instruction set of the 8086/88 CPUs and the 8087 floating point processor. (SofTech Microsystems developed this assembler to support these three Intel processors, but the 8087 is not necessarily supported on all 8086 based computers.) The p-System assembler is a powerful tool for creating assembly routines to be run inside or outside of the UCSD p-System environment.

To completely understand the 8086/88/87 assembly language, use the *Intel 8086 Family Users' Manual* with this manual. Refer also to the *MCS 86 Assembly Language Reference Manual* from Intel.

Chapter 1, The UCSD p-System Assembler, of this manual describes the UCSD p-System assembler. Note that the p-System assembler differs substantially from the Intel assembler.

Chapter 2, Overview of the CPU, gives a brief overview of the 8086/88 CPU; it covers the registers, flags, and addressing modes. For a more detailed description of the 8086/88 processor see the Intel manual.

Chapter 3, Operators, lists the 8086/88 and 8087 operations and gives a brief summary of their actions. Again, for more detailed information, refer to the Intel manual. Chapter 3 also describes assembler notational conventions and the differences between the standard Intel mnemonics and the mnemonics accepted by the UCSD p-System assembler.

Appendix A describes the linker. The linker combines separately assembled code files. It also can be used to link a high level host program with assembled routines.

Appendix B describes the Compress Utility. This utility allows you to produce relocatable or absolute assembled object code files so that it can run outside of the p-System environment.

Appendix C lists the 8086/88/87 errors.

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.

Contents

1

The UCSD p-System Assembler

INTRODUCTION

This chapter describes the UCSD p-System 8086/88/87 Assembler. It covers assembler-related concepts, assembler directives, and associated technical terms. Other topics covered here include:

- Linking assembled routines with host compilation units
- Assembled listings
- Error messages
- Sharing machine resources with the Interpreter

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

Use the UCSD p-System to develop assembly language programs to provide:

- Assembly language procedures to run under control of a host program
- Stand-alone assembly language programs to use outside of the operating system's environment

The UCSD p-System 8086/88/87 Assembler, in conjunction with the system linker and some support programs, has been designed to meet these needs.

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 sixteen bits or two 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 is not significant. Only the first eight characters of the label are significant. This definition of identifiers is equivalent to the Pascal definition.

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. Do not 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 cannot 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 this is 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 −32,768 to 32,767 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 477.

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

Default Radix Integer Constants

The radix of an integer constant lacking a trailing radix character is decimal on the p-System 8086/87 assembler.

Character Constants

Character constants are special cases of character strings; you can 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 single 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 following section on Data and Constant Definitions 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. Do not use subexpressions that evaluate to relocatable quantities as arguments to a multiplication, division, or logical operation. Also, do not apply unary operators to relocatable quantities.

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

Linking and Restrictions

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

An expression can 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, do not place an externally defined label in an expression containing a forward-referenced label.

Arithmetic and Logical Operators

You can use the following operators in expressions:

- Unary operations—
  + plus
  − minus (two's complement negation)
  ~ logical not (one's complement negation)

- 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 the normal parentheses to group expressions. Using parentheses to group expressions does not generate an error but causes the assembler to interpret the expression as an 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 can 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 is not used when referencing the label. As in Pascal, only the first eight characters of the label are significant; the assembler ignores the rest. Also, as in Pascal, the underscore character is not 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. Do not 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 can 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 which contains its own relocation information. During linking, a Pascal host program may refer to each routine 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.

Global Declarations

An assembly routine cannot directly access objects declared in another assembly routine, even if the routines are assembled in the same source file; however, sometimes it is 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 or .FUNC 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 non-code-generating directives (.EQU, .REF, .DEF, .MACRO, .LIST, and so on).

Absolute Sections

You will 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 gives 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 cannot 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 can 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 0DF00H             ; 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 the user.
- 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 earlier 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 preceding paragraph entitled Character Strings.)
file identifier	Any legal name for a Pascal text file.

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

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

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 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 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
```

The output code would be:

```
              0000
              0002
              0000    ; this is a default value.
              0004

L1    .WORD     L2
```

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

.EQU Equates a value to a label. 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 do not generate code, and their source lines do not 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 can 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

.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 is not 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 is not 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 following section called Macro Language, gives a detailed description of macro language.

Form: [b] .MACROLIST

Example: .MACROLIST

.NOMACROLIST — Specifies that all subsequent macro definitions will not have their macro bodies printed when they are called in the source program. Only the identified macro 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. The following section, 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 Pascal 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 Pascal 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 interpreter. .INTERP is a predefined symbol for a processor-dependent location in the resident interpreter code; you may use offsets from this base location to access any code in the interpreter. To use this feature correctly, you must know the interpreter'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 BOMBINT
```

.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 the section, 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 which is 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 cannot use .INCLUDE in an included source file (that is, nested use of the directive) and 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. Do not use .ABSOLUTE when creating a Pascal external procedure. (Refer to the Program Linking and Relocation section, presented later in this chapter, for a detailed description of absolute 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 Absolute Sections paragraph 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 paragraph 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 decimal with the 8086 Assembler.
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 can contain the .ELSE directive. Use conditional expressions to control inclusion of conditional sections. Conditional sections can 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 is 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 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 sets 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
t3 }; 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 can 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 should not include a macro definition within another macro definition. However, you can include macro calls. You can 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 can be forward referenced within the bodies of other macro definitions.

Macro Calls

You can place macro calls anywhere in a source program that code can 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 can 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 *not* expanded within the quotes of an ASCII data string.

Three cases are possible:

- The parameter exists — the substitution is made.
- The n'th parameter does not exist in the parameter list being checked (less than n parameters were passed); a null string is substituted.
- Another symbol in the form of *%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     DEUX
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 it 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 that 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 are not passed as uninterpreted text. The scope of a local label passed in a macro call does not change as it is passed through increasing levels of macro nesting, regardless of naming conflicts along the way. One use of this property is the passing of 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 AND RELOCATION

The adaptable 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 that 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.

- Pascal 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 Pascal system environment must always be relocatable; the system interpreter performs loading and relocation at a load address that the state of the system determines.

Absolute code will not 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 does not have to remain present throughout the life of the program.

The directives .PROC and .FUNC designate statically relocatable routines; .RELPROC and .RELFUNC designate dynamically relocatable routines. Code segments that contain dynamically relocatable code do not 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 can be swapped in and out of main memory while the host program (or unit) is running. Thus, dynamically relocatable code *cannot* maintain internal data spaces; data that is meant to last across different calls of the assembly routine must be kept in your host data segments by using .PRIVATEs and .PUBLICs. (You must make sure that this is the case.)

- Data space is embedded in the code, but the code does not move:

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

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

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

- **Wrong:** The code moves, and the data is embedded in the code, so the data may be destroyed.

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

Code pool management is described in the *UCSD p-System Internal Architecture*, TI part number 2232400-0001.

Program Linking Directives

This section describes the overall use of linking directives. All linking of assembly procedures involves word quantities; it is not 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 interpreter 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 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 multiword variables in host programs; it is not 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 cannot access. The optional length attribute of the arguments allows multiword 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 to the paragraph in this chapter entitled 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 cannot .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 .DEFed 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 Pascal host program; they are not 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 Pascal host's external procedure declaration matches the number specified by the assembly procedure declaration. It is not 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. You can simplify linking by adding the assembled routine to the system library with the librarian program.

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 following paragraph entitled 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. A potentially fatal system crash can occur if you do not do this, since 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 to the section entitled 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 your Texas Instruments Professional Computer has been set to run on. 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 (two or four for type real; one, otherwise) containing the function result onto the stack before passing control back to the host. The 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 Pascal data structures; however, with this freedom comes the responsibility of respecting the integrity of the Pascal

run-time environment. To give you a better understanding of the Pascal/assembly language interface, this section enumerates the p-machine's parameter passing conventions for all data types; it does not actually describe data representations.

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 can 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 are not 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 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 the next section).

Pascal procedures protect the original variables by using the passed pointer to copy their values into a local data space for processing. You should respect this convention and not alter the contents of the original variables.

Accessing Byte Array Parameters with a Segment Pointer

A segment pointer consists of two words on the stack. The first word (tos) contains either NIL or a pointer to a segment environment record.

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

If the first word is not NIL, then to find the parameter it is necessary to chain through some records. The first word is a pointer and the second word is an offset. The first word points to a segment environment record. The second word of this record contains a pointer to a pointer which points to the base of the segment where the parameter resides. The exact location of the parameter is given by the second word on the stack (tos-1), which is an offset into the code segment.

This address chain may be described as follows (offsets are word offsets):

(first_word+1)^^+<contents of second_word>

For a full description of these mechanisms, refer to the *UCSD p-System Internal Architecture.*

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 occur after the end of the machine instructions or 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 Pascal

.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         RETADR      ; return addr pushed on
                        ;    stack

; does nothing

PUSH        RETADR      ; set up stack for
                        ;    return
RETL
                        ; data area
RETADR      .EQU    TEMP1
TEMP1       .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

; 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 UCSD p-System 8086/88/87 Assembler can produce absolute (core image) code files for use outside of the p-System's run-time environment.

The p-System does not include a linking loader or an assembly language debugger, as the p-machine architecture is not 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 in this chapter, to the paragraph entitled Executing Absolute Code Files.)

Use the utility Compress for a much easier and more versatile way of doing this task. It allows you to relocate and compact code. Refer to the *UCSD p-System Program Development*, TI part number 2232399-0001.

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 do not produce continuous object code and cannot 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 is 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 UCSD p-System to load and execute absolute code files. The program outlined is not 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 user 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 is not overwritten while

being moved to its final location. You can detect this possibility before proceeding with loading.

Note that in most cases, loading object code into arbitrary memory locations while a Pascal 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 Pascal system.

The loader program consists of:

- A Pascal host program that calls two external procedures
- One or more linkable absolute code files to be loaded (.RELPROCs are not allowed)
- A small assembly procedure, MOVE_AND_GO, that moves the above object code files from their system load addresses to their proper locations and then transfers control to them
- A small assembly language procedure, LOAD_ADDRESS, that returns the system load addresses of the aforementioned 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 Pascal host performs the following actions:

1. Prints 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.

   Issues 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 is not 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 are not in use are usually saved with the file name *ASM8086.CODE*.

Support Files

The UCSD p-System 8086/88/87 Assembler has three associated support files: two opcodes files and an error file. Always store these along with the assembler code file.

In order for the assembler to run correctly, the proper opcode files must be present on some on-line disk. The assembler will search all units in increasing order of the unit number until it finds them. The opcode files must have the names *8086.OPCODES* and *8087.FOPS*. The *8087.FOPS* file is neccessary only if you use 8087 instructions in your procedures. The opcode files contain all predefined symbols (instruction and register names) and their corresponding values for the associated assembly language. If the *8086 .OPCODES* file is not on-line, the assembler writes *<opfilename> not on any vol* and aborts the assembly. (If *8086 .FOPS* is not present it will not abort the assembly.)

The assembler also has an error file that contains a list of 8086/88/87 specific error messages. The error file must have the name *8086.ERRORS*. 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 the listing prompt and the generated code file will be named *SYSTEM.WRK.CODE*. If not, the following prompt appears:

```
Assemble what text?
```

Enter the file name of the input file; then press the RETURN key. Pressing only the RETURN key aborts the assembly; otherwise, the following prompt appears:

```
To what codefile?
```

Enter the desired name of the output code file, followed by pressing the RETURN key.

Pressing only the RETURN key 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
Output file for assembled listing: CR for none
```

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

- Press the ESC key followed by the RETURN key to abort the assembly and return to the command menu
- *CONSOLE:* or *#1:*; this sends an assembled listing of the source program to the screen during assembly
- *PRINTER:* or *#6:*; this sends an assembled listing to the printer unit
- *REMOUT:* or *#8:*; this sends an assembled listing to the REMOTE unit

- A carriage RETURN; this causes the assembler to suppress generation of an assembled listing and ignore all listing directives.

- 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

```
#5:listing.text
#5:listing
```

In all cases, it is 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 display unit 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 display unit for your benefit. One dot is written to the display unit for every line assembled; on every 50th line, the number of lines currently assembled is displayed on the left side of the display unit (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 does not 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 cannot be opened, the file does not exist or there is not enough memory, no message appears. This is followed by the menu:

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

Pressing E calls the editor, pressing the space bar continues the assembly, and pressing the ESC key aborts the assembly. The following restrictions exist when you call the editor or attempt to continue:

- In most cases, pressing the space bar restarts the assembly process with no problems; since assembly language source statements are independent of one another with respect to syntax, it is 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 the space bar 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.

- If you press E, the system automatically calls the editor, which opens the file containing the offending error and positions the cursor at the location where the error occurred as long as you are assembling the work file; otherwise the editor will prompt for a file name. This feature always works correctly when the source program is wholly contained in one file. However, when you use include files, set up the input and output files manually, so the editor can position the cursor in the file that contains the error. (Refer back to the paragraph, Setting Up Input and Output Files.)

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 will not 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 is 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 is 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 is 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 *UCSD p-System Internal Architecture.*

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, except for source lines that contain list directives, is included in the assembled listing. 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 are not listed in macro expansions.

Source lines with conditional assembly directives are listed; however, source statements in an unassembled part of a conditional section are not listed.

Error Messages

Error messages in assembled listings have the same format as the error messages sent to the console, except that the user menu is not included. (Refer to the preceding 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 in this chapter to the following section, Sharing PME Resources.) 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 cannot 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. Each 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.

SHARING PME RESOURCES

Calling and Returning

The p-machine emulator (PME) calls an assembly routine using the call long (CALLL) operator (*long* refers to *intersegment*). 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 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 is 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 do not, the results cannot 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. The following are examples.

```
MOV BX, (BP+4)    ;     pick up parameter(pointer)
MOV AX, SS:(BX)   ;     pick up VAR parameter value
```

.PRIVATE and .PUBLIC variables are also SS relative.

.BYTE quantities, .WORD quantities, and .REFed labels are relative to CS, DS, or ES.

2

Overview of the CPU

INTRODUCTION

This chapter briefly describes the registers, flags, and addressing modes of the 8086/88 CPU. For more detailed information concerning the 8086/88 processor see the *Intel 8086 Family User's Manual.*

GENERAL REGISTERS

The 8086/88 CPU contains eight 16-bit general registers. The general registers are subdivided into two sets of four registers each: the data registers, sometimes called the H and L group for *high* and *low*; and the pointer and index registers, sometimes called the P and I group.

The data registers are unique because their upper (high) and lower halves are separately addressable. This means that you can use each data register interchangeably as a 16-bit register or as two 8-bit registers. The other CPU registers always are accessed as 16-bit units only. You can use the data registers without constraint in most arithmetic and logic operations. In addition, some instructions use certain registers implicitly; thus allowing compact, yet powerful, encoding.

The pointer and index registers can also be used in most arithmetic and logic operations. Except for BP, the P and I registers are also used implicitly in some instructions.

Data Register Group

Accumulator:	AX	(16 Bits)
	AH	(Bits 8-15)
	AL	(Bits 0-7)
Base:	BX	(16 Bits)
	BH	(Bits 8-15)
	BL	(Bits 0-7)
Count:	CX	(16 Bits)
	CH	(Bits 8-15)
	CL	(Bits 0-7)
Data:	DX	(16 Bits)
	DH	(Bits 8-15)
	DL	(Bits 0-7)

Pointer and Index Register Group Registers

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Destination Index

Implicit Use of General Registers

AX	Word Multiply Word Divide Word I/O
AL	Byte Multiply Byte Divide Byte I/O Translate Decimal Arithmetic
AH	Byte Multiply Byte Divide
BX	Translate
CX	String Operations Loops
CL	Variable Shift and Rotate
DX	Word Multiply Word Divide Indirect I/O
SP	Stack Operations
SI	String Operations
DI	String Operations

SEGMENT REGISTERS

The megabyte of memory that the 8086/88 can address is divided into logical segments of up to 64K bytes each. (Memory segmentation is described in detail in the *Intel 8086 Family User's Manual.*) The CPU has access to four segments at a time. Their base addresses (starting locations) are contained in the segment registers. The following table lists the segment registers:

Segment Registers

CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Extra Segment

The CS register points to the current code segment; instructions are fetched from this segment. The SS register points to the current stack segment; stack operations are performed on locations in this segment. The DS register points to the current data segment; it generally contains program variables. The ES register points to the current extra segment, which also is typically used for data storage.

Programs can access the segment registers and several instructions can manipulate them. See the *Intel 8086 Family User's Manual* for suggested guidelines for using segment registers.

FLAGS

The 8086/88 has six 1-bit status flags that reflect certain properties of the result of an arithmetic or logic operation. A group of instructions is available that allows a program to alter its execution depending on the state of these flags; that is, depending upon the result of a prior operation. Different instructions affect the status flags differently; in general, however, the flags reflect the following conditions:

If AF (the auxiliary carry flag) is set, there has been a carry out of the low nibble (4-bit) into the high nibble, or a borrow from the high nibble into the low nibble of an 8-bit quantity. This flag is used by decimal arithmetic instructions.

If CF (the carry flag) is set, there has been a carry out of, or a borrow into, the high-order bit of the result (8- or 16-bit). The flag is used by instructions that add and subtract multibyte numbers. Rotate instructions can also isolate a bit in memory or a register by placing it in the carry flag.

If OF (the overflow flag) is set, an arithmetic overflow has occurred; that is, a significant digit has been lost because the size of the result exceeded the capacity of its destination location. An Interrupt On Overflow instruction is available that generates an interrupt in this situation.

If SF (the sign flag) is set, the high-order bit of the result is a 1. Since negative binary numbers are represented in the 8086/88 in standard two's complement notation, SF indicates the sign of the result (0 = positive, 1 = negative).

If PF (the parity flag) is set, the result has even parity, an even number of 1 bits. This flag can be used to check for data transmission errors.

If ZF (the zero flag) is set, the result of the operation is 0.

To alter processor operations, programs can set and clear three additional control flags:

Setting DF (the direction flag) causes string instructions to auto-decrement; that is, to process strings from high addresses to low addresses, or from *right to left.* Clearing DF causes string instructions to auto-increment, or to process strings from *left to right.*

Setting IF (the interrupt-enable flag) allows the CPU to recognize external (maskable) interrupt requests. Clearing IF disables these interrupts. IF does not affect either nonmaskable external or internally generated interrupts.

Setting TF (the trap flag) puts the processor into single-step mode for debugging. In this mode, the CPU automatically generates an internal interrupt after each instruction, allowing a program to be inspected as it executes instruction by instruction. The *Intel 8086 Family User's Manual* contains an example showing the use of TF in a single-step and breakpoint routine.

The following is a summary of the flags:

CF	Carry
PF	Parity
AF	Auxiliary Carry
ZF	Zero
SF	Sign
OF	Overflow
IF	Interrupt-Enable
DF	Direction
TF	Trap

ADDRESSING MODES

The 8086/88 provides many different ways to access instruction operands. Operands may be contained in registers, within the instruction itself, in memory, or in I/O ports. In addition, the addresses of memory can be calculated in several different ways. This section briefly describes these addressing modes. For a more complete description, see the *Intel 8086 Family User's Manual.*

Register and Immediate Operands

Instructions that specify only register operands are generally the most compact and fastest executing of all instruction forms. This is because the register *addresses* are encoded in instructions in just a few bits, and because these operations are performed entirely within the CPU (no bus cycles are run). Registers may serve as source operands, destination operands, or both.

Immediate operands are constant data contained in an instruction. The data may be either 8 or 16 bits long. Immediate operands can be accessed quickly because they are available directly from the instruction queue; like a register operand, no bus cycles need be run to obtain an immediate operand. Immediate operands are limited because they may only serve as source operands and they are constant values.

Direct Addressing

Direct addressing is the simplest memory addressing mode; it involves no registers. The effective address is taken directly from the displacement field of the instruction. (The effective address is the unsigned 16-bit number that expresses the operand's distance in bytes from the beginning of the segment in which it resides.) The default segment is the current data segment. Direct addressing is typically used to access simple variables (scalars).

Register Indirect Addressing

The effective address may be taken directly from one of the base or index registers (BX, BP, SI, or DI). One instruction can operate on many different memory locations if the value in the base or index register is updated appropriately. The Load Effective Address (LEA) and arithmetic instructions can be used to change the register value.

You can use *any* 16-bit general register for register indirect addressing with the JMP or CALL instructions.

Based Addressing

In based addressing, the effective address is the sum of a displacement value and the content of register BX or register BP. Specifying BX as a base register directs the Bus Interface Unit to obtain the operand from the current data segment (DS), unless a segment override prefix is present. Specifying BP as a base register directs the Bus Interface Unit (see the Intel manual) to obtain the operand from the current stack segment (SS), unless a segment override prefix is present. This makes based addressing with BP a very convenient way to access stack data (the Intel manual contains examples of this).

Based addressing also provides a straightforward way to address structures that may be located at different places in memory. You can point a base register to the base of a structure and address elements of the structure by their displacements from the base. By simply changing the base register, you can access a different structure.

Based Indexed Addressing

Based indexed addressing generates an effective address that is the sum of a base register, an index register, and a displacement. Based indexed addressing is a very flexible mode because two address components can be varied at execution time.

Based indexed addressing provides a convenient way for a procedure to address an array allocated on a stack. Register BP can contain the offset of a reference point on the stack, typically the top of the stack after the procedure has saved registers and allocated local storage. Using a displacement value, you can express the offset of the beginning of the array from the reference point, and you can use an index register to access individual array elements.

Based indexed addressing can also access arrays contained in structures and matrices (two-dimensional arrays).

String Addressing

String instructions do not use the normal memory addressing modes to access their operands. Instead, the index registers are used implicitly. When a string instruction is executed, SI is assumed to point to the first byte or word of the source string in the current data segment (DS), and DI is assumed to point to the first byte or word of the destination string in the current extra segment (ES). In a repeated string operation, SI and DI are automatically adjusted up or down according to the direction flag (DF) to obtain subsequent bytes or words.

3

Operators

3

Operators

INTRODUCTION

This chapter describes how the UCSD p-System 8086/88/87 Assembler notational conventions differ from the Intel standard assembler.

Also, each of the 8086/88 and 8087 operators is briefly described. These descriptions are intended for quick reference use only. For detailed information concerning the instruction set, see the *Intel 8086 Family User's Manual.*

SYNTAX CONVENTIONS

The UCSD 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 or operators are implemented. Instead, the assembler directives described in Chapter 1 of this manual are available.

Parentheses — Enclose index or base register references in a memory operand in parentheses, not square brackets; for example, FIRST(BX) rather than FIRST[BX].

Angle Brackets — Group expressions within angle brackets (< >), not in 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, and SARMB are added to the vocabulary to specify memory byte operands.

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

```
MUL     memop
```

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

```
MUL     AL,memop
```

The same holds true for IMUL, DIV, and 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 (four versus eight).

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 is not 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 are not.

Segment Override — XLAT and the string instructions (nine) 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

> Only the default segment for SI, namely DS, can be overridden. The segment for DI is ES and cannot be overridden. A segment override prefix of DS applied to SI does not generate a segment override prefix.

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

```
XLAT          AL,(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 on.

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.

THE 8086/88 INSTRUCTION SET

The following are the 8086/88 opcode mnemonics recognized by the UCSD p-System 8086/88/87 Assembler. The differences between these mnemonics and the standard Intel mnemonics are discussed in the beginning of this chapter. This is meant as a quick reference list only. For a detailed description of the 8086/88 operations, see the *Intel 8086 Family Users' Manual.*

NOTE

The special case mnemonics (which are not Intel standard), such as ADDBIM, are listed with the standard mnenonic to which they correspond; for example, ADD. This does not mean that the special case mnemonics indicate operations that take all of the addressing modes listed. For example, ADDBIM is meant for adding immediate bytes only. The mnemonic ADD is meant to take any of the other addressing modes listed and defaults to a word add if an immediate quantity is indicated.

The following list of 8086/88 code mnemonics includes a section on flags which informs you of the effect each operation has on the flags (discussed in Chapter 2). The following list explains the codes used below.

U	Flag is undefined after this operation
X	Flag is affected after this operation
1	Flag is set after this operation
0	Flag is cleared after this operation
blank	Flag is unaffected

AAA (ASCII Adjust for Addition)

Form:	AAA (no operands)
Flags:	O D I T S Z A P C U U U X U X
Operands:	None
Coding Example:	AAA

AAD (ASCII Adjust for Division)

Form:	AAD (no operands)
Flags:	O D I T S Z A P C U X X U X U
Operands:	None
Coding Example:	AAD

AAM (ASCII Adjust for Multiply)

Form:	AAM (no operands)
Flags:	O D I T S Z A P C U X X U X U
Operands:	None
Coding Example:	AAM

AAS (ASCII Adjust for Subtraction)

Form:	AAA (no operands)
Flags:	O D I T S Z A P C U U U X U X
Operands:	None
Coding Example:	AAS

ADC (Add with Carry)
ADCBIM (Add with Carry, Immediate Byte)
ADCM (Add with Carry, Direct Addressing Mode)

Form:	ADC destination, source
Flags:	O D I T S Z A P C X X X X X X
Operands:	register, register register, memory memory, register register, immediate memory, immediate accumulator, immediate
Coding Example:	ADC AX,SI

ADD (Addition)
ADDBIM (Add Immediate Byte)
ADDM (Add Direct Addressing Mode)

Form:	ADD destination, source
Flags:	O D I T S Z A P C X X X X X X
Operands:	register, register register, memory memory, register register, immediate memory, immediate accumulator, immediate
Coding Example:	ADD DI,(BX).ALPHA

AND (Logical AND)
ANDBIM (Logical AND, Immediate Byte)
ANDM (Logical AND, Direct Addressing Mode)

Form:	AND destination, source
Flags:	O D I T S Z A P C 0 X X U X 0
Operands:	register, register register, memory memory, register register, immediate memory, immediate accumulator, immediate
Coding Example:	AND CX,FLAG_WORD

CALL (Call a Procedure)
CALLL (Long Call of a Procedure)

Form:	CALL target
Flags:	O D I T S Z A P C
Operands:	near-proc far-proc memptr 16 regptr 16 memptr 32
Coding Example:	CALL NEAR_PROC

CBW (Convert Byte to Word)

Form:	CBW (no operands)
Flags:	O D I T S Z A P C
Operands:	None
Coding Example:	CBW

CLC (Clear Carry Flag)

Form:	CLC (no operands)
Flags:	O D I T S Z A P C 0
Operands:	None
Coding Example:	CLC

CLD (Clear Direction Flag)

Form:	CLD (no operands)
Flags:	O D I T S Z A P C 0
Operands:	None
Coding Example:	CLD

CLI (Clear Interrupt Flag)

Form:	CLI (no operands)
Flags:	O D I T S Z A P C 0
Operands:	None
Coding Example:	CLI

CMC (Complement Carry Flag)

Form:	CMC (no operands)
Flags:	O D I T S Z A P C X
Operands:	None
Coding Example:	CMC

CMP (Compare Destination to Source)
CMPBIM (Compare Immediate Byte)
CMPM (Compare Direct Addressing Mode)

Form:	CMP destination, source
Flags:	O D I T S Z A P C X X X X X X
Operands:	register, register register, memory memory, register register, memory memory, immediate accumulator, immediate
Coding Example:	CMP (BP+2),SI

CMPSW (Compare String, Wordwise)
CMPSB (Compare String, Bytewise)

Form:	CMPSB dest-string, source-string
Flags:	O D I T S Z A P C X X X X X X
Operands:	dest-string, source-string (repeat) dest-string, source-string
Coding Example:	COMPSB BUFF1, BUFF2

CWD (Convert Word to Double Word)

Form:	CWD (no operands)
Flags:	O D I T S Z A P C
Operands:	None
Coding Example:	CWD

DAA (Decimal Adjust for Addition)

Form:	DAA (no operands)
Flags:	O D I T S Z A P C X X X X X X
Operands:	None
Coding Example:	DAA

DAS (Decimal Adjust for Subtraction)

Form:	DAS (no operands)
Flags:	O D I T S Z A P C U X X X X X
Operands:	None
Coding Example:	DAS

DEC (Decrement by One)
DECMB (Decrement Memory Byte)

Form:	DEC destination
Flags:	O D I T S Z A P C X X X X X
Operands:	reg16 reg8 memory
Coding Example:	DEC AX

DIV (Division, Unsigned)
DIVMB (Division, Unsigned, Memory Byte)

Form:	DIV source
Flags:	O D I T S Z A P C U U U U U U
Operands:	reg8 reg16 mem8 mem16
Coding Example:	DIV TABLE(SI)

ESC (Escape)

Form:	ESC external-opcode, source
Flags:	O D I T S Z A P C
Operands:	immediate, memory immediate, register
Coding Example:	ESC 20, AL

HLT (Halt)

Form:	HLT (no operands)
Flags:	O D I T S Z A P C
Operands:	None
Coding Example:	HLT

IDIV (Integer Division)
IDIVMB (Integer Division, Memory Byte)

Form:	IDIV source
Flags:	O D I T S Z A P C U U U U U U U
Operands:	reg8 reg16 mem8 mem16
Coding Example:	IDIV (BX).DIVISOR_WORD

IMUL (Integer Multiplication)
IMULMB (Integer Multiplication Memory Byte)

Form:	IMUL source
Flags:	O D I T S Z A P C X U U U U X
Operands:	reg8 reg16 mem8 mem16
Coding Example:	IMUL CL

IN (Input Byte or Word)
INB (Input Byte)
INW (Input Word)

Form:	IN accumulator, port
Flags:	O D I T S Z A P C
Operands:	accumulator, immed8 accumulator, DX
Coding Example:	IN AX, DX

INC (Increment by One)
INCMB (Increment Memory Byte)

Form:	INC destination
Flags:	O D I T S Z A P C X X X X X X
Operands:	reg16 reg8 memory
Coding Example:	INC CX

INT (Interrupt)

Form:	INT interrupt-type
Flags:	O D I T S Z A P C 0 0
Operands:	immed8 (type=3) immed8 (type < > 3)
Coding Example:	INT 3

INTR (External Maskable Interrupt)

Form:	Interrupt if INTR and IF=1
Flags:	O D I T S Z A P C 0 0
Operands:	None
Coding Example:	Not applicable

INTO (Interrupt if Overflow)

Form:	INTO (no operands)
Flags:	O D I T S Z A P C 0 0
Operands:	None
Coding Example:	INTO

IRET (Interrupt Return)

Form:	IRET (no operands)
Flags:	O D I T S Z A P C R R R R R R R R R
Operands:	None
Coding Example:	IRET

JA/JNBE (Jump if Above/Jump if not Below nor Equal)

Form:	JA short-label JNBE short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JA ABOVE

JAE/JNB (Jump if Above or Equal/Jump if not Below)

Form:	JAE short-label JNB short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JAE ABOVE_EQUAL

JB/JNAE (Jump if Below/Jump if not Above nor Equal)

Form:	JB short-label JNB short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JB BELOW

JBE/JNA (Jump if Below or Equal/Jump if not Above)

Form:	JBE short-label JNA short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JNA NOT_ABOVE

JC (Jump if Carry)

Form:	JC short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JC CARRY_SET

JCXZ (Jump if CX is Zero)

Form:	JCXZ short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JCXZ COUNT_DONE

JE/JZ (Jump if Equal/Jump if Zero)

Form:	JE short-label JZ short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JZ ZERO

JG/JNLE (Jump if Greater/Jump if not Less nor Equal)

Form:	JG short-label JNLE short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JG GREATER

JGE/JNL (Jump if Greater or Equal/Jump if not Less)

Form:	JGE short-label JNL short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JGE GREATER_EQUAL

JL/JNGE (Jump if Less/Jump if not Greater nor Equal)

Form:	JL short-label JNGE short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JL LESS

JLE/JNG (Jump if Less or Equal/Jump if not Greater)

Form:	JLE short-label JNG short-label
Flags:	O D I T S Z A P C
Operands:	short-label

JMP (Jump)
JMPL (Jump Long)

Form:	JMP target
Flags:	O D I T S Z A P C
Operands:	short-label near-label far-label memptr16 regptr16 memptr32
Coding Example:	JMP NEAR_LABEL

JNC (Jump if not Carry)

Form:	JNC short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JNC NOT_CARRY

JNE/JNZ (Jump if not Equal/Jump if not Zero)

Form:	JNE short-label JNZ short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JNE NOT_EQUAL

JNO (Jump if not Overflow)

Form:	JNO short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JNO NO_OVERFLOW

JNP/JPO (Jump if not Parity/Jump if Parity Odd)

Form:	JNP short-label JPO short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JPO ODD_PARITY

JNS (Jump if not Sign)

Form:	JNS short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JNS POSITIVE

JO (Jump if Overflow)

Form:	JO short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JO SIGNED_OVERFLOW

JP/JPE (Jump if Parity/Jump if Parity Even)

Form:	JP short-label JPE short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JPE EVEN_PARITY

JS (Jump if Sign)

Form:	JS short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	JS NEGATIVE

LAHF (Load AH from Flags)

Form:	LAHF (no operands)
Flags:	O D I T S Z A P C
Operands:	None
Coding Example:	LAHF

LDS (Load Pointer Using DS)

Form:	LDS destination, source
Flags:	O D I T S Z A P C
Operands:	reg16, memptr32
Coding Example:	LDS SI,DATA.SEG(DI)

LEA (Load Effective Address)

Form:	LEA destination, source
Flags:	O D I T S Z A P C
Operands:	reg16, memptr16
Coding Example:	LEA BX,(BP)(DI)

LES (Load Pointer Using ES)

Form:	LES destination, source
Flags:	O D I T S Z A P C
Operands:	reg16, memptr32
Coding Example:	LES DI,(BX).TEXT_BUFF)

LOCK (Lock Bus)

Form:	LOCK (no operands)
Flags:	O D I T S Z A P C
Operands:	None
Coding Example:	LOCK XCHGFLAG,AL

LODSB (Load String Bytewise)
LODSW (Load String Wordwise)

Form:	LODS source-string
Flags:	O D I T S Z A P C
Operands:	source-string (repeat) source-string
Coding Example:	REP LODS NAME

LOOP (Loop)

Form:	LOOP short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	LOOP AGAIN

LOOPE/LOOPZ (Loop if Equal/Loop if Zero)

Form:	LOOPE short-label LOOPZ short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	LOOPE AGAIN

LOOPNE/LOOPNZ (Loop if not Equal/Loop if not Zero)

Form:	LOOPNE short-label LOOPNZ short-label
Flags:	O D I T S Z A P C
Operands:	short-label
Coding Example:	LOOPNE AGAIN

NMI (External Nonmaskable Interrupt)

Form:	Interrupt if NMI=1
Flags:	O D I T S Z A P C 0
Operands:	None
Coding Example:	Not applicable

MOV (Move)
MOVBIM (Move Immediate Byte)
MOVM (Move Direct Addressing Mode)

Form:	MOV destination, source
Flags:	O D I T S Z A P C
Operands:	memory, accumulator accumulator, memory register, register register, memory memory, register register, immediate memory, immediate seg-reg, reg16 seg-reg, mem16 reg16, seg-reg memory, seg-reg
Coding Example:	MOV BP,STACK_TOP

MOVSB (Move String Bytewise)
MOVSW (Move String Wordwise)

Form:	MOVS dest-string, source-string
Flags:	O D I T S Z A P C
Operands:	dest-string, source-string (repeat) dest-string, source-string
Coding Example:	MOVS LINE, EDIT_DATA

MOVSB/MOVSW (Move String (Byte/Word))

Form:	MOVSB/MOVSW (no operands)
Flags:	O D I T S Z A P C
Operands:	None
Coding Example:	REP MOVSW

MUL (Multiplication, Unsigned)
MULMB (Multiplication, Unsigned, Memory Byte)

Form:	MUL source
Flags:	O D I T S Z A P C X U U U U X
Operands:	reg8 reg16 mem8 mem16
Coding Example:	MUL CX

NEG (Negate)
NEGMB (Negate Memory Byte)

Form:	NEG destination
Flags:	O D I T S Z A P C X X X X X 1*
Operands:	register memory
Coding Example:	NEG AL

NOP (No Operation)

Form:	NOP
Flags:	O D I T S Z A P C
Operands:	None
Coding Example:	NOP

* 0 if destination=0.

NOT (Logical NOT)
NOTMB (Logical NOT, Memory Byte)

Form:	NOT destination
Flags:	O D I T S Z A P C
Operands:	register memory
Coding Example:	NOT AX

OR (Logical Inclusive OR)
ORBIM (Logical Inclusive OR, Immediate Byte)
ORM (Logical Inclusive OR, Direct Addressing Mode)

Form:	OR destination, source
Flags:	O D I T S Z A P C 0 X X U X 0
Operands:	register, register register, memory memory, register accumulator, immediate register, immediate memory, immediate
Coding Example:	OR FLAG_BYTE, CL

OUT (Output Byte or Word)
OUTB (Output Byte)
OUTW (Output Word)

Form:	OUT port, accumulator
Flags:	O D I T S Z A P C
Operands:	immed8, accumulator DX, accumulator
Coding Example:	OUT DX, AL

POP (Pop Word off Stack)

Form:	POP destination
Flags:	O D I T S Z A P C
Operands:	register seg-reg (CS illegal) memory
Coding Example:	POP DX

POPF (Pop Flags off Stack)

Form:	POPF (no operands)
Flags:	O D I T S Z A P C R R R R R R R R R
Operands:	None
Coding Example:	POPF

PUSH (Push Word Onto Stack)

Form:	PUSH source
Flags:	O D I T S Z A P C
Operands:	register seg-reg (CS legal) memory
Coding Example:	PUSH ES

PUSHF (Push Flags Onto Stack)

Form:	PUSHF (no operands)
Flags:	O D I T S Z A P C
Operands:	None
Coding Example:	PUSHF

RCL (Rotate Left Through Carry)
RCLMB (Rotate Left Through Carry, Memory Byte)

Form:	RCL destination, count
Flags:	O D I T S Z A P C X X
Operands:	register, 1 register, CL memory, 1 memory, CL
Coding Example:	RCL AL, CL

RCR (Rotate Right Through Carry)
RCRMB (Rotate Right Through Carry, Memory Byte)

Form:	RCR destination, count
Flags:	O D I T S Z A P C X X
Operands:	register, 1 register, CL memory, 1 memory, CL
Coding Example:	RCR (BX).STATUS, 1

REP (Repeat String Operation)

Form:	REP (no operands)
Flags:	O D I T S Z A P C
Operands:	None
Coding Example:	REP MOVS DEST,SRCE

REPE/REPZ (Repeat String Operation While Equal/While Zero)

Form:	REPE/REPZ (no operands)
Flags:	O D I T S Z A P C
Operands:	None
Coding Example:	REPE CMPS DATA,KEY

REPNE/REPNZ (Repeat String Operation While not Equal/not Zero)

Form:	REPNE/REPNZ (no operands)
Flags:	O D I T S Z A P C
Operands:	None
Coding Example:	REPNE SCASW INPUT_LINE

RET (Return from Procedure)
RETL (Return Long from Procedure)

Form:	RET optional pop value
Flags:	O D I T S Z A P C
Operands:	(intra-segment, no pop) (intra-segment, pop) (inter-segment, no pop) (inter-segment, pop)
Coding Example:	RET 4

ROL (Rotate Left)
ROLMB (Rotate Left, Memory Byte)

Form:	ROL destination, count
Flags:	O D I T S Z A P C X X
Operands:	register, 1 register, CL memory, 1 memory, CL
Coding Example:	ROL BX,1

ROR (Rotate Right)
RORMB (Rotate Right, Memory Byte)

Form:	ROR destination, count
Flags:	O D I T S Z A P C X X
Operands:	register, 1 register, CL memory, 1 memory, CL
Coding Example:	ROR CMD_WORD, CL

SAHF (Store AH Into Flags)

Form:	SAHF (no operands)
Flags:	O D I T S Z A P C R R R R R
Operands:	None
Coding Example:	SAHF

SAL/SHL (Shift Arithmetic Left/Shift Logical Left)
SALMB/SHLMB (Shift Left, Memory Byte)

Form:	SAL/SHL destination, count
Flags:	O D I T S Z A P C X X
Operands:	register, 1 register, CL memory, 1 memory, CL
Coding Example:	SAL AL, 1

SAR (Shift Arithmetic Right)
SARMB (Shift Arithmetic Right, Memory Byte)

Form:	SAR destination, count
Flags:	O D I T S Z A P C X X X U X X
Operands:	register, 1 register, CL memory, 1 memory, CL
Coding Example:	SAR DI, CL

SBB (Subtract with Borrow)
SBBBIM (Subtract with Borrow, Immediate Byte)
SBBM (Subtract with Borrow, Direct Addressing Mode)

Form:	SBB destination, source
Flags:	O D I T S Z A P C X X X X X X
Operands:	register, register register, memory memory, register accumulator, immediate register, immediate memory, immediate
Coding Example:	SBB BX, CX

SCASB (Scan String, Bytewise)
SCASW (Scan String, Wordwise)

Form:	SCASW dest-string
Flags:	O D I T S Z A P C X X X X X X
Operands:	dest-string (repeat) dest-string
Coding Example:	REPNE SCASB BUFFER

SHR (Shift Logical Right)
SHRMB (Shift Logical Right, Memory Byte)

Form:	SHR destination, count
Flags:	O D I T S Z A P C X X
Operands:	register, 1 register, CL memory, 1 memory, CL
Coding Example:	SHR SI, 1

STC (Set Carry Flag)

Form:	STC (no operands)
Flags:	O D I T S Z A P C 1
Operands:	None
Coding Example:	STC

STD (Set Direction Flag)

Form:	STD (no operands)
Flags:	O D I T S Z A P C 1
Operands:	None
Coding Example:	STD

STI (Set Interrupt Enable Flag)

Form:	STI (no operands)
Flags:	O D I T S Z A P C 1
Operands:	None
Coding Example:	STI

STOSB (Store Byte String)
STOSW (Store Word String)

Form:	STOSB dest-string
Flags:	O D I T S Z A P C
Operands:	dest-string (repeat) dest-string
Coding Example:	REP STOSB DISPLAY

SUB (Subtraction)
SUBBIM (Subtraction, Immediate Byte)
SUBM (Subtraction, Direct Addressing Mode)

Form:	SUB destination, source
Flags:	O D I T S Z A P C X X X X X X
Operands:	register, register register, memory memory, register accumulator, immediate register, immediate memory, immediate
Coding Example:	SUB CX, BX

TEST (Test or Nondestructive logical AND)
TESTBIM (Test, Immediate Byte)
TESTM (Test, Direct Addressing Mode)

Form:	TEST destination, source
Flags:	O D I T S Z A P C 0 X X U X 0
Operands:	register, register register, memory accumulator, immediate register, immediate memory, immediate
Coding Example:	TEST SI, END_COUNT

WAIT (Wait While TEST Pin not Asserted)

Form:	WAIT (no operands)
Flags:	O D I T S Z A P C
Operands:	None
Coding Example:	WAIT

XCHG (Exhange)

Form:	XCHG destination, source
Flags:	O D I T S Z A P C
Operands:	accumulator, reg16 memory, register register, register
Coding Example:	XCHG AX, BX

XLAT (Translate)

Form:	XLAT (source-table)
Flags:	O D I T S Z A P C
Operands:	source-table
Coding Example:	XLAT ASCII_TAB

XOR (Logical Exclusive OR)
XORBIM (Logical Exclusive OR, Immediate Byte)
XORM (Logical Exclusive OR, Direct Addressing Mode)

Form:	XOR destination, source
Flags:	O D I T S Z A P C 0 X X U X 0
Operands:	register, register register, memory memory, register accumulator, immediate register, immediate memory, immediate
Coding Example:	XOR CL, MASK_BYTE

8087 FLOATING POINT OPERATORS

The following is a reference list of the 8087 floating point operators. (The introduction to this chapter describes the differences between the UCSD p-System 8087 Assembler mnemonics suffixes and the standard Intel mnemonics.)

Key to 8087 Exception Codes

I = Invalid Operand
Z = Zero Divide
D = Denormalized
O = Overflow
U = Underflow
P = Precision

Many instructions allow you to code their operands in more than one way. For example, you can write FADD (add real) without operands, with only a source, or with a destination and a source. The instruction descriptions in this section employ the simple convention of separating alternative operand forms with slashes; the slashes, however, are not coded. Consecutive slashes indicate an option of no explicit operands. Thus, the operands for FADD are described as:

//source/destination,source

This means that you can write FADD in any of three ways:

FADD
FADD source
FADD destination,source

ST indicates the top of the stack. ST(i) indicates a stack element where i is a three-bit quantity in the range zero to seven. (See the Intel documentation for a complete description of this.)

FABS (Absolute Value)

Form:	FABS (no operands)
Operands:	None
Exceptions:	I
Coding Example:	FABS

FADD (Add Real)

Form:	FADD //source/destination, source)
Operands:	//ST,ST(i)/ST(i),ST short-real long-real
Exceptions:	I, D, O, U, P
Coding Example:	FADD ST, ST(4)

FADDP (Add Real and Pop)

Form:	FADDP destination, source)
Operands:	ST(i),ST
Exceptions:	I, D, O, U, P
Coding Example:	FADDP ST(2), ST

FBLD (Packed Decimal (BCD) load)

Form:	FBLD source
Operands:	packed decimal
Exceptions:	I
Coding Example:	FBLD YTD SALES

FBSTP (Packed Decimal (BCD) Store and Pop)

Form:	FBSTP source
Operands:	packed decimal
Exceptions:	I
Coding Example:	FBSTP (BX).FORCAST

FCHS (Change Sign)

Form:	FCHS (no operands)
Operands:	None
Exceptions:	I
Coding Example:	FCHS

FCLEX/FNCLEX (Clear Exceptions)

Form:	FCLEX/FNCLEX (no operands)
Operands:	None
Exceptions:	None
Coding Example:	FCLEX

FCOM (Compare Real)

Form:	FCOM //source
Operands:	//ST(i) short-real long-real
Exceptions:	I, D
Coding Example:	FCOM ST(1)

FCOMP (Compare Real and Pop)

Form:	FCOMP //source
Operands:	//ST(i) short-real long-real
Exceptions:	I, D
Coding Example:	FCOMP ST(2)

FCOMPP (Compare Real and Pop Twice)

Form:	FCOMPP (no operands)
Operands:	None
Exceptions:	I, D
Coding Example:	FCOMPP

FDECSTP (Decrement Stack Pointer)

Form:	FDECSTP (no operands)
Operands:	None
Exceptions:	None
Coding Example:	FDECSTP

FDISI/FNDISI (Disable Interrupts)

Form:	FDISI/FNDISI (no operands)
Operands:	None
Exceptions:	None
Coding Example:	FDISI

FDIV (Divide Real)

Form:	FDIV (//source/destination,source)
Operands:	//ST(i), ST short-real long-real
Exceptions:	I, D, Z, O, U, P
Coding Example:	FDIV ARC(DI)

FDIVP (Divide Real and Pop)

Form:	FDIVP destination, source
Operands:	ST(i), ST
Exceptions:	I, D, Z, O, U, P
Coding Example:	FDIVP ST(4), ST

FDIVR (Divide Real Reversed)

Form:	FDIVR destination, source
Operands:	//ST,ST(i)/ST(i), ST short-real long-real
Exceptions:	I, D, Z, O, U, P
Coding Example:	FDIVR ST(2), ST

FDIVRP (Divide Real Reversed and Pop)

Form:	FDIVRP destination, source
Operands:	ST(i), ST
Exceptions:	I, D, Z, O, U, P
Coding Example:	FDIVRP ST(1), ST

FENI/FNENI (Enable Interrupts)

Form:	FENI/FNENI (no operands)
Operands:	None
Exceptions:	None
Coding Example:	FENI

FFREE (Free Register)

Form:	FFREE destination
Operands:	ST(i)
Exceptions:	None
Coding Example:	FFREE

FIADD (Integer Add)

Form:	FIADD source
Operands:	word-integer short-integer
Exceptions:	I, D, O, P
Coding Example:	FIADD DISTANCE TRAVELED

FICOM (Integer Compare)

Form:	FICOM source
Operands:	word-integer short-integer
Exceptions:	I, D
Coding Example:	FICOM TOOL.N PASSES

FICOMP (Integer Compare and Pop)

Form:	FICOMP source
Operands:	word-integer short-integer
Exceptions:	I, D
Coding Example:	FICOMP N SAMPLES

FIDIV (Integer Divide)

Form:	FIDIV source
Operands:	word-integer short-integer
Exceptions:	I, D, Z, O, U, P
Coding Example:	FIDIV RELATIVE ANGLE(DI)

FIDIVR (Integer Divide Reversed)

Form:	FIDIVR source
Operands:	word-integer short-integer
Exceptions:	I, D, Z, O, U, P
Coding Example:	FIDIVR FREQUENCY

FILD (Integer Load)

Form:	FILD source
Operands:	word-integer short-integer long-integer
Exceptions:	I
Coding Example:	FILD (BX).SEQUENCE

FIMUL (Integer Multiply)

Form:	FIMUL source
Operands:	word-integer short-integer
Exceptions:	I, D, O, P
Coding Example:	FIMUL BEARING

FINCSTP (Increment Stack Pointer)

Form:	FINCSTP
Operands:	None
Exceptions:	None
Coding Example:	FINCSTP

FINIT/FNINIT (Initialize Processor)

Form:	FINIT
Operands:	None
Exceptions:	None
Coding Example:	FNINIT

FIST (Integer Store)

Form:	FIST destination
Operands:	word-integer short-integer
Exceptions:	I, P
Coding Example:	FIST OBS.COUNT(SI)

FISTP (Integer Store and Pop)

Form:	FISTP destination
Operands:	word-integer short-integer long-integer
Exceptions:	I, P
Coding Example:	FISTP (BX).ALPHA_COUNT(SI)

FISUB (Integer Subtract)

Form:	FISUB source
Operands:	word-integer short-integer
Exceptions:	I, D, O, P
Coding Example:	FISUB BASE_FREQUENCY

FISUBR (Integer Subtract Reversed)

Form:	FISUBR source
Operands:	word-integer short-integer
Exceptions:	I, D, O, P
Coding Example:	FISUBR BALANCE

FLD (Load Real)

Form:	FLD source
Operands:	ST(i) short-real long-real temp-real
Exceptions:	I, D
Coding Example:	FLD ST(0)

FLDCW (Load Control Word)

Form:	FLDCW source
Operands:	2-bytes
Exceptions:	None
Coding Example:	FLDCW CONTROL WORD

FLDENV (Load Environment)

Form:	FLDENV source
Operands:	14-bytes
Exceptions:	None
Coding Example:	FLDENV(BP+6)

FLDLG2 (Load Log (Base 10) of 2)

Form:	FLDLG2
Operands:	None
Exceptions:	I
Coding Example:	FLDLG2

FLDLN2 (Load Log (Base E) of 2)

Form:	FLDLN2
Operands:	None
Exceptions:	I
Coding Example:	FLDLN2

FLDL2E (Load Log (Base 2) of E)

Form:	FLDL2E
Operands:	None
Exceptions:	I
Coding Example:	FLDL2E

FLDL2T (Load Log (Base 2) of 10)

Form:	FLDL2T
Operands:	None
Exceptions:	I
Coding Example:	FLDL2T

FLDPI (Load Pi)

Form:	FLDPI
Operands:	None
Exceptions:	I
Coding Example:	FLDPI

FLDZ (Load +0.0)

Form:	FLDZ
Operands:	None
Exceptions:	I
Coding Example:	FLDZ

FLD1 (Load +1.0)

Form:	FLD1
Operands:	None
Exceptions:	I
Coding Example:	FLD1

FMUL (Multiply Real)

Form:	FMUL //source/destination, source
Operands:	//ST(i),ST/ST,ST(i) short-real long-real
Exceptions:	I, D, O, U, P
Coding Example:	FMUL SPEED_FACTOR

FMULP (Multiply Real and Pop)

Form:	FMULP destination, source
Operands:	ST(i),ST
Exceptions:	I, D, O, U, P
Coding Example:	FMULP ST(1),ST

FNOP (No Operation)

Form:	FNOP
Operands:	None
Exceptions:	None
Coding Example:	FNOP

FPATAN (Partial Arctangent)

Form:	FPATAN
Operands:	None
Exceptions:	U, P (operands not checked)
Coding Example:	FPATAN

FPREM (Partial Remainder)

Form:	FPREM
Operands:	None
Exceptions:	I, D, U
Coding Example:	FPREM

FPTAN (Partial Tangent)

Form:	FPTAN
Operands:	None
Exceptions:	I,P (operands not checked)
Coding Example:	FPTAN

FRNDINT (Round to Integer)

Form:	FRNDINT
Operands:	None
Exceptions:	I, P
Coding Example:	FRNDINT

FRSTOR (Restore Saved State)

Form:	FRSTOR source
Operands:	94-bytes
Exceptions:	None
Coding Example:	FRSTOR (BP)

FSAVE/FNSAVE (Save State)

Form:	FSAVE destination
Operands:	94-bytes
Exceptions:	None
Coding Example:	FSAVE (BP)

FSCALE (Scale)

Form:	FSCALE
Operands:	None
Exceptions:	I, O, U
Coding Example:	FSCALE

FSQRT (Square Root)

Form:	FSQRT
Operands:	None
Exceptions:	I, D, P
Coding Example:	FSQRT

FST (Store Real)

Form:	FST destination
Operands:	ST(i) short-real long-real
Exceptions:	I, O, U, P
Coding Example:	FST MEAN READING

FSTCW/FNSTCW (Store Control Word)

Form:	FSTCW destination
Operands:	2-bytes
Exceptions:	None
Coding Example:	FSTCW SAVE_CTRL

FSTENV/FNSTENV (Store Environment)

Form:	FSTENV destination
Operands:	14-bytes
Exceptions:	None
Coding Example:	FSTENV (BP)

FSTP (Store Real and Pop)

Form:	FSTP destination
Operands:	ST(i) short-real long-real temp-real
Exceptions:	I, O, U, P
Coding Example:	FSTP ST(2)

FSTSW/FNSTSW (Store Status Word)

Form:	FSTSW destination
Operands:	2-bytes
Exceptions:	None
Coding Example:	FSTSW SAVE_STATUS

FSUB (Subtract Real)

Form:	FSUB //source/destination, source
Operands:	//ST,ST(i)/ST(i),ST short-real long-real
Exceptions:	I, D, O, U, P
Coding Example:	FSUB BASE_VALUE

FSUBP (Subtract Real and Pop)

Form:	FSUBP destination, source
Operands:	ST(i),ST
Exceptions:	I, D, O, U, P
Coding Example:	FSUBP ST(2),ST

FSUBR (Subtract Real Reversed)

Form:	FSUB //source/destination, source
Operands:	//ST,ST(i)/ST(i),ST short-real long-real
Exceptions:	I, D, O, U, P
Coding Example:	FSUBR (BX).INDEX

FSUBRP (Subtract Real Reversed and Pop)

Form:	FSUBRP destination, source
Operands:	ST(i),ST
Exceptions:	I, D, O, U, P
Coding Example:	FSUBRP ST(2),ST

FTST (Test Stack Top Against 0.0)

Form:	FTST
Operands:	None
Exceptions:	I, D
Coding Example:	FTST

FWAIT (CPU Wait While 8087 is Busy)

Form:	FWAIT
Operands:	None
Exceptions:	None (CPU instruction)
Coding Example:	FWAIT

FXAM (Examine Stack Top)

Form:	FXAM
Operands:	None
Exceptions:	None
Coding Example:	FXAM

FXCH (Exchange Registers)

Form:	FXCH //destination
Operands:	//ST(i)
Exceptions:	I
Coding Example:	FXCH ST(2)

FXTRACT (Extract Exponent and Significand)

Form:	FXTRACT
Operands:	None
Exceptions:	I
Coding Example:	FXTRACT

FYL2X (Y * Log (Base 2) of X)

Form:	FYL2X
Operands:	None
Exceptions:	P (operands not checked)
Coding Example:	FYL2X

FYL2XP1 (Y * Log (Base 2) of (X+1))

Form:	FYL2XP1
Operands:	None
Exceptions:	P (operands not checked)
Coding Example:	FYL2XP1

A

The Linker

The linker is an item on the command menu which allows assembled code to be linked into a Pascal 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 function 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 are not in **SYSTEM.LIBRARY*, you are responsible for manually linking the code before executing it.

When the linker is called automatically and cannot 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 by pressing the **RETURN** key, 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 the user presses the **RETURN** key. 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:

```
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 the user presses the **RETURN** key, **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.

B

The Compress Utility

The Compress utility program takes input as code files consisting of one or more linked assembly procedures and produces object files suitable for applications outside the UCSD 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 a user 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 should not 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 execute *COMPRESS.CODE*. It will respond with the following prompt.

```
Execute what file?
```

Enter *COMPRESSOR.CODE* and press the RETURN key. The system will display 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. Enter it as a sequence of one to four hexadecimal digits and press the RETURN key. 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 is not necessary to enter the *.CODE* suffix. If the file cannot 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 does not append a .CODE suffix). Pressing the **RETURN** key causes the output file to have the same name as the input file, thus eliminating the original input file. If the file cannot be opened, Compress will print an error message and abort.

In all the previous prompts, pressing the exclamation point 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 does not have sufficient memory space.

While running, Compress displays for each procedure the starting and ending addresses (in hexadecimal) and the length in bytes. At completion Compress displays the total number of bytes in the output file. 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 user 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 pad bytes, which are emitted (when necessary) to preserve the word alignment of all procedures. Code files always contain integral numbers 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 the user relocates a file towards the high end of the 16-bit address space, you must ensure that the relocated file will not 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.

C

Errors

0:	
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 first 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: 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
88: Illegal to use BP indirect alone
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: Not implemented
107: Open parenthesis expected
108: Expected register alone as right operand
109: Segovpre then regalone, that is 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) Improper 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 (

Index

THREE-MONTH LIMITED WARRANTY TEXAS INSTRUMENTS PROFESSIONAL COMPUTER SOFTWARE MEDIA

TEXAS INSTRUMENTS INCORPORATED EXTENDS THIS CONSUMER WARRANTY ONLY TO THE ORIGINAL CONSUMER/PURCHASER.

WARRANTY DURATION

The media is warranted for a period of three (3) months from the date of original purchase by the consumer.

Some states do not allow the exclusion or limitation of incidental or consequential damages or limitations on how long an implied warranty lasts, so the above limitations or exclusions may not apply to you.

WARRANTY COVERAGE

This limited warranty covers the cassette or diskette (media) on which the computer program is furnished. It does not extend to the program contained on the media or the accompanying book materials (collectively the Program). The media is warranted against defects in material or workmanship. THIS WARRANTY IS / VOID IF THE MEDIA HAS BEEN DAMAGED BY ACCIDENT, UNREASONABLE USE, NEGLECT, IMPROPER SERVICE, OR OTHER CAUSES NOT ARISING OUT OF DEFECTS IN MATERIALS OR WORKMANSHIP.

PERFORMANCE BY TI UNDER WARRANTY

During the above three-month warranty period, defective media will be replaced when it is returned postage prepaid to a Texas Instruments Service Facility listed below or an authorized Texas Instruments Professional Computer Dealer with a copy of the purchase receipt. The replacement media will be warranted for three months from date of replacement. Other than the postage requirement (where allowed by state law), no charge will be made for the replacement. TI strongly recommends that you insure the media for value prior to mailing.

WARRANTY AND CONSEQUENTIAL DAMAGES DISCLAIMERS

ANY IMPLIED WARRANTIES ARISING OUT OF THIS SALE INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO THE ABOVE THREE-MONTH PERIOD. TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL COSTS, EXPENSES, OR DAMAGES INCURRED BY THE CONSUMER OR ANY OTHER USER ARISING OUT OF THE PURCHASE OR USE OF THE MEDIA. THESE EXCLUDED DAMAGES INCLUDE, BUT ARE NOT LIMITED BY, COST OF REMOVAL OR REINSTALLATION, OUTSIDE COMPUTER TIME, LABOR COSTS, LOSS OF GOODWILL, LOSS OF PROFITS, LOSS OF SAVINGS, OR LOSS OF USE OR INTERRUPTION OF BUSINESS.

LEGAL REMEDIES

This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

TEXAS INSTRUMENTS CONSUMER SERVICE FACILITIES

U.S. Residents:

Texas Instruments
Service Facility
P.O. Box 1444, MS 7758
Houston, Texas 77001

Canadian Residents:

Geophysical Service Inc.
41 Shelley Road
Richmond Hill, Ontario
Canada L4C 5G4

Consumers in California and Oregon may contact the following Texas Instruments offices for additional assistance or information.

Texas Instruments
Consumer Service
831 South Douglas St.
Suite 119
El Segundo, California 90245
(213) 973-2591

Texas Instruments
Consumer Service
6700 S.W. 105th
Kristin Square, Suite 110
Beaverton, Oregon 97005
(503) 643-6758

IMPORTANT NOTICE OF DISCLAIMER REGARDING THE PROGRAM

The following should be read and understood before using the software media and Program.

TI does not warrant that the Program will be free from error or will meet the specific requirements of the purchaser/user. The purchaser/user assumes complete responsibility for any decision made or actions taken based on information obtained using the Program. Any statements made concerning the utility of the Program are not to be construed as expressed or implied warranties.

TEXAS INSTRUMENTS MAKES NO WARRANTY, EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING THE PROGRAM AND MAKES ALL PROGRAMS AVAILABLE SOLELY ON AN "AS IS" BASIS.

IN NO EVENT SHALL TEXAS INSTRUMENTS BE LIABLE FOR SPECIAL, COLLATERAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF THE PURCHASE OR USE OF THE PROGRAM. THESE EXCLUDED DAMAGES INCLUDE, BUT ARE NOT LIMITED BY, COST OF REMOVAL OR REINSTALLATION, OUTSIDE COMPUTER TIME, LABOR COSTS, LOSS OF GOODWILL, LOSS OF PROFITS, LOSS OF SAVINGS, OR LOSS OF USE OR INTERRUPTION OF BUSINESS. THE SOLE AND EXCLUSIVE LIABILITY OF TEXAS INSTRUMENTS, REGARDLESS OF THE FORM OF ACTION, SHALL NOT EXCEED THE PURCHASE PRICE OF THE PROGRAM. TEXAS INSTRUMENTS SHALL NOT BE LIABLE FOR ANY CLAIM OF ANY KIND WHATSOEVER BY ANY OTHER PARTY AGAINST THE PURCHASER/USER OF THE PROGRAM.

COPYRIGHT

TEXAS INSTRUMENTS PROFESSIONAL COMPUTER
UCSD p-System Assembler
TI Part No. 2232402-0001

Original Issue: 15 April 1983

Your Name: _____

Company: _____

Telephone: _____

Department: _____

Address: _____

City/State/Zip Code: _____

Your comments and suggestions assist us in improving our products. If your comments concern problems with this manual, please list the page number.

Comments:

This form is not intended for use as an order blank.

FOLD

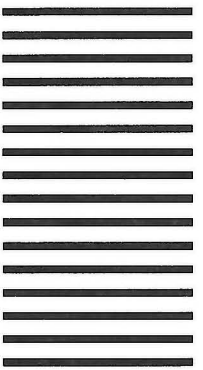
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

Texas Instruments Incorporated
Attn: Marketing M/S 7896
P.O. Box 1444
Houston, TX 77001



FOLD

TEXAS INSTRUMENTS PROFESSIONAL COMPUTER
UCSD p-System Assembler
TI Part No. 2232402-0001

Original Issue: 15 April 1983

Your Name: _____

Company: _____

Telephone: _____

Department: _____

Address: _____

City/State/Zip Code: _____

Your comments and suggestions assist us in improving our products. If your comments concern problems with this manual, please list the page number.

Comments:

This form is not intended for use as an order blank.

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

Texas Instruments Incorporated
Attn: Marketing M/S 7896
P.O. Box 1444
Houston, TX 77001

FOLD

TEXAS INSTRUMENTS PROFESSIONAL COMPUTER
UCSD p-System Assembler
TI Part No. 2232402-0001

Original Issue: 15 April 1983

Your Name: _____

Company: _____

Telephone: _____

Department: _____

Address: _____

City/State/Zip Code: _____

Your comments and suggestions assist us in improving our products. If your comments concern problems with this manual, please list the page number.

Comments:

This form is not intended for use as an order blank.

FOLD

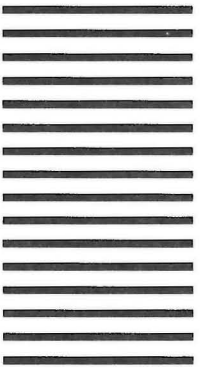
NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 6189 HOUSTON, TX

POSTAGE WILL BE PAID BY ADDRESSEE

Texas Instruments Incorporated
Attn: Marketing M/S 7896
P.O. Box 1444
Houston, TX 77001



FOLD

NOTES

NOTES

Texas Instruments reserves the right to change its product and service offerings at any time without notice.

TEXAS INSTRUMENTS

Part No. 2232402-0001

Printed in U.S.A.