AD-778 310

FAIL

STANFORD UNIVERSITY

PREPARED FOR

ADVANCED RESEARCH PROJECTS AGENCY

APRIL 1974

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD 778 310

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER STAN-CS-74-407	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) FAIL		5. TYPE OF REPORT & PERIOD COVERED Technical (April 1974)
		6. PERFORMING ORG. REPORT NUMBER Same as 1
7. AUTHOR(s) F.H.G. Wright II and R.E. Gorin		8. CONTRACT OR GRANT NUMBER(s) ARPA DAHC15-73-C-0435
9. PERFORMING ORGANIZATION NAME AND ADDRESS Stanford University Computer Science Department Stanford CA 94305		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ARPA 2494
11. CONTROLLING OFFICE NAME AND ADDRESS ARPA/IPT, Attn: Stephen D. Crocker 1400 Wilson Blvd., Arlington VA 22209		12. REPORT DATE April 1974
		13. NUMBER OF PAGES ~~61~~ 68
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) ONR Representative: Jack Ducey Durand Aeronautics Bldg., Rm.165 Stanford University Stanford CA 94305		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

releasable without limitations on dissemination

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This is a reference manual for FAIL, a fast, one-pass assembler for PDP-10 and PDP-6, machine language. FAIL statements, pseudo-operations, macros, and conditional assembly features are described. Although FAIL uses substantially more main memory than MACRO-10, it assembles typical programs about five times faster. FAIL assembles the entire Stanford time-sharing operating system (two million characters) in less than four minutes of CPU time on a KA-10 processor. FAIL permits an ALGOL-style block structure which provides a way

68

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

of localizing the usage of some symbols to certain parts of the program, such that the same symbol name can be used to mean different things in different blocks.

Acknowledgments

The original version of FAIL and the original manual (SAILON-26) were written by Phil Petit in 1967. Various additions and modifications were subsequently contributed by William Weiher, Fred Wright, Ralph Gorin, and others. This manual was prepared using PUB, the document compiler created by Larry Tesler, using the Xerox Graphics Printer, with fonts by Brian Harvey. Brian McCune and Les Earnest reviewed the manuscript and made helpful suggestions. Cover picture: United Press International.

Table of Contents

1. Introduction

FAIL is an assembly program for PDP-6 and PDP-10 machine language. FAIL operates in one pass, which means that it reads the input file only once; the linking loader program (LOADER or LINK-10) completes any aspects of the assembly which could not be done by FAIL. The efficiencies which have been employed in its coding make FAIL five times faster than MACRO-10, the DEC assembler.

FAIL processes source program statements by translating mnemonic operation codes into the binary codes needed in machine instructions, relating symbols to numeric values, and assigning relocatable or absolute core addresses for program instructions and data. The assembler can prepare a listing of the program which includes a representation of the assembled code. Also, the assembler notifies the user of any errors detected during the assembly.

FAIL has a powerful macro processor which allows the programmer to create new language elements to perform special functions for each programming job.

FAIL permits an ALGOL-style block structure which provides a way of localizing the usage of symbols to particular parts of the program, called *blocks*. Block structure allows the same symbol name to be given different meanings in different blocks.

The reader of this manual should be familiar with the PDP-10 instruction set, which is described in both *DECsystem-10 System Reference Manual* and *PDP-10 and PDP-6 Instruction Sets* (SAILON-71).

Other documents of interest:

Frost, M.	*UUO Manual*, SAILON-55.3, December 1973
Petit, P.	*RAID*, SAILON-58, September 1969
Harvey, B.	*Monitor Command Manual*, SAILON-54.3, December 1973

The following are available in the *DECsystem-10 Software Notebooks*:

Cross-Reference Listing: CREF, June 1973
DDT-10 Programmer's Reference Manual, June 1973
Linking Loader Programmer's Reference Manual, August 1971
LINK-10 Programmer's Reference Manual, May 1973
MACRO-10 Assembler Programmer's Reference Manual, June 1972
DECsystem-10 Operating System Commands, February 1974
DECsystem-10 Monitor Calls, June 1973

2. Basic Syntax

This section describes the basic components of a typical FAIL source program. It covers the normal mode of turning each source statement into a binary word. Pseudo-operations and macro features are explained in later sections.

This section is organized in a *top-down* manner: the complex constructs, statements, are described first, followed by a description of the language elements from which statements are built, etc.

Statements are the elements of the language that generate machine code and other binary data. A statement is generally free format, consisting of several *fields*, each of which is an *expression*. Expressions are composed of *atoms* and *operators*. The operators signify typical arithmetic and boolean operations, such as addition or logical OR. Atoms are either *constants*, *symbols*, or *complex atoms*.

2.1 Statements

Statements are the syntactic units which actually produce code. The statements that are described in this section usually generate one word of code. A *null statement*, which consists of no expressions, generates no code. A typical statement consists of one or more expressions separated by spaces, commas, or parentheses.

There are five kinds of statements: *instruction statements*, *full-word expressions*, *truncated expressions*, *halfword statements*, and *input-output statements*. The most common of these is the instruction statement. Also, there are pseudo-operations (called *pseudo-ops*), which are described in section 3, page 22. A pseudo-op may direct FAIL to perform an assembler control function or to assemble data in a particular format.

The examples that are given below are intended to be as general as possible. In most cases, many of the indicated fields may be omitted.

2.1.1 Instruction Statement

```
OPCODE AC,@ADDRESS(INDEX)        ;COMMENT
```

An *instruction statement* is used to assemble one machine instruction. The typical format is shown above; the parts will be explained later. Any portion of the instruction statement may be omitted. The comment field is not really part of the instruction statement, but may be included on the same line for clarity and conciseness. The parts may appear in any order, except that the opcode field, if present, must be the first expression. Also, each part must be syntactically identifiable. The form above is hallowed by years of use; departure from it will render a program less intelligible to other readers.

If the opcode field is omitted, all other fields will be recognized and handled normally, unless the address expression is the first field seen, in which case the statement is treated as a full-word expression.

2.1.1.1 Opcode Field

If the first atom appearing in the statement (excluding labels and assignment statements) is an identifier, it will be looked up in the opcode table to see if it is an opcode, in which case the opcode alone will be returned as the first expression, overriding any significance it may have as a symbol. An *opcode* (short for *operation code*) may be a machine instruction mnemonic, a UUO mnemonic, a pseudo-op, or a user-defined opcode (see OPDEF in section 3.2.1, page 26). An opcode, if it appears, must be the first thing in the statement (except for labels or assignment statements).

If an opcode is a pseudo-op mnemonic, FAIL will process that particular pseudo-op as appropriate. The syntax of pseudo-ops differs from that of normal statements.

If an opcode is a machine instruction, UUO mnemonic, or user opcode, its value is placed in the binary word being assembled. These opcodes are treated as having full-word values, but in most cases only the *opcode field* (bits 0-8) is non-zero. A few machine instructions, and many UUO mnemonics, specify values for other fields as well. The values of the other fields (except the address field, if non-zero) can be modified by subsequent operands.

Whenever an opcode is recognized, it is immediately processed without regard for any arithmetic operator that might follow. Although FAIL tries to allow a symbol and opcode with the same name to co-exist, it cannot resolve the ambiguity in all circumstances; it is a good idea to avoid conflicts as much as possible. FAIL will *not* recognize an identifier as an opcode if the identifier is followed by any one of the characters colon (:), left-arrow (←), up-arrow (↑), tilde (~), or number sign (#).

2.1.1.2 Accumulator (AC) Field

If an expression appears in a statement followed by exactly one comma, its value will be placed in the *accumulator field* of the current word (bits 9-12), possibly replacing the accumulator field indicated by an opcode. This expression must be defined, available, and absolute (some of these terms are defined in section 2.3.2, page 8). For the sake of brevity, "accumulator" is often written as "AC".

2.1.1.3 Indirect (@) Field

If one or more at-sign characters (@) appear as part of a statement, the *indirect bit* (bit 13) will be turned on in the word being assembled. The at-sign may appear anywhere in the statement as long as it is not embedded inside symbols or expressions. The character open single quote (') may be used as an alternative to at-sign.

2.1.1.4 Address Field

If in a statement an expression appears which is neither enclosed in parentheses nor followed by a comma, it is considered to be an *address expression* unless it is the first expression (including the opcode) in the statement. Address expressions are truncated to 18 bits and placed in the *address field* (bits 18-35) of the word being assembled.

Only one address field may be assembled per statement; an attempt to assemble more than one is an error. This error sometimes occurs because an undefined opcode is used, which is treated as an expression in case it is really an undefined symbol. This error can also occur when an opcode includes an address field and the user attempts to supply another address field.

2.1.1.5 Index Field

If an expression is enclosed in parentheses in a statement, the right half of its value will be ORed into the left half of the current word. Also, if no address field has appeared yet, the left half of its value will be ORed into the right half of the current word. The expression must be defined, available, and absolute. This construct is most commonly used for specifying the *index field* (bits 14-17)

Sometimes, this construct is used for putting left-half quantities in address fields, or as a general halfword-swapping operation. Often when this is done, the expression in parentheses must be enclosed in brokets (< and >) to force its evaluation as an atomic statement; see section 2.3.5.1, page 20. If the left half of the expression is non-zero, the word will be flagged as containing an address field, making another address field illegal.

Examples:

```
MOVEI 2,-1(6)          ;assembles 201106 777777
MOVSI 1,(<JRST>)       ;assembles 205040 254000
```

2.1.2 Halfword Statement

```
EXPR,,@ADDRESS(INDEX)          ;COMMENT
```

If an expression is followed by *comma-comma* (,,), it will be placed in the left halfword of the current location, and FAIL will continue to process an address field, index field, and indirect field. This is more convenient than the XWD pseudo-op for assembling halfwords since it allows the entire effective address to be specified in the usual way. The only restriction is to beware of possible interpretation of the first symbol as an opcode. If the expression followed by the comma-comma is not the first thing assembled in the word, the warning message *Illegal ,,* will be printed, although the statement will assemble correctly. This prevents confusion if an extra comma is typed after an accumulator field.

2.1.3 Full-Word Expression

```
EXPR                                ;COMMENT
```

When the first expression in a statement is not preceded by a comma and is not an opcode, FAIL assumes that the expression is a *full-word expression*. The entire 36-bit value of the expression is placed in the current word. The full-word expression is the only ordinary statement (i.e., not a pseudo-op) that assembles a single expression with a full 36-bit value. Full-word expressions are treated as address fields for purposes of the multiple address field error.

If a full-word expression contains any undefined symbols, unavailable symbols, or strange relocation constants, the entire word will be updated with the value of the expression when it becomes known. This will obliterate any index, indirect, or accumulator field appearing after the expression on the line. If the expression actually has only an 18-bit value, this can be fixed by prefixing the expression with a comma (i.e., by using a truncated expression). If a full-word value is actually needed and the problem is not just one of availability (curable by the use of GLOBAL or down-arrow (↓); see section 2.3.4.6, page 16), it may be necessary to use an explicit expression to set the accumulator, index, and indirect fields.

2.1.4 Truncated Expression

```
,EXPR                               ;COMMENT
```

If a comma appears before any expression in a statement, it flags the current word as containing data in order to force a subsequent expression to be treated as an address field even when it is the only expression in the statement. This can be used to form an 18-bit *truncated expression*. Note that a statement consisting of a single comma will assemble a zero word.

2.1.5 Input-Output Instruction Statement

```
OPCODE DEV,@ADDRESS(INDEX)          ;COMMENT
```

An *input-output instruction statement* is used to assemble one hardware I/O instruction. Most parts are the same as in an instruction statement, except that a *device selection field* appears instead of an accumulator field. Also, the opcode portion must be one of the PDP-10 *hardware input-output instructions* (e.g., DATAO). Note that hardware I/O instructions are *not* related to operating system UUOs.

2.1.5.1 Device Selection Field

The same syntax and restrictions that apply to an accumulator field apply also to the *device selection field*. The value of the device selection field is placed in bits 3-9 of the current word. This value is often called the *device code*.

2.1.6 Comment Field

When FAIL's statement processor encounters a carriage return or semicolon (;), all characters up to the next line feed or form feed are completely ignored except for listing and certain macro processor functions (see section 4.1, page 38). Upon reaching the line feed or form feed, the comment is terminated. Usually, this is used to insert a relevant comment at the end of a line of code.

2.1.7 Statement Termination

A statement is terminated by a comment or by any of the characters line feed, double-arrow (↔), right bracket (]), or right broket (>) when not processing a comment. When a statement is terminated, the value of the current word (if any) is returned. A statement returns no value at all if no expressions appear in it or if it is a pseudo-op which assembles no code. Terminating a statement with one of the bracket characters often has special significance, as in atomic statements or literals. Double-arrow can be used for assembling more than one statement on a line, but will not terminate a comment.

2.2 Expressions

Expressions are built from *atoms* connected by *operators* which allow the specification of values based upon arithmetic and logical functions of several values. These expressions follow essentially the same rules as conventional programming languages. Each operand in an expression may be an atom, an atomic statement, or an expression in parentheses, preceded by any number of unary operators. If parentheses are used, the expression inside the parentheses is evaluated before performing any operations using that operand. If a unary operator appears, its function will be evaluated before any operations using that operand (but after the expression in parentheses, if parentheses are used). Multiple unary operators are evaluated from right to left, so -¬1 is processed as -(¬1). Finally, these operands can be connected with binary infix operators whose order of evaluation is determined by their assigned precedence levels (highest first) and is left-to-right for operators of the same level. An expression may, of course, consist of a single operand (i.e., atom) with no operators at all.

Surrounding an entire expression with parentheses sometimes signifies an index field (see section 2.1.1.5, page 4). All arithmetic is integer or boolean; no type conversion is done for floating-point operands.

The following is a list of the available operators and their precedence levels:

Symbol	Meaning	Precedence Level
	binary operators	
+	Addition	1
-	Subtraction	1
*	Multiplication	2
/	Division	2
&	Logical AND	3
∧	Logical AND	3
!	Logical OR	3
∨	Logical OR	3
≠	Exclusive OR	3
≥	Exclusive OR	3
⊗	Logical Left-Shift	4
	unary operators	
-	Negation (two's complement)	5
¬	Logical NOT (one's complement)	5

If an expression contains any undefined values, its own value is undefined. If an expression is used in a context where undefined values are legal, FAIL retains a structure describing the evaluation needed, called a *Polish fixup* for its similarity to Polish arithmetic notation, in order to complete the evaluation when the unknowns become defined. As soon as all values in the expression are defined, a fixup will be output (to the loader) to correct the value (or the value will be corrected directly in the case of a literal). If the expression is not completely defined by the end of the assembly (due to external references or errors), the Polish structure is sent to the loader for evaluation at load time. In other words, the right thing usually happens with a partially undefined expression as long as it is legal in the context where it is used.

Expressions may also begin with any number of labels or assignment statements, which have no effect on the value of the expression.

Examples

```
FOO⊗2            ;value of FOO shifted left 2 bits
(BAR-1)⊗-2       ;value of BAR-1 shifted right 2 bits
(A+2)*B
-(A+2)*-B        ;same value as above
<A+2>*B          ;another way  (The symbol A must
                 ;be defined and available.  See
                 ;Atomic Statements, section 2.3.5.1, page 20)
=60*=60
"A"-40
[0]-1            ;even literals can appear in expressions
FOO:BAR←1 105    ;the value of this expression is 105
                 ;(labels and assignment statements have no
                 ;effect or the value of the expression)
```

2.3 Atoms

An *atom* is the most basic syntactic element. An atom is either a *symbol* or a *constant*. There are also *complex atoms* which are not really atoms at all, but which can be used in the same way as atoms in forming expressions. Every atom represents a value.

2.3.1 Identifiers

Identifiers are very basic syntactic elements. They have many different uses, all of which involve referring to something by a convenient symbolic name. The uses of identifiers will be covered as the various applications arise. Identifiers may be defined either by the programmer or by FAIL.

The characters legal in an identifier are letters, digits, and the four characters dollar sign ($), percent sign (%), point (.), and underbar (_). An identifier is any non-null string of characters from this set, delimited by characters not from this set, except that the first character of an identifier must not be a digit. Only the first six characters of an identifier are significant, and upper and lower case letters are treated as equivalent. Thus "FOOBAR" and "foobarbletch" are equivalent identifiers. Also, "_" is considered equivalent to ".", so, for example, "A_7" and "A.7" are equivalent identifiers.

Certain identifiers have special meaning in FAIL, and cannot be used except with their own special meanings. Some of these *reserved identifiers* are IFAVL, IFDEF, IFDIF, IFE, IFG, IFGE, IFIDN, IFL, IFLE, IFMAC, IFN, IFNAVL, IFNDEF, IFNMAC, IFNOP, IFOP, IOWD, .FNAM1, .FNAM2, ".", and "$.".

2.3.2 Values

Most of the normal assembly process consists of translating text strings into their corresponding binary *values*. The main transformation happens when the atomic elements are converted to their binary representations; these are combined by binary operations into more complex constructs.

Often the final 36-bit value of an atom depends upon information not available at the time the atom is seen. This value may become known when a later part of the program is assembled, or it may not be known until the program is actually loaded. Consequently, up until the final loading of a program into a core image, its representation must be a slightly expanded form of simple binary so that the steps necessary to complete the calculation of all binary values can be adequately described. Partially defined values are commonly used in writing FAIL programs; several mechanisms exist to enable FAIL (and the loader) to handle such values correctly. The full impact of forward references and relocatable values is discussed in appendix B, page 50.

Some of the different kinds of values that often occur in FAIL are distinguished by particular names: *relocatable*, *absolute*, *defined*, *undefined*, *available*, and *unavailable*. The definitions that follow involve symbols and block structure to some extent. Refer to section 2.3.4, page 11, and section 2.3.4.6, page 15, for further elucidation.

A value that depends on where the program is when it is loaded in core is called *relocatable*. Relocatable values occur most frequently when some location in the program or in the data is referred to. Values that do not depend on where the program is located are called *absolute* or *unrelocatable*. An example of an absolute value is a constant. Another example of an unrelocatable value is the length of a table (that is, the difference between two relocatable values).

A *symbol* is an identifier that has a value. A symbol is *defined* when a value is assigned to it. A symbol can be referenced before it is defined, that is, when the value of the symbol is *undefined*. FAIL makes sure that the right thing happens when the value becomes defined as long as an undefined value is legal in the particular context where it is used.

A symbol that is defined is said to be *available* (after the point of definition) in the block where it is defined. When another (lower) block is entered, such a symbol becomes *unavailable* unless the programmer has taken steps to force the availability of that symbol in lower blocks.

2.3.3 Constants

Constants are the simplest forms of atoms; their values do not depend on context or previous operations (with the exception of the radix for interpretation of numbers). Constants are absolute, i.e., independent of where the program is loaded. A constant may be one of several types of numerical or text constants. In addition to the atomic constants described here, there are various data entry pseudo-ops described in section 3.3, page 30.

2.3.3.1 Simple Numbers

A *simple number* consists of a string of digits, optionally followed by the letter "B" and one or two additional digits which represent a scale factor. The digit string is interpreted as a number in the current radix. Since the radix is initialized to 8, simple numbers are usually interpreted as *octal* by default. In this case, the accumulation is done by logical shifting, so the number is considered unsigned. If the radix is anything other than 8, the accumulation is done by multiplication, and the sign bit cannot be set (but a negative number can be entered as an expression). The current radix can be set with the RADIX pseudo-op (see section 3.5.4, page 35).

The one- or two-digit argument following the "B", interpreted in decimal, specifies the low-order bit position of the number in the word. The number is shifted left logically a number of bit positions equal to 35 (decimal) minus the argument.

Examples:

```
1743
2
254B8
1B33     ;equivalent to 4
22B18
10B37    ;equivalent to 2
```

2.3.3.2 Decimal Numbers

Decimal numbers provide a way of entering decimal information regardless of the current radix. A decimal number is a simple number preceded by an equal sign (=). Since decimal numbers are handled identically to simple numbers except for the radix, the "B" shifting operation may also be used with decimal numbers.

Examples:

```
=100
=69
=10B27
```

2.3.3.3 Floating-Point Numbers

Numbers may also be entered in standard floating-point notation, in which case they will be converted to PDP-10 single-precision floating-point format. Floating-point numbers are always interpreted in decimal regardless of the current radix. Note that any arithmetic performed by FAIL on numbers is always integer arithmetic, even if the operands are floating-point numbers.

A *floating-point number* consists of two strings of digits, separated by a decimal point and followed by an optional scale factor. The digit strings before and after the decimal point represent the integer and fraction parts of the floating-point number, respectively. The scale factor is the letter "E", an optional minus sign, and one or two digits. The number following the "E" specifies a power of ten by which the number will be multiplied.

Although the fraction part of the number may be omitted, it is probably better to include the redundant 0 to avoid a possible future conflict that could arise if FAIL were modified to allow a decimal point following a digit string to signify a decimal number.

Examples:

```
10.7E1          ;equivalent to 107.0
9.973
0.13
10.             ;better to write this as 10.0
1.86E05
31.4159E-1
69E1            ;presently equivalent to 690.0
```

2.3.3.4 Ascii Constants

Constants may also be specified as the ascii value of a character or string of characters. The ascii value of a character is its 7-bit code in the Stanford Character Set, a modified form of the USASCII code (see appendix D, page 56). An *ascii constant* is written as a string of characters not containing a double quote ("), enclosed with double quotes, e.g. "Foo". If the string is null, i.e., "", the resulting value will be zero. If the string contains exactly one character the resulting value will be the ascii value of that character. If the string contains more than one character, each additional character will shift the total left 7 bits and add its own value, much as an octal number is accumulated. This results in packing characters into right-justified 7-bit bytes. Only the low-order 36 bits of the total are used, so if more than 5 characters appear in the string, only the last 5 characters and the low-order bit of the sixth-from-last character will affect the value.

This right-justified form is not the standard way of packing text for addressing with byte instructions, but is intended mainly for small immediate operands, etc. Text pseudo-ops (described in section 3.3.6, page 32) are used to store text in the usual left-justified format in multiple words.

Examples:

```
"A"             ;101 octal
"↑C"            ;27503
"foobar"        ;337576130362
```

2.3.3.5 Sixbit Constants

Another character code that is frequently used is sixbit. It is a modified version of ascii code which uses only 6, instead of 7, bits in order to pack 6 characters into a word rather than 5.

The basic ascii to sixbit transformation consists of subtracting 40 (octal) from the ascii code, which maps ascii 40-137 (all the printing characters of 64-character ASCII) into the desired 0-77. Since the 140-177 range consists mostly of lower-case versions of the 100-137 characters, a better transformation also maps this range to 40-77. The method used by FAIL is to copy the 100 bit into the 40 bit and set the 100 bit to 0. The inverse transformation is accomplished by adding 40 to each sixbit character.

Sixbit constants can be specified in FAIL in the same way as ascii constants, except that close single quotes (apostrophes) (') should be used instead of double quotes. Naturally, if more than one character appears in the string, the shifting will be 6 bits at a time instead of 7, and the last 6 characters of the string will always be completely significant. Again, a pseudo-op is available (see section 3.3.6, page 32) to pack longer strings into multiple words.

Examples:

```
'a'             ;41
'DSK   '        ;446353000000
'gronker'       ;625756534562
```

2.3.4 Symbols

Symbols are one of the most important features provided by an assembler. One capability provided by symbols is the ability to abbreviate a complex expression with a single identifier.

Another is to represent an assembly parameter, so that its value can be changed at the symbol definition only, without having to modify the places where the parameter is used. A third use is to represent values which are difficult for the programmer to calculate, such as values dependent upon exactly where certain parts of the program are stored.

A *symbol* is an identifier which at some point in the program (or possibly in an external program) is assigned a value which will be associated with that identifier whenever it is used in a context where symbols are recognized (see section 2.1.1.1, page 3, and section 4.1.4, page 40, for discussion of possible conflicts with opcodes or macros). The point at which a value is assigned to a symbol is said to be the point where it is *defined*.

In most circumstances, a symbol may be used to stand for a value either before or after it is defined. A symbol is said to be *referenced* when it is used to stand for a value. If this reference occurs earlier in the source file(s) than the definition, it is said to be a *forward reference*; if the reference follows the definition, it is said to be a *backward reference*. Backward references can be handled fairly easily, by merely replacing the symbol by its known value. However, forward references create some complication since FAIL does not know the value of the symbol until later in the file.

Two-pass assemblers avoid the forward reference problem by assembling the program twice. On the first pass the assembler calculates the value for each symbol; on the second pass these known values are used when the corresponding symbols are referenced. This method probably has the smaller storage requirements, but it requires more cpu time since the entire source file is scanned twice.

FAIL uses the one-pass approach to save execution time (at the expense of increasing the storage requirements). In this method, each forward reference assembles an incomplete word, but sufficient information is included in the binary file to enable the loader to complete the assembly. Part of the necessary mechanism exists in the loader anyway in order to handle externally defined symbols, which must be treated as forward references even by a two-pass assembler. Information placed in the binary file to update the value of an incompletely assembled word is referred to as a *fixup*.

Because of the problem of forward references in a one-pass assembler, the meaning of "defined" as used in this manual is not "defined somewhere within the program", but rather "defined in the program before the place being considered". In this sense a symbol is not considered to be "defined" at the time of a forward reference, even if it is defined later in the program.

A symbol may be defined in one of four ways. It may be defined as a label, as a parameter, or as a variable, or it may be a predefined symbol. These types of symbols are discussed in the following subsections.

2.3.4.1 Labels

Labels are the most common type of symbol. They are used as symbolic references to locations in the program. Labels help to keep such references independent of the exact placement of those parts of the program in the core image. The value of a label is calculated automatically by FAIL, so that the programmer need not keep careful account of the exact numeric locations of all parts of his program.

A *label* is defined by simply writing an identifier followed by a colon (:) at the beginning of any expression being scanned. This will normally define the symbol as equal to the *location counter*, i.e., the location where the next word will be assembled. However, in some circumstances involving the use of literals (section 2.3.5.2, page 20) or the PHASE pseudo-op (section 3.1.3, page 23), the value of the label may differ from the location counter. The value assigned to a label is usually relocatable because the location counter is initialized to relocatable zero, but it may be absolute.

Although labels may occur at the beginning of any expression, they almost always occur at the beginning of a line. This convention improves the readability of programs by keeping labels in a place where they are easily recognized.

In order to detect possible conflicts in label usage, FAIL does not allow any label to be defined more than once. (However, FAIL block structure allows a label to be redefined in different blocks; see section 2.3.4.6, page 15.) Once a symbol has been defined as a label, it cannot be redefined; a symbol cannot be defined as a label if it has any previous definition. An attempt to do either of these things will result in a *multiple definition* error message, and the new definition will not take effect.

Examples:

```
LOOP:   JRST LOOP       ;points to itself
FOO:                    ;labels the location of the next instruction
```

2.3.4.2 Parameters (Assignment Statements)

A *parameter* is a symbol that is given an arbitrary 38-bit value by an assignment statement. Actually, the final value is 36 bits, but since either 18-bit halfword may be relocatable two more bits are included in the representation of the value. The basic format of an *assignment statement* is an identifier followed by a left-arrow (←) followed by an expression. The 38-bit value of the expression, which must be defined, will be given to the specified symbol. An equal sign (=) may also be used as an alternative to left-arrow to allow partial compatibility with other assemblers, but if the first atom after the = begins with another = to indicate a decimal number, at least one space should separate the two to distinguish them from ==, which has a different function (see section 2.3.4.5, page 15).

As with labels, any number of assignment statements may appear at the beginning of any expression, but they are normally written as separate statements for improved readability. In its full generality, an assignment statement may define more than one symbol by beginning with several symbol names, each followed by a left-arrow, and finally followed by the expression, whose value will be given to all symbols mentioned.

Unlike labels, parameters may be redefined as often as desired. Once a parameter has been defined, each reference to it will use the value in effect at the time of that reference (i.e., as of the last assignment). The value appearing in the symbol table in the binary output file will be the last value assigned. The value used for forward references (i.e., before the first definition) will be that of the first assignment. Note that this is an incompatibility with two-pass assemblers, which would instead use the last value assigned during pass one.

Examples:

```
FOO←105
BAR←=69
BLETCH←BARF←LOSS←FOO+BAR*3
garp= =97        ;note space between = is necessary
```

2.3.4.3 Variables

Variables are symbols whose values are the addresses of cells automatically allocated by FAIL for data storage. A variable is usually created by immediately following a symbol reference with a number sign (#). The symbol, which must not be previously defined, is declared to be a variable and will have its location assigned when the location of the variables area is known (see section 3.1.7, page 25). The symbol is not defined at this point; it cannot be used in contexts which do not allow forward references. However, it can be used as any other forward-referenced symbol; the number sign need not be used with more than one occurrence of the symbol. Similar effects can also be obtained with the INTEGER and ARRAY pseudo-ops (see section 3.4.2, page 33).

Examples:

```
SETZM FOO#
MOVEI A,BAR#-1
```

2.3.4.4 Predefined Symbols

Predefined symbols are available for use in *all* circumstances where symbols are recognized.

Two predefined symbols, point (.) and dollar-point ($.) refer to the *location counter*, which is the location where the next complete word will be stored. In the absence of special circumstances, "." and "$." have the same value; "." is the one usually used. These values are usually relocatable but may be absolute; see section 3.1.1, page 22.

The reason for having two of these symbols is that some features of FAIL create complications affecting the location counter; see the discussion of literals (section 2.3.5.2, page 20) and the PHASE pseudo-op (section 3.1.3, page 23).

Examples:

```
JRST .-1
JUMPN T,$.+3
```

The predefined symbols, .FNAM1 and .FNAM2 refer to the name of the current source file. The value of .FNAM1 is the 36-bit binary representation of the source file name; .FNAM2 has the value of the source file extension (or second file name).

2.3.4.5 Half-Killed Symbols

Symbols are included in the binary output file to aid debugging and to allow the loader to link several programs together. The debuggers (RAID and DDT) have symbolic disassemblers which take binary words and interpret their fields to display mnemonic opcodes, addresses, accumulator names, etc. Sometimes, the user wants to prevent particular symbol names from being displayed by the symbolic disassembler. Symbols that have been marked to prevent their display are called *half-killed*. Half-killing a symbol is useful for parameters which might incorrectly be displayed as core addresses or accumulator names. Half-killing is also handy for labels in code that is relocated at runtime. The debuggers do recognize half-killed symbols when they are input.

FAIL treats half-killed symbols precisely the same as other symbols, except, when the symbol is written in the binary output file, a bit is set to inform the debugger that the symbol is half-killed.

In FAIL, half-killing a symbol is accomplished by doubling the defining character (e.g., ::, ←←, or ==). In the case of ==, the two equal signs must not be separated by any spaces, because this is how the ambiguity is resolved with respect to the other use of equal sign to indicate decimal numbers. A parameter will be half-killed if any one of its definitions specifies half-killing.

Examples:

```
ERRFLG←←100            ;the usual way of writing it
IOFLG ← ← 2000         ;this can have spaces anywhere
BUFSIZ == 100          ;but this can't (100 is octal)
BUFSIZ = = 100         ;since this means decimal, not half-killed
BUFSIZ === 100         ;this is unambiguous (100 is decimal)
BUFSIZ == = 100        ;(100 is decimal)
LOOP::  SKIPN A,(B)    ;a half-killed label
```

2.3.4.6 Block Structure

Block structure is very basic to the usage of symbols. This section may be skipped if the reader does not plan to use block structure. The one thing to remember is that in the absence of block structure any symbol which is *defined* is also *available*.

FAIL *block structure* provides a way of localizing the usage of symbols to particular parts of the program, called *blocks*. Block structure allows the same symbol name to be given different meanings in different blocks. The block structure used in FAIL is similar to that of ALGOL, but is somewhat less restrictive.

A program is considered to be a block whose name is the same as the program name (set by the TITLE statement; see section 3.5.1, page 34). Each block may contain any number of inner blocks, but the depth of nesting may not exceed 17 (decimal). A definition of a symbol, a user-defined opcode (see section 3.2.1, page 26), or a macro (see section 4, page 38) applies only within the scope of the outermost block in which it is defined. The scope of a block includes the scope of each block it contains, unless the symbol (etc.) in question is defined again in an inner block, in which case the more local definition takes precedence within the scope of that block. A block is delimited by a BEGIN statement and a BEND statement (see section 3.2.2, page 26).

Features exist in FAIL for controlling the block level of symbols. If a symbol, when defined as a label or parameter, is preceded by an up-arrow (↑), it will be treated as if it were defined in the next-outer block. If a double up-arrow (↑↑) is used, the symbol will be treated as though it were defined in the outermost block of the program. These features are most commonly used for such things as making subroutine entry points available to outer blocks when the subroutines themselves are contained in blocks. In simple cases, this could be done by beginning the block after the entry label(s) or even after some of the code, but this makes reading the routine more difficult and hence the up-arrow construct is preferred. Tilde (~) may be used instead of up-arrow.

Here are some examples of symbol usage, with and without block structure. Both examples generate the same code:

```
F001:    JRST F001          F001:    JRST F001
         JRST F002                   JRST F002
         JRST F003                   JRST F003
         JRST F005                   JRST F005
BEGIN
F002:    JRST F001          F0022:   JRST F001
↑F003:   JRST F002          F003:    JRST F0022
         JRST F003                   JRST F003
BEGIN
         JRST F001                   JRST F0013
↑↑F005:  JRST F002          F005:    JRST F0022
         JRST F003                   JRST F003
F001:    JRST F004          F0013:   JRST F004
BEND
↑F004:   JRST F004          F004:    JRST F004
BEND
F002:    JRST F004          F002:    JRST F004
```

A complication arises with FAIL block strurcture due to the absence of the ALGOL requirement that all identifiers be declared at block entry time. FAIL allows forward references, yet does not require any declaration of symbols other than their defining occurrences. Hence, FAIL cannot decide whether to use an existing outer-block version of a symbol or to make a forward reference to a more local definition that may occur later.

To resolve this ambiguity, FAIL always considers a symbol reference to be a forward reference when the symbol has not been defined in the current block, even if it has been defined in some outer block. If no other definition is given by the time the block ends, then the outer-block definition is used to resolve the forward reference. While in the inner block in this situation, the outer-block symbol is still said to be *defined*, but it is also said to be *unavailable*. Thus block structure forces many references to be forward references, even when they would not otherwise be such.

Macros and user-defined opcodes cannot be forward-referenced. Such symbols are always available; references to them will use their outer-block definitions.

Examples:

```
FOO:    MOVSI 1,-62     ;FOO is defined as a label
BAR:    CAME 2,ZOT(1)   ;so is BAR
        AOBJN 1,BAR     ;BAR is referenced
BEGIN                   ;FOO and BAR are defined, but now unavailable
LOSS:   MOVEI 1,0       ;LOSS is defined
        JRST LOSS       ;a backward reference to LOSS
        JRST FOO        ;this is treated as a forward reference
FOO:    HRRM 6,LOSS     ;so it can reference this definition
BAZ:    JRST BAR        ;this is treated as a forward reference
        JRST FOO        ;this refers to this block's FOO
BEND                    ;The outer-block definition of BAR becomes
                        ;available at this BEND.  A fixup is emitted
                        ;to fix the reference to BAR at BAZ
```

Many contexts do not accept forward references (e.g., accumulator and index fields). In these contexts unavailable symbols cannot be used, even if they are defined. Therefore, FAIL provides two mechanisms for forcing defined symbols to be *available* to lower blocks. One is the down-arrow mechanism, which is used at the defining occurrence of the symbol, and the other is the GLOBAL pseudo-op (see section 3.2.3, page 26), which is used in the referencing block.

The down-arrow mechanism is the more commonly used method, since this problem is most often associated with particular symbols (accumulator names, assembly parameters, etc.). Preceding the symbol name in a label or assignment statement with a down-arrow (↓) causes that symbol to remain available whenever inner blocks are entered. Usually it is dangerous to redefine such symbols locally, since any forward references will have incorrectly referred to the outer-block definition. Consequently a warning message is printed in this case, but if no forward references are made to the local version, it will assemble correctly. However, if the redefinition of a down-arrowed parameter is effective at its original block level (possibly via ↑ or ↑↑), FAIL will change the original definition without complaint. This allows redefinition of global parameters from inner blocks. A question mark (?) may be used instead of down-arrow.

Examples:

```
↓A←1                    ;some accumulator (AC) definitions
B←2
↓FOO←←=69               ;and a parameter
BEGIN
        ADD B,A         ;this is illegal because AC
                        ;symbols must be available
        MOVE A,B        ;but this is legal since A is available
                        ;by ↓
A←5                     ;this will produce a message and is too
                        ;late to affect the instruction above
B←6                     ;this is legal and will fix up the MOVE
                        ;to be MOVE 1,6
        MOVE A,B        ;whereas this will be MOVE 5,6
↑FOO←←105               ;this is legal since it is "aimed" at the
                        ;FOO in the outer block
BEND
```

There are further details in section 3.2.2, page 26, and section 3.2.3, page 26, about the block structure pseudo-ops BEGIN, BEND, and GLOBAL.

2.3.4.7 Linkage with Separately Assembled Programs

It is sometimes desirable to have a program which is assembled in several parts, either to save reassembling the entire program for each change or because the program is written in a mixture of languages. Even with a single assembly it is usually necessary to use some of the job data area symbols, and sometimes symbols from the debuggers (RAID or DDT), all of which are reached through the linking loader. In this context, the word *program* refers to the result of one assembly or compilation, and thus a core image may contain several programs.

To allow reasonable communication between these programs, the loader allows symbol definitions to be passed between programs. For this purpose, symbols are divided into two classes, local symbols and global symbols. (There is no relation between the GLOBAL pseudo-op and the global symbols discussed here.)

Symbols are normally considered *local*, which means that they will not be available outside their own program and may be defined in more than one program without conflict. *Global symbols*, however, are available to all programs and hence must not have conflicting definitions within the set of programs to be loaded. The easiest way to declare a symbol to be global is to follow some occurrence of the symbol by an up arrow. This flags the symbol as a global without specifying whether it is defined in this program or another program, since FAIL will have figured that out by the end of the assembly. Undefined globals (*external symbols*) will have appropriate fixup information passed to the loader for resolution when the defining programs are loaded. Globals may also be declared with the EXTERNAL (section 3.2.5, page 27) and INTERNAL (section 3.2.4, page 27) pseudo-ops.

Declaring a symbol global forces its scope to the outermost block in the same way as does a double up-arrow. Therefore, if a symbol is defined and declared global in an inner block, there must not be a conflicting definition in an outer block.

One other related feature is the library mechanism. A *library* is a file that contains a set of utility programs. Each program in the library may be loaded independent of the others, depending on whether it is required by the programs that have been loaded thus far. To implement this, there is associated with each program in the library (in one or more *entry blocks*) a list of certain global *entry points* defined in that program. In most cases these are the names of the routines contained in the program. When the loader is in *library search mode*, it loads only those programs for which at least one of the entry points corresponds to an existing unsatisfied global request (external symbol). Only those programs actually needed are loaded from the library; the rest are ignored. The ENTRY pseudo-op (section 3.2.4, page 27) is used to declare symbols to be entry points which will be available to a library search.

2.3.4.8 Symbols and Arrows

This is a brief restatement of the ways that identifiers are used as symbols in conjunction with arrows.

Examples:

```
↑SYM:              ;SYM is available at the next-outer block
↑↑BOL←←10          ;BOL is half killed and available at the
                   ;outermost block
↑↑ZOT= =69         ;ZOT is available at the outermost block
↓A↑←7              ;A is global and available to lower blocks
FOO↑:              ;FOO is global and defined here (internal),
                   ;available at the outermost block
PUSHJ P,BAZ↑       ;BAZ is global, may be external, available at the
                   ;outermost block
```

2.3.5 Complex Atoms

Two constructs exist which assemble one or more statements in much the same way as FAIL's normal top-level statement processor, but then return as an atom the value associated with the statement(s) assembled, rather than outputting the binary data. Both of these constructs involve the use of opening and closing characters to delimit the text. For an atomic statement, broken brackets, called *brokets* (< and >), are the delimiters. For literals, the delimiters are square brackets ([and]).

When the opening character is recognized, FAIL saves its present state and enters an auxiliary statement-assembly loop, continuing to assemble statements until a statement is encountered which terminates with the closing character. The closing character is located as a statement delimiter, not by keeping a count of the opening and closing characters. Thus if the delimiter character appears in a text constant, it will not be counted toward the match; also, attempting to use a comment (see section 2.1.6, page 6) in the final statement of the sequence will prevent recognition of the closing delimiter. Note that this method of counting brokets is different from the macro processor, which counts brokets rigidly, independent of context. Nesting of complex atoms is handled by the recursive nature of FAIL's statement processor.

2.3.5.1 Atomic Statements

When it is useful to have the value of an entire statement treated as an atom, enclose that statement in brokets. Some number of statements will be assembled as described above, and the value of the first word assembled will be returned as the value of the atom, just as if the corresponding number had been typed. The values of any additional words assembled up to the closing broket will be ignored, although their side effects (if certain pseudo-ops are used) may remain. For example, if one of the multiple-word text pseudo-ops is used inside brokets, only the first text word will be returned, and the rest will be dispatched to the great bit bucket in the sky. This type of atom is constrained by FAIL to be handled as a number, so all symbols used in this context must be defined and available.

Examples:

```
<JRST>          ;equivalent to 254000000000
<JRST 105       ;254000000105 will be the value
JRST BAR        ;and this statement won't do anything except
                ;possibly produce an error message if BAR
                ;isn't defined and available         >
         >      ;this broket will end it, not the one above
```

2.3.5.2 Literals

Although the PDP-10 instruction set allows a large percentage of constants to be specified as immediate operands, it is still frequently necessary to reference constants stored elsewhere in memory. Instead of explicitly setting up these constants and referencing them by labels, it is possible to reference these constants as *literals*. The basic function of literals is to allow the programmer to write the value of the desired constant directly (i.e., literally), while the assembler automatically allocates a memory location for it, stores the value in it, and supplies the address of the cell for the reference. Also, an operation called *constants optimization* occurs, which consists of comparing (the binary value of) each literal with previous literals to see if the required constant has already been allocated, in which case the existing cell will be used rather than allocating another. This avoids multiple copies of a given constant.

To use a literal, put a statement of the desired value in square brackets and use it as an atom. The value (of the literal) will be the address of the literal in memory, which is treated like an undefined symbol since the actual location will not be assigned until later (usually the end of the program; also, see section 3.1.6, page 25). Literals can be used *only* where forward references are legal.

Because of the constants optimization, it is often dangerous (and considered poor form) to write a program which changes the contents of a literal. Such a change affects all parts of the program attempting to use that constant, which is not usually the desired effect.

A literal may contain more than one word if desired. The syntax of literals is basically the same as that of atomic statements, except that *all* words assembled are used. Multiple-word literals are most commonly used to store long text strings, but may be used to store sequences of instructions. There is no rigid limit on the maximum size of a literal, but large literals do consume assembler core fairly rapidly.

For purposes of assembling code in literals, it should be noted that the predefined symbol "." retains its value during the assembly of a literal, rather than referring to the current location within the literal. Thus it refers to the location where the reference to the outermost literal is being made. The current location within the (current) literal can be referred to by using the symbol "$." (but this may not do the right thing if the PHASE pseudo-op is in use).

Naturally, labels may appear inside literals, but if they do they will be assigned the value of the current location within the literal, rather than the value outside. (Labels that appear inside literals are called *literal-labels*.) This is the only time that FOO: and FOO←. assign different values to FOO. The location of a literal is unknown at the time it is processed; hence, labels that are defined within literals (and "$." when used inside literals) are undefined symbols. For example, it is illegal to say FOO←$. inside a literal because assignment statements do not accept undefined values. Note also that constants optimization will still occur with labeled literals, and this may result in several labels having the same value, if appropriate.

Examples:

```
PUSH P,[5]                         ;no PUSHI, so a literal is handy
OUTSTR [ASCIZ /FOOBAR/]            ;a two-word text constant
JRST [  MOVEI C,12                 ;some code in a literal
        PUSHJ P,WRCH
        SUB P,[1,,1]               ;a nested literal
        JRST .+1]                  ;returns to the next instruction
                                   ;outside the literal

PUSHJ P,[YTST:  CAIE C,"Y"         ;a subroutine in a literal
                CAIN C,"y"         ;(very rarely done actually)
                AOS (P)
                POPJ P,]

PUSHJ P,YTST                       ;calling the above subroutine
```

3. Pseudo-Ops

Most statements are translated into operations for the computer to perform when the program is executed. *Pseudo-ops* (short for *pseudo-operations*), on the other hand, signify operations to be performed at assembly time. Some of these operations affect the behavior of the assembler in particular ways; others serve as convenient methods of entering data in commonly used formats.

3.1 Destination of Assembled Code

The assembler uses a *location counter* to keep track of the location where the code it is assembling will go. This counter is initialized to relocatable 0 at the start of the assembly; it is incremented by 1 for each instruction assembled. The value in the location counter is the location where the next word assembled will go.

3.1.1 LOC, RELOC, and ORG

The contents of the location counter can be changed with the LOC, RELOC, and ORG statements.

The LOC pseudo-op takes one argument, an expression, which must be defined and available. The effect of LOC is to put the value of the expression into the location counter and to set the relocation of the counter to absolute, regardless of the relocation of the argument.

The RELOC statement has the same effect as the LOC statement except that the relocation is set to relocatable, regardless of the relocation of the argument.

The ORG statement has the same effect as the LOC and RELOC statements except that the relocation is set to the relocation of the argument.

Whenever LOC, RELOC or ORG is used, the current value (and relocation) of the location counter is saved (there is only one such saved location counter, *not* one for each pseudo-op). A LOC, RELOC, or ORG statement with no argument will cause the saved value and relocation to be swapped with the current value (and relocation) of the location counter.

3.1.2 SET and USE

It is possible to have *multiple location counters* and to switch back and forth among them. Only the currently active location counter is incremented. Location counters may be given any names which fit the syntax of identifiers. There is no relationship between location counters and labels with the same name.

The SET pseudo-op is used to initialize a location counter. It takes two arguments separated by a comma. The first is the name of the location counter; the second is the value to which the counter will be set. SET has the same effect as ORG except that it changes the indicated location counter and has no effect on the current location counter unless it is the same as the indicated one. SET is usually used to create a new location counter.

The USE pseudo-op is used to change location counters. It takes one argument, the name of the location counter to change to. USE causes the current location counter value to be saved away and the value of the indicated counter to be used. If a subsequent USE indicates the location counter which was saved away, the value it had when it was saved away will become the current value. If the indicated location counter has not appeared in a SET before its appearance in a USE (i.e., if it has no value), it will be given the value of the current location counter. The location counter which the assembler starts with has a blank name (i.e., a null argument indicates this first one).

In the example below, a close single quote (apostrophe) (') is used to denote that the value it follows is *relocatable*. This is the convention that FAIL uses when making a listing of the assembled code.

Example:

```
        Location          Instructions

           0'             JRST FOO
           1'             JRST BAZ
           2'             SET GARP,37
           2'             USE GARP
          37              JRST FOO
          40              USE
           2'             JRST FOO
```

3.1.3 PHASE and DEPHASE

It is sometimes desired to assemble code in one place which will later be moved by the program itself to another place. In this case, it is desired that labels be defined as referring to locations in the place where the code will be moved, rather than where the assembler will put it. To accomplish this, the PHASE pseudo-op is used. PHASE has one argument, the location to which the next word assembled will be moved by the program. For instance, if, while the location counter is at 74, a PHASE 32 appears and a label appears on the next line, the label will be given the value 32, but the code on that line will be placed in location 74. Under these circumstances, the "." symbol will have the value 32, but the "$." symbol will have the value 74. The PHASE pseudo-op remains in effect until cancelled by a DEPHASE pseudo-op (no argument).

If a RELOC, LOC, or ORG pseudo-op (see section 3.1.1, page 22) appears while PHASE is in effect, the following considerations apply. If the relocation of the location counter remains unchanged by the RELOC (or LOC or ORG), then the value of the phase will be offset by the same amount as the location counter changes. That is, the value of "." and "$." will be changed by the same amount. If the relocation of the location counter changes and the relocation of the phase was the same as the relocation of the (old) location counter, then the relocation of the phase will be changed and the phase will be offset by the same amount as the location counter changes. Otherwise, the error message *Indeterminate Phase due to RELOC* will occur and FAIL will dephase.

3.1.4 HISEG

This statement outputs information directing the loader to load the program into the high segment. It should appear before any code is assembled.

3.1.5 TWOSEG

This statement directs FAIL and the loader to assemble and load a two-segment program. This complicates the relocation process because the loader must maintain two relocation constants, one for each segment. Since only one bit of relocation information is available for each value in the relocatable binary file, a kludge is used to decide which relocation to apply to each relocatable value. To do this, the loader compares the unrelocated value to a quantity known as the *high-segment origin*, which is the first address used for the high segment within that program. Any value greater than or equal to this quantity will be considered a high-segment address, while any value less than this quantity will be considered a low-segment address. When the value in question is a location specifier, the choice of relocation will determine which segment the code is actually loaded into.

Unfortunately, there is a possible bug in this relocation method. It is possible to have an expression which evaluates, through normal relocation arithmetic, to a relocatable quantity whose unrelocated value does not correspond to the segment the relocation was originally derived from. For example, if FOO is a label at high segment location 120, it will probably have a value of relocatable 400120. The expression FOO-400000 would be calculated by FAIL to have the value relocatable 120. This value would be passed directly to the loader since Polish appears unnecessary. However, the loader would apply the low-segment relocation to this value and probably have incorrect results. At present, the best way to get around this is to say FOO*1-400000, which will force the Polish to be passed to the loader.

The high-segment origin is specified by an optional argument to the TWOSEG pseudo-op, or set to the default of 400000 in its absence. In this case a RELOC 400000 followed by a RELOC 0 will initialize the dual location counter to assemble into the low segment and to switch segments whenever a RELOC statement with no argument is encountered (see section 3.1.1, page 22). Like HISEG, TWOSEG should be used before any code is assembled.

Example:

```
        TITLE EXAMPLE
PDLEN←←100
P←17
        TWOSEG 400000   ;initialize to two segments
        RELOC 0         ;initialize dual location counters
        RELOC 400000    ;now assemble code in the high segment
START:  TDZA 1,1
        MOVNI 1,1
        MOVEM 1,RPGSW#
        CALLI 0
        MOVE P,[IOWD PDLEN,PDLIST]
        RELOC           ;set the relocation to low segment
PDLIST: BLOCK PDLEN     ;define space for data storage
        RELOC           ;set location counter to the high segment
        PUSHJ P,CORINI  ;code is assembled in the high segment
                        ;the rest of the program goes here
        RELOC           ;back to the low segment
        VAR             ;do variables in the low segment
        RELOC           ;to the high segment
        LIT             ;and literals here
        END START
```

3.1.6 LIT

The LIT statement causes all previously defined literals to be placed where the LIT statement occurs. The LIT statement must not appear inside a literal. If a two segment sharable program is being assembled, LIT should appear in the upper segment.

3.1.7 VAR

The VAR statement causes all variables which appeared with a # in this block (or a sub-block of this one) to be placed where the VAR appears. VAR must not appear inside a literal. If a two segment sharable program is being assembled, VAR should appear in the lower segment.

3.2 Symbol Modifiers

The pseudo-ops in this section perform several functions, all relating to the definition or availability of symbols, or affecting the linkage of this program to others.

3.2.1 OPDEF

The OPDEF statement has the following form:

```
OPDEF symbol [value]
```

OPDEF inserts the symbol into FAIL's opcode table with the indicated value. The symbol, which is a *user-defined opcode*, may then be used as any other opcode. The value part of the OPDEF must be defined and available. User-defined opcodes are sometimes called *opdefs* because of the pseudo-op by which they are defined.

3.2.2 BEGIN and BEND

The BEGIN statement is used to start a block. The block it starts will end at the corresponding BEND statement. The BEGIN may be followed by an identifier that will be used as the name of that block. DDT and RAID recognize block names. If no identifier appears, the assembler will create one of the form A.000, where the 000 will be replaced by the block number of this block in octal. (The *block number* is initialized to zero and incremented for each BEGIN.) There is no relationship between labels and blocks with the same name. All text following the identifier is ignored until the next line feed or double-arrow.

BEND may be followed by an identifier which, if present, is compared to the block name of the block being ended; if they don't match, FAIL prints an error message.

FAIL does not require block names to be unique; however, the loader and the debuggers sometimes depend on unique block names, so the user would be wise to avoid conflicts.

For a discussion of block structure, see section 2.3.4.6, page 15.

3.2.3 GLOBAL

The GLOBAL pseudo-op should be followed by a list of symbols separated by commas. Each symbol should be defined in an outer block. The effect of GLOBAL is to find the nearest outer block in which that symbol is defined and to make the definition in that block immediately available in the block in which the GLOBAL appears. GLOBAL does not affect the definition of the symbol in any intervening blocks.

If a symbol has been declared GLOBAL in a block and later is redefined in that block, the redefinition affects the definition in the outer block where GLOBAL found the original definition. Doing this causes strange effects if the definition was not in the next-outer block; it should not be done without some careful thought.

The GLOBAL pseudo-op has no relation to the concept of global symbols.

3.2.4 INTERNAL and ENTRY

These statements declare certain locally defined symbols to be internal symbols. *Internal symbols* are those which are made available to other programs by the loader. INTERNAL (or ENTRY) should be followed by a list of symbols separated by commas. These symbols need not be defined before the INTERNAL (or ENTRY) statement appears, but they must be defined by the end of the program.

ENTRY emits special library entry blocks to the loader; see section 2.3.4.7, page 19. ENTRY statements must appear before any other statements that emit code, except that it is specifically legal to precede ENTRY statements by a TITLE statement.

3.2.5 EXTERNAL

The EXTERNAL statement declares that certain symbols are external symbols. An *external symbol* is a symbol that is declared internal in some other program. EXTERNAL is followed by a list of symbols separated by commas. The loader will fix up any references to an external symbol when the program in which it is defined is loaded.

Symbols must not be defined at the time they are declared with an EXTERNAL statement. If an external symbol is subsequently defined, it is automatically converted to an internal symbol.

If any occurrence of a symbol is immediately followed by an up-arrow (↑), that symbol is made external if it is not yet defined, or internal if it is defined. If an external symbol is subsequently defined, it will be made internal.

3.2.6 LINK and LINKEND

LINK and LINKEND are used to establish a single-linked list among several separately assembled programs. Each linked list is identified by a link number in the range 1-20 (octal). The formats are

```
LINK number,location

LINKEND number,location
```

The number is the link number; the location is the address where the link information will be stored. The effect is to allow 20 lists to be threaded through several separately assembled programs.

The loader initializes each link (and linkend) to zero. LINK N,FOO causes the loader to store in FOO the current value of link N. Then link N is set to (point at) FOO. LINKEND N,BAZ causes the the loader to store the address BAZ as the linkend for link N. When the loader finishes loading all programs, the final value of each link will be stored in the corresponding linkend address, only if that address is non-zero. The LINKEND feature allows the head of the list to be in a known place, rather than in the last place LINKed.

3.2.7 .LOAD and .LIBRARY

The .LOAD pseudo-op causes the loader to load a specific REL file as a consequence of loading the program in which this pseudo-op occurs. The format is

```
.LOAD DEV:FILE[PRJ,PRG]
```

The DEV: field is optional (the default is DSK:); it specifies the device where the REL file can be found. The [PRJ,PRG] field is optional; it has the usual meaning. The file named must have the extension REL (this is a loader restriction).

In non-Stanford FAIL installations, the file name is scanned in accordance with the convention that prevails at that site.

The .LIBRARY pseudo-op is similar to .LOAD, except that instead of loading the file, the loader will search the named file as a library.

3.2.8 PURGE

The PURGE pseudo-op takes a list of symbols, separated by commas, as its argument. Each of the symbols named will be purged, i.e., removed from FAIL's symbol table. A purged symbol can be an opcode, macro, label or other symbol. For PURGE to be legal, the symbol must be defined and available when the PURGE occurs. Some symbols, such as variable names literal-labels, and global symbols, cannot be purged. Purged symbols are not passed to the loader or debugger.

PURGE searches the symbol table for opcodes first, then macro names, and finally labels (and parameters). This means that if a symbol has a definition as both an opcode and a label, purging that symbol will delete the opcode, and a second purge of that symbol will delete the label definition.

If the identifier name of some purged symbol is used after the purge, FAIL makes a new and totally different symbol, which has no relation to the purged symbol. The CREF program will also consider such a symbol to be different from the purged symbol.

Caution: if an opcode, pseudo-op, or other predefined symbol is purged, it will remain unavailable to subsequent assemblies performed by the FAIL core-image from which it was purged. Also, it is unwise to purge a macro while it is being expanded.

3.2.9 XPUNGE

XPUNGE is used to delete all local symbols from one block. XPUNGE takes effect *only* at the next BEND (or END or PRGEND) statement following the XPUNGE. At that BEND, most local symbols will *not* be emitted to the loader. This decreases the size of the REL file and makes loading it faster. Block names, internal and external symbols, variables, and literal-labels will be passed to the loader.

3.2.10 SUPPRESS and ASUPPRESS

When a parameter file (i.e., a file that contains assembly parameters for use in several assemblies) is used in assemblies, many symbols get defined but are never used. Unused defined symbols take up space in the binary file. Unused symbols may be removed from symbol tables by means of the SUPPRESS or ASUPPRESS pseudo-ops. These pseudo-ops control a *suppress bit* associated with each symbol; if the suppress bit is on and the symbol is not referenced, the symbol will not be output to the binary file.

SUPPRESS takes a list of symbols, separated by commas, as its argument. The suppress bit is turned on for each symbol named. A symbol may be an opdef, a parameter, or a label. The symbol should be defined before the SUPPRESS statement occurs.

ASUPPRESS turns on the suppress bit for every user-defined symbol and opcode that exists in the symbol table at the time the ASUPPRESS occurs.

Variables, literal-labels, internals, and entry point symbols are never suppressed. Externals that are not referenced can be suppressed.

If ASUPPRESS appears in a universal program (see section 3.2.11, page 29), then *all* symbols in the universal symbol table will have the suppress bit set when they are used in a subsequent SEARCH.

3.2.11 UNIVERSAL and SEARCH

The UNIVERSAL pseudo-op has the same syntax as TITLE (see section 3.5.1, page 34). In addition to the functions of TITLE,.UNIVERSAL declares the symbols defined in this program to be universal symbols *Universal symbols* are symbols which can be accessed by other programs that are assembled after the universal symbols have defined. That is, UNIVERSAL causes symbols to be retained by FAIL after it finishes assembling the universal file. When subsequent files are assembled (using this copy of FAIL, which has the universal symbols), the universal symbols can be accessed as any other local symbols. The program name set by UNIVERSAL is used to name the universal symbol table created to contain the universal symbols defined by this program. Only outer block symbols (and macros and opdefs) are retained in the universal symbol table. Variables, literal-labels, and internal symbols are *not* retained.

Universal files are intended for making definitions, not for assembling code. The usual use for a universal file is to define opcodes, macros and parameters for subsequent assemblies. It is not wise to include relocatable symbols in the universal file. The exception is that a universal file may declare a symbol to be external; that declaration can be used by subsequent assemblies that search this universal symbol table.

SEARCH controls access to the universal symbols. SEARCH takes a list of arguments, each of which is the name of a universal symbol table. For each universal table named, all the symbols in that table are added to *the end of* the main symbol (or macro or opcode) table. Then, when the symbol table is searched, if there is no other definition of the symbol, the universal definition will be found. Universal symbols are considered to be defined at the outer block. If such symbols are to be made available to inner blocks, they must be defined with a down-arrow, or declared GLOBAL.

3.3 Entering Data

3.3.1 DEC and OCT

The DEC and OCT statements both take a string of arguments, each a number, separated by commas. The radix is temporarily set for this one statement to 10 for DEC or to 8 for OCT. The numbers are placed in successive locations.

Examples:

```
DEC 5,9,4096            ;assembles three words
OCT 5,11,10000          ;assembles the same three words
```

3.3.2 BYTE

The BYTE statement is used to enter bytes of data. Arguments in parentheses indicate the byte size to be used until the next such argument. The first argument of a BYTE statement must be a byte size argument. Other arguments are the byte values. An argument may be any expression that is defined, available, and absolute. Arguments in parentheses (byte size) are interpreted in decimal (base 10) and other arguments in the prevailing radix. Bytes are deposited with the byte instructions, so if a byte will not fit in the current word, it will be put in the left part of the next word. Unused parts of words are filled with zeros. Byte size arguments are not surrounded by commas, but other arguments are separated by commas. For instance, the statement

```
BYTE (7) 3,5(11)6
```

will put two 7-bit bytes (3 and 5) and an 11-bit byte (6) in a word, left justified.

Two successive delimiters, i.e., two commas or a comma and parenthesis, indicate a null argument, which is the same as a zero.

3.3.3 POINT

The POINT pseudo-op assembles a byte pointer in one word. The first argument should be an expression and is interpreted in decimal. The expression must be defined and available. It indicates the byte size, and its value is placed in the size field of the assembled word. The second argument should contain one or more of an index field, an address field, and an at-sign. The third field, if present, indicates the bit position of the low order bit of the byte, i.e., its value is subtracted from 35 (decimal) and placed in the position field. It is interpreted in decimal and must be available. If the third argument is omitted (no comma should be present after the second argument), the position field is set to 36 (decimal) so that the first time the pointer is incremented, it will point to the first byte of the word.

3.3.4 XWD

The XWD statement takes two arguments, separated by a comma, and assembles a single word with the value of the first argument in the left half and the value of the second argument in the right half. Both arguments must be present.

3.3.5 IOWD

IOWD is a permanently defined macro (see section 4, page 38). Its definition is

```
DEFINE IOWD (A,B)
< XWD -(A),B-1 >
```

IOWD takes two arguments and assembles a word in which the negative of the first argument goes in the left halfword and one less than the value of the second argument goes in the right halfword. This format (i.e., negative word count, and memory address minus 1) is often used in communicating with the operating system to specify the address and length of a data block. Also, IOWD may be used to initialize an accumulator for use as a push down pointer.

3.3.6 ASCII, ASCIZ, ASCID, and SIXBIT

There are four text statements: ASCII, ASCIZ, ASCID, and SIXBIT. Each takes as its argument a string of characters starting and ending with, and not otherwise containing, some non-blank character which serves as a delimiter. This delimiter should *not* be any one of the characters: left-arrow (←), colon (:), up-arrow (↑), tilde (~), or number sign (#).

ASCII puts the 7-bit representation of each successive character in the string (excluding the delimiters) in successive words, 5 characters per word, until the string is exhausted. The low order bit of each word and the left-over part of the last word are filled with zero.

ASCIZ is the same as ASCII except that if the last character is the 5th of a word, a word of zero is added at the end. This is to ensure that there is at least one 0 byte at the end.

ASCID works as ASCII except that the low order bit of each word generated is a 1. ASCID assembles data suitable for either the III or Data Disc display systems at Stanford. Also, the ASCID format is used for line numbers in the SOS editor.

SIXBIT works as ASCII except that the characters are converted to the sixbit representation and packed 6 to a word. The last word is filled out with zeros if necessary. Ascii characters are converted to sixbit by replacing the 40 bit with the 100 bit and removing the 100 bit.

3.3.7 RADIX50

This pseudo-op takes two arguments, separated by a comma. The first argument is a number; the second argument is an identifier. The value assembled by the RADIX50 statement is the radix50 representation of the identifier, with the number ORed into the high-order 6 bits. The 2 low-order bits of the number are cleared before ORing.

Radix50 is the representation used for symbol names in the loader, DDT, and RAID. Radix50 is used to condense 6-character symbols into 32 bits. Legal characters are reduced to a value in the range 0-47 octal. The radix50 value is obtained by accumulating a total composed of each character value times a weight. The weight is the power of 50 (octal) corresponding to the character position. The weight of the rightmost non-blank character is 1; the second from the right has weight 50; the third has weight 50*50; etc. The correspondence between characters and their radix50 value is given below:

```
Blank   → 0
0-9     → 1-12
A-Z     → 13-44
.       → 45
$       → 46
%       → 47
```

3.4 Reserving Space for Data

3.4.1 BLOCK

The BLOCK statement is used to reserve a storage area for data. The value of the argument is added to the location counter, so subsequent statements will be assembled beyond the area reserved by BLOCK. The argument must be defined and available. A warning will be given if the argument is negative. The loader will initialize each word reserved by the BLOCK statement to zero; however, well-written programs do their own initialization. Note that the BLOCK pseudo-op has no relation to block structure.

BLOCK N and ORG .+N are equivalent.

3.4.2 INTEGER and ARRAY

INTEGER should be followed by a list of symbols, separated by commas. Each of these symbols is then treated as a variable, i.e., as though it had appeared in the block where the INTEGER appears, followed by a number sign.

The ARRAY statement takes a list of arguments separated by commas. Each argument is a symbol followed by an expression in brackets. The effect is similar to INTEGER, except that the expression (which ought to be defined and available) denotes the number of locations to be reserved (as in BLOCK), with the symbol being the address of the first one. For example,

```
ARRAY FOO[10],BAZ[20]
```

will reserve 10 words for FOO and 20 words for BAZ. The symbols FOO and BAZ are not defined by this statement; they can only be used where forward references are legal.

3.5 Assembler Control Statements

3.5.1 TITLE

TITLE names the program and sets the heading for the pages of the listing. There should be precisely one TITLE statement per program; it should appear before any statement that generates code.

TITLE should be followed by a string of characters, the first part of which should be an identifier. That identifier is used as the program name which DDT and RAID will recognize. It is also used as the name of the outermost block.

The string of characters in the TITLE statement is printed as a part of the heading on all pages subsequent to the one on which the TITLE statement appears; if TITLE appears on the first line of a page, it also affects the heading on that page. The string used in the heading for TITLE is terminated by the first carriage return or semicolon.

If no TITLE statement appears before the first symbols are emitted (generally, at the first BEND or END), then FAIL will generate a title with program name ".MAIN". If a TITLE statement appears after code has been emitted (except for entry blocks), the resulting binary file may be unsuitable for use as part of a library file.

3.5.2 END and PRGEND

The END statement is the last statement of a program. It signals the assembler to stop assembling; no text following it will be processed. If an argument is given, it is taken as the starting address of the program.

An END statement includes implicit VAR and LIT statements (see section 3.1.7, page 25, and section 3.1.6, page 25). That is, all outstanding variables and literals are placed starting at the current value of the location counter when the END is seen. Variables are put out first.

PRGEND is used in place of END when it is desired to assemble more than one program to and/or from a single file. It behaves exactly like END, including taking an optional argument as the starting address, and then restarts FAIL completely, except that I/O is undisturbed. It therefore cannot appear in a macro expansion or similar situation. PRGEND is particularly useful for directly assembling a library which consists of many small programs.

3.5.3 COMMENT

The first non-blank character following the COMMENT pseudo-op is taken as the delimiter. All text from it to the line feed following the next occurrence of this delimiter is ignored by the assembler, except that it is passed to the listing file. The delimiter should *not* be any one of the characters left-arrow (←), colon (:), up-arrow (↑), tilde (~), or number sign (#).

3.5.4 RADIX

The RADIX statement changes the prevailing radix until the next RADIX statement is encountered. It has no effect on numbers preceded by an equal sign. The one argument of RADIX is interpreted in the current radix unless it is preceded by a equal sign. Thus, the statement `RADIX 10` will have no effect (since 10 in the current radix equals the current radix). The radix may be set to almost anything, but for radices above 10 (decimal) there are no digits to represent 10, 11, etc. Zero is not permitted, and 1 should be avoided if one is going to use either an arithmetic FOR macro or a macro argument with this radix.

3.5.5 .INSERT

The .INSERT pseudo-op causes FAIL to remember its position in the current input file and then start reading (and assembling) another file. When the end of the inserted file is reached, FAIL continues processing the original file from the point where it left off. The format is:

```
.INSERT DEV:FILE.EXT[PRJ,PRG]
```

The `DEV:` field is optional (the default is `DSK:`); it specifies the device where the inserted file can be found. The `[PRJ,PRG]` field is optional; it has the usual meaning. In non-Stanford FAIL installations, the file name is scanned in accordance with the convention that prevails at that site.

This pseudo-op will *not* work if it appears in the input stream from any device other than DSK, since random access features are required to accomplish the repositioning of the file.

3.6 Listing Control Statements

These pseudo-ops affect the format of the assembly listing. Several descriptions below refer to command line switches; appendix A, page 48, describes the command line format and the different switches.

3.6.1 TITLE and SUBTTL

The TITLE statement can be used to set the heading that appears on the pages of the listing. See section 3.5.1, page 34.

SUBTTL is followed by a string of characters which is used as a subheading on all subsequent pages until another SUBTTL appears. If SUBTTL appears on the first line of a page, it will affect the subheading of that page also. The string used in the heading for SUBTTL is terminated by the first carriage return or semicolon.

3.6.2 LIST, XLIST, and XLIST1

The XLIST statement causes listing to stop until the next LIST statement. LIST causes listing to resume if it has been stopped by an XLIST or XLIST1 statement. Otherwise it is ignored. LIST is the default.

The XLIST1 statement has exactly the same effect as XLIST unless the /I switch was used in the command string, in which case it is ignored.

3.6.3 LALL and XALL

XALL causes the listing of the body of macros, REPEATs, and FORs to be suppressed during macro expansion. LALL causes it to start up again. LALL is the default.

3.6.4 NOLIT

This statement causes the binary listing of code in literals to be suppressed. This has the same effect as /L in the command string.

3.6.5 NOSYM

This statement disables the listing of the symbol table, counteracting /S in the command string.

3.6.6 CREF and XCREF

These turn on and off the emission of information to CREF, the *Cross-Reference Listing* program. These pseudo-ops have no effect unless /C was used in the command string. CREF is the default.

3.6.7 PAGE

This pseudo-op has the same function as a form feed; it is included for compatibility with MACRO-10. A form feed is placed in the listing immediately following PAGE. The effect is to skip to the top of the next page of the listing. Use of this pseudo-op will destroy the correspondence between listing pages and source file pages, so its use is generally not recommended.

3.6.8 PRINTX

This pseudo-op causes the line on which it appears to be printed on the user's terminal. This is sometimes useful for giving a progress report during long assemblies.

4. Macro Operations

The FAIL macro processor provides features for modifying the input text stream in many ways, such as the ability to abbreviate a frequently occurring sequence with a single identifier or to iterate the input of a stream of text a number of times. In both cases, substitutions can be specified which allow each different occurrence of the text to be somewhat modified. Provision for making the assembly of a body of text conditional on any of a variety of circumstances is also included.

4.1 Macros

Macros are named text strings which may have substitutable arguments. Macros may be used whenever the same or similar pieces of text (code) occur in several places. A macro has a *name* and a *macro body*; also, it may have a *concatenation character* and an *argument list*. The several characteristics of a macro are specified by a DEFINE statement.

DEFINE and the macro name must appear on the same line. The *macro name* is an identifier; it may be followed by an optional concatenation character, which must also be on the same line as DEFINE. The formal arguments, if any, are enclosed in parentheses and separated by commas. The argument list may occur on a subsequent line. The macro body, enclosed in *braces* ({ and }), appears after the argument list in DEFINE.

In the macro processor, braces and brokets are equivalent, i.e., "{" and "<" are equivalent, as are "}" and ">". The equivalence between brokets and braces applies at all times within the macro processor; the text and examples that follow use braces, but brokets can be used instead. The macro processor counts braces independent of context; specifically, braces and brokets that appear in comments, text constants, etc. *are* counted by the macro processor. In the discussion that follows, "non-blank character" omits both blank and tab characters.

4.1.1 Macro Bodies

The *macro body* may be any string of characters, subject to the restriction that the right and left braces must be balanced. The macro body itself is enclosed in braces and appears after the argument list in a DEFINE statement. The macro body is stored in FAIL's memory, associated with the macro name. At any point following the DEFINE statement, the macro body will be substituted for occurrences of the macro name.

4.1.2 Concatenation

The *concatenation character* may be any non-blank character (excluding also carriage return, line feed, and right brace) that appears in DEFINE after the macro name and before the argument list and macro body. This character may then be used to delimit identifiers so that they will be recognized as arguments. Appearances of this character will be deleted from the macro body whenever they appear. This allows a macro argument to be part of an identifier, instead of an entire identifier. See the example at the end of section 4.1.6, page 42.

4.1.3 Arguments in Macro Definitions

Arguments in macro definitions must be identifiers. A list of them, enclosed in parentheses, may appear after the macro name in the definition. If no list of arguments appears before the macro body, it is assumed that there are no arguments.

Each instance of an identifier in the macro body which is the same as one of the arguments will be replaced with the string of text corresponding to that argument when the macro is called. Thus, if FUDLY is one of the arguments in the definition of a macro and the following text appears in the body:

```
A+FUDLY B
```

then FUDLY will be recognized as an argument. But if the following appears:

```
A+FUDLYB
```

then, since FUDLYB is an identifier and is different from FUDLY, it will not be recognized as an argument. To concatenate the "B" above with an actual argument, use a concatenation character. For example, if the concatenation character is "$" and

```
A+FUDLY$B
```

appears in the macro body, then FUDLY will be recognized as an argument, and the "$" will disappear when the macro is expanded.

Here is a sample macro definition:

```
DEFINE FOO (AC,ADDRS)
{MOVNI AC,3
IMUL AC,ADDRS
ADDI AC,37
MOVEM AC,ADDRS+1}
```

If the text:

```
FOO (A,FARB+7)
```

appears in the program somewhere after the DEFINE above, it will expand into:

```
MOVNI A,3
IMUL A,FARB+7
ADDI A,37
MOVEM A,FARB+7+1
```

4.1.4 Macro Calls

A macro name may appear anywhere and will be replaced by the macro body, as long as the name appears as an identifier and is considered to be an identifier by the assembler. A macro name may appear alone on a line or in the accumulator, index, or address field. If the macro name appears in a context where it is not considered to be an identifier, the macro will not be expanded. For example, macro names that appear in a comment or in the text argument of an ASCII statement will not be expanded. Also, there are some other cases where a macro name will not be expanded:

the macro name in DEFINE,
the formal argument list in DEFINE and FOR,
the symbol name in OPDEF, PURGE, SUPPRESS and RADIX50,
the tested symbol in a symbol IF,
the block name in BEGIN and BEND,
the location counter name in USE and SET,
the universal symbol table name in SEARCH, and
the program name in TITLE and UNIVERSAL.

Macros may be used recursively. That is, a macro body may contain a macro call or macro definition. However, if such macro calls are nested too deep, the macro push-down list may overflow, resulting in an error message and termination of the assembly. If this occurs, the /P switch should be used in the command string. Every occurrence of /P in the command string causes the assembler to allocate an extra 200 (octal) words of memory for the macro push-down list (see appendix A, page 48).

4.1.5 Arguments in Macro Calls

The list of arguments to a macro call may be enclosed in parentheses, or not. The arguments themselves are separated by commas. For example, if FOO is the name of a macro that requires two arguments, `FOO A,FARB+7` and `FOO (A,FARB+7)` have the same effect.

If the argument list is enclosed in parentheses, then the first argument begins with the first character after the "(", even if it is blank. Subsequent arguments begin with the first character after the comma that terminates the previous argument. Arguments do not include the comma or ")" used to terminate them. Arguments are scanned until the matching ")" is seen.

If the argument list is not enclosed in parentheses, the first argument begins with the first non-blank character after the macro name. Subsequent arguments begin with the first character after the comma that terminated the previous argument. Arguments do not include the comma or other character used to terminate them. Arguments are scanned until any one of right bracket, right broket, right brace, semicolon, or carriage return is seen.

Two commas in a row with nothing in between signify a *null argument*, i.e., an argument that consists of no characters. If more arguments are called for than are supplied, the last ones are considered to be null. If more arguments are supplied than are called for, the extras are ignored by the macro processor; see section 4.1.6, page 41.

Unless the first character of an argument is "{", the argument terminates at the first comma, right parenthesis, right brace (or broket), right bracket, or carriage return. If the first character of an argument is "{" (or "<"), then all characters included *between* the matching braces are taken as the argument. This allows the argument to contain commas, parentheses, etc. which would not be legal otherwise, but the braces must be kept balanced. In addition, all characters between the "}" that closes the argument and the next argument terminator are ignored. This allows the continuation of a list of arguments from one line to the next (i.e., enclose the last argument on the line in braces and put the comma for it at the start of the next line).

If the first character of an argument is a backslash (\) or right-arrow (→), then the next thing after the backslash (or right-arrow) is considered to be an expression (and it better be defined). The expression is evaluated and the value is converted to a string of ascii digits in the current radix (the radix ought not be 1). This string of digits is taken as the argument. All characters from the end of the expression to the next argument termination character (comma, etc.) are ignored.

4.1.6 How Much is Eaten by a Macro Call

When a macro call appears, some of the text following the macro name is considered to be part of the call. Any text that is not part of the macro call will be assembled as usual. For instance, if

```
DEFINE FOO (A) {A + 7/6}
```

has appeared, then when

```
MOVEI   A,FOO (3) (6) ;comment
```

appears, it will be assembled as

```
MOVEI   A,3 + 7/6 (6) ;comment
```

Thus, the text FOO (3) is considered to be part of the macro call and is "eaten".

The following rules govern how much text gets eaten in a macro call. If the macro was defined as having no arguments, then only the macro name and any following spaces (or tabs) are eaten. If the macro was defined as having arguments and the first non-blank character after the macro name is a left parenthesis, then everything from the macro name to the right parenthesis which closes the argument list, inclusive, is eaten. If the macro was defined as having arguments and the first non-blank character is not a left parenthesis, then everything from the macro name to the comma or carriage return which terminates the last macro argument used is eaten. Thus, if parentheses are not used and too few arguments are supplied, everything from the macro name to the carriage return will be eaten. If parentheses are not used and the macro was defined as having arguments and enough or too many arguments are supplied, then everything from the macro name to the comma (or carriage return) which terminates the last argument used will be eaten.

Example:

```
DEFINE FOO $ (A,B) {A$B}
MOVEI FOO 1,2,  ,37(6)   ;will expand to:
                          ;MOVEI 12  ,37(6)
                          ;"FOO 1,2," has been eaten
```

4.1.7 Complex Example

This example is given without a full explanation. It shows an example of an information carrying macro. The macro BAR is expanded (by being redefined) every time that ADD1 is used. The \BAR in the definition of ADD1 is necessary to cause the evaluation of BAR as an expression (which causes a macro expansion to occur).

Example:

```
          DEFINE BAR {0,}
          DEFINE FOO (A,B,C) {DEFINE BAR {0,<B
C>}}
          DEFINE ADD1 (X) {FOO(\BAR,X)}
          DEFINE SEC (A,B) {B}

                          ;BAR = 0,
          ADD1(X1)
                          ;BAR = 0,<
                          ;       X1>
          ADD1(X2)
                          ;BAR = 0,<
                          ;       X1
                          ;       X2>
          ADD1(X3)
                          ;BAR = 0,<
                          ;       X1
                          ;       X2
                          ;       X3>

          SEC(\BAR)       ;THIS GENERATES THE FOLLOWING:
                          ;       X1
                          ;       X2
                          ;       X3
```

4.2 FOR

There are three types of FORs; all have the same general form. Each consists of the word FOR, an optional *concatenation character*, a *range specifier*, and a *FOR-body*. The FOR statement expands into the text of its FOR-body, possibly with substitutions, repeated once for each element in the range of the FOR. FOR replaces the IRP and IRPC pseudo-ops found in MACRO-10.

The optional *concatenation character* is specified by following the word FOR with an at-sign followed immediately by the concatenation character. If a FOR is used inside a macro and concatenation of FOR arguments is desired, it is necessary to have a concatenation character specified for the FOR which is different from the one for the macro.

The *range specifier* is different for each type of FOR and will be explained below. The FOR statement may have one or two *formal arguments* which are specified in the range specification.

The *FOR-body* has the same form as a macro body; the text is enclosed in braces, and braces must be balanced.

4.2.1 String FOR

The range specification consists of one or two formal argument identifiers, followed by either the identifier "IN" or the *containment character* (c), followed by an argument list. The argument list has the same syntax as a macro call argument list (see section 4.1.5, page 40), but the list must be in parentheses. The effect is that the body of the FOR is assembled once for each element in the argument list, and that element is substituted for the first (or only) formal argument each time. The second formal argument, if present, will have the remainder of the argument list (starting with the element following the one currently substituted for the first argument) substituted for it.

Examples:

Source	Expansion

FOR A IN (QRN,((<JRST 4,>)),STORP)
{MOVSI 13,A
PUSHJ P,GORP
}
	MOVSI 13,QRN
	PUSHJ P,GORP
	MOVSI 13,(<JRST 4,>)
	PUSHJ P,GORP
	MOVSI 13,STORP
	PUSHJ P,GORP

Source	Expansion

FOR ZOT,FUB ⊂ (A,B,C,D)
{MOVEI ZOT,137 ; FUB LEFT
}
	MOVEI A,137 ; B,C,D LEFT
	MOVEI B,137 ; C,D LEFT
	MOVEI C,137 ; D LEFT
	MOVEI D,137 ; LEFT

4.2.2 Character FOR

The range specifier consists of one or two formal arguments followed by either the letter "E" or the character epsilon (ϵ), followed by a string of characters enclosed in braces. The only restriction on the string of characters is that the braces must balance. The body of the FOR is assembled once for each character in the list, with that character substituted for the first formal argument each time and the rest of the string substituted for the second formal argument, if any.

Examples:

Source	Expansion

`FOR ZOT,FUB ϵ {ABCD}`
`{MOVEI ZOT,137 ; FUB LEFT`
`}`
	`MOVEI A,137 ; BCD LEFT`
	`MOVEI B,137 ; CD LEFT`
	`MOVEI C,137 ; D LEFT`
	`MOVEI D,137 ;  LEFT`

Source	Expansion

`FOR @$ QRN E {AZ1Q5}`
`{ZORP$QRN←0`
`}`
	`ZORPA←0`
	`ZORPZ←0`
	`ZORP1←0`
	`ZORPQ←0`
	`ZORP5←0`

4.2.3 Arithmetic FOR

This type of FOR is similar to the ALGOL FOR statement. The range specifier consists of one or two formal arguments followed by a left-arrow, followed by two or three expressions, separated by commas. The expressions are like the two or three arguments of a FORTRAN DO statement. The value of the first is the starting value, the value of the second is the ending value, and the value of the third is the increment. If the third expression is not present, 1 is used as the increment.

The body of the FOR is assembled repeatedly, first for the starting value, then for the starting value plus the increment, etc. until it has been assembled once for each such value which is less than or equal to the ending value (greater than or equal if the increment is negative). If the starting value is already greater than the ending value (less than, for negative increment), the FOR body is not assembled at all. For each repetition, the current value is converted to ascii digits in the current radix, and that string is substituted for the formal argument(s) (both arguments have the same value). Note that all expressions must be defined, available, and absolute.

Examples (assume RADIX =8):

Source	Expansion
FOR I←1+3, 25, 7 {XWD FOO,I }	XWD FOO,4 XWD FOO,13 XWD FOO,22

Source	Expansion
FOR @$ ZOT←11,4,-1 {ZOTQ$ZOT : ZOT +3 }	ZOTQ11 : 11 +3 ZOTQ10 : 10 +3 ZOTQ7 : 7 +3 ZOTQ6 : 6 +3 ZOTQ5 : 5 +3 ZOTQ4 : 4 +3

4.3 REPEAT

The REPEAT statement is included for compatibility with MACRO-10. The format is

```
REPEAT exp, {text}
```

The expression exp is evaluated, and the text is assembled that number of times, with a carriage return and line feed inserted at its end each time. The text is like a macro body: braces must balance.

For example, the statement:

```
REPEAT 3, {0}
```

will expand to:

```
0
0
0
```

4.4 Conditional Assembly

The *conditional assembly* opcodes (the IFs) are like macros: they will be recognized wherever they appear, as long as the assembler sees them as identifiers. Thus, an IF need not be the first thing on a line. Attempts to use IFs as symbols will produce erroneous results.

4.4.1 Numeric IFs

There are six numeric IFs:

IFE exp, {text}	assembles text if exp=0
IFN exp, {text}	assembles text if exp≠0
IFG exp, {text}	assembles text if exp>0
IFL exp, {text}	assembles text if exp<0
IFGE exp, {text}	assembles text if exp≥0
IFLE exp, {text}	assembles text if exp≤0

The expression exp is evaluated. If its value bears the indicated relation to zero, the text is assembled once; otherwise it is not assembled. The text, which is called the *IF-body*, is like a macro body: braces must balance.

Examples:

IFE 3, {ZOT}	assembles nothing
IFGE 15, {JRST START}	assembles JRST START
PUSHJ P,IFN PARM, {BAZ;}FOO	assembles PUSHJ P,BAZ;FOO if PARM≠0
PUSHJ P,IFN PARM, {BAZ;}FOO	assembles PUSHJ P,FOO if PARM=0

4.4.2 Text IFs

There are two text IFs. They are IFIDN and IFDIF, which stand for "if identical" and "if different", respectively. The format is

```
IFIDN {text 1} {text 2} {text 3}
```

The texts can be any string of characters in which the braces balance. For IFIDN, if the two strings text 1 and text 2 are identical in each and every character, the string text 3 will be assembled, otherwise it will not. For IFDIF, if text 1 and text 2 are different, text 3 will be assembled, otherwise it will not.

4.4.3 Symbol IFs

There are eight symbol IFs. They are IFDEF, IFNDEF, IFAVL, IFNAVL, IFOP, IFNOP, IFMAC, and IFNMAC. A typical example is

```
IFDEF symbol, {text}
```

If the indicated condition is true for the symbol, the text is assembled; otherwise it is not. These conditionals come in pairs; if one of a pair is true, the other is false.

IFDEF is true if the symbol is defined in this block or in an outer block. Defined symbols may be either opcodes, macro names, labels, or parameters. IFDEF will be true if the symbol could be used on a line by itself (ignoring possible future definitions).

IFAVL is true if the symbol is available. That is, IFAVL is true if the symbol is defined as an opcode or macro or if it has been defined in this block, declared global in this block and defined in an outer block, or defined in an outer block with a down-arrow.

IFOP is true if the symbol is defined as an opcode.

IFMAC is true if the symbol is defined as a macro (including the IFs, IOWD, and the predefined symbols .FNAM1, .FNAM2, "$." and ".").

Command Language

The basic format of a FAIL command is

```
binary-file,listing-file←source-file-1, ... ,source-file-n
```

File specifications consist of

```
device:file
```

If device: is missing, DSK: is assumed. Either (or both) output file(s) may be omitted. If the listing-file is included, a comma must precede it. Source-file names are separated by commas. The device name for source files is sticky, so to change devices the device name must be explicit, even if it is DSK:. Multiple source files are concatenated as one assembly. If the last source-file name on a line ends with a comma (and carriage return-line feed) then the next line is taken as a continuation of this command.

If no file extension is given for the binary and list files, REL and LST are assumed, respectively (in the non-Stanford FAIL, CRF is the default extension for the list file). If no extension is given for the source file(s), FAI is tried first; failing that, a blank extension is tried.

Switches should follow file names and may be either of the slash type or parentheses type (e.g., "/x" or "(x)").

Device switches (must follow the name of the affected file):

nA	advance magnetic tape n files
nB	backspace magnetic tape n files
T	skip to logical end of magnetic tape
W	rewind magnetic tape
Z	zero DECtape directory

Assembler switches (may appear after any file name):

C	make a cross-reference (CREF) listing
F	don't pause after errors (inverse of R)
I	ignore XLIST1 pseudo-op
J	turn on cross-reference listing output
K	turn off cross-reference listing output
L	don't list literal values with text
N	don't list assembly errors on TTY
R	pause after each assembly error
S	list symbol table
U	underline macro expansions on listing
nV	set the number of lines/page in listing to n
X	don't list macro expansions

The P switch is used to allocate extra space for the macro push-down list (PDL), which is normally 200 (octal) locations long. If recursive macros are used, more space may be needed. The macro PDL will be expanded by 200 words for every occurrence of the P switch in the command string. A numeric argument may given with the P switch to specify a multiple of 200 words by which to expand the PDL.

Sometimes, assembly parameters are specified from the user terminal, rather than being included in the source program. Suppose the line SEGSW←←1 needs to be included in the assembly of the file BAZ. The following command sequence would do that (and make a cross-reference listing of BAZ):

```
BAZ,BAZ/C←TTY:,DSK:BAZ
SEGSW←←1
↑Z
```

The text is typed to FAIL and terminated with control-Z (↑Z) (at Stanford displays, control-meta-line feed is used instead of control-Z). Using RPG (known elsewhere as COMPIL), the command sequence would be

```
COMPILE/CREF TTY:F+DSK:BAZ
SEGSW←←1
↑Z
```

The file name F is needed to satisfy the RPG syntax; the device name DSK: is needed to switch the default input device to DSK.

If the command FILE@ is seen, the named file will be read and interpreted as containing a series of commands of the usual form.

The command FILE! causes FAIL to exit and run the named program. The default device for this command is SYS:.

To provide some compatability with RPG-style commands, FAIL accepts "=" for "←" in the command line. Also, either "+" or ";" may be used instead of "," to separate source-file names.

Relocatable and Undefined Values

FAIL binary programs are usually required to be *relocatable*, i.e., loadable anywhere in a core image. Many values depend upon the absolute location of a program within its core image, e.g., the target address of a branch instruction. The final determination of these values must be made by the loader.

The problem of relocation can usually be reduced to a question of whether or not to augment a value by the *relocation constant*, which is simply the location at which the loader decides to begin loading this program. The mechanism for handling this involves associating with each value a *relocation factor*, which is (at load time) to be multiplied by the relocation constant and added to the value. For the simple relocation mechanism to work, the relocation factor must be a constant and either 0 or 1. Since 36 bits may contain two 18-bit addresses, a relocation factor is provided for each halfword. Thus, a value which is completely determined except for simple relocation can be expressed in 38 bits. A value in which at least one relocation factor is non-zero is said to be *relocatable*; one in which both are zero is said to be *unrelocatable* or *absolute*.

There is a more general, less efficient mechanism for delaying calculations until load time. This is used in more complex cases where the simple relocation scheme is inadequate. Whenever a value cannot be calculated immediately and cannot be handled by the relocation mechanism because it requires some other type of deferred calculation, the value is said to be *undefined*. Undefined values are represented by relatively complex structures which are retained in FAIL for final evaluation or, if necessary, passed to the loader for evaluation. Undefined values are illegal in those contexts which require the value to be immediately known, including some situations where the relocation factor mechanism is legal. The legality of undefined or relocatable values is indicated in the discussion of each possible usage.

Predefined Opcodes

The standard machine instruction mnemonics of the PDP-10 (KA-10) are defined in FAIL.

When the Stanford version of FAIL is started, it obtains from the system the definitions for all system UUOs and CALLIs that are available at the time of the assembly.

The table that follows includes all the pseudo-ops, machine instruction mnemonics, special symbols, and UUO mnemonics currently available at Stanford. The indication SAIL is used to indicate UUOs and machine instructions available only at Stanford. The indication UUO is used to mark system calls that are also available on a DEC system. Hardware I/O instructions are indicated by I/O; these instructions are not available to normal user programs. Machine instruction mnemonics for the KI-10 processor are available as a conditional assembly feature in FAIL; these are flagged with the indication KI. The entry for each pseudo-op includes the page number where that pseudo-op is explained.

Note that there are sometimes subtle differences between DEC system UUOs and Stanford UUOs; consult the appropriate reference manual. Also note that some DEC mnemonics conflict with those used at Stanford.

$.	Predefined	page 14
.	Predefined	page 14
.FNAM1	Predefined	page 15
.FNAM2	Predefined	page 15
.INSERT	Pseudo-Op	page 35
.LIBRARY	Pseudo-Op	page 28
.LOAD	Pseudo-Op	page 28
ACCTIM	CALLI 400101	SAIL
ACTCHR	CALLI 400105	SAIL
ADD	270000,,0	
ADDB	273000,,0	
ADDI	271000,,0	
ADDM	272000,,0	
ADSMAP	CALLI 400110	SAIL
AND	404000,,0	
ANDB	407000,,0	
ANDCA	410000,,0	
ANDCAB	413000,,0	
ANDCAI	411000,,0	
ANDCAM	412000,,0	
ANDCB	440000,,0	
ANDCBB	443000,,0	
ANDCBI	441000,,0	
ANDCBM	442000,,0	
ANDCM	420000,,0	
ANDCMB	423000,,0	
ANDCMI	421000,,0	
ANDCMM	422000,,0	
ANDI	405000,,0	
ANDM	406000,,0	
AOBJN	253000,,0	
AOBJP	252000,,0	
AOJ	340000,,0	
AOJA	344000,,0	
AOJE	342000,,0	
AOJG	347000,,0	
AOJGE	345000,,0	
AOJL	341000,,0	
AOJLE	343000,,0	
AOJN	346000,,0	
AOS	350000,,0	
AOSA	354000,,0	
AOSE	352000,,0	
AOSG	357000,,0	
AOSGE	355000,,0	
AOSL	351000,,0	
AOSLE	353000,,0	
AOSN	356000,,0	
APRENB	CALLI 16	UUO
ARRAY	Pseudo-Op	page 33
ASCID	Pseudo-Op	page 32
ASCII	Pseudo-Op	page 32
ASCIZ	Pseudo-Op	page 32
ASH	240000,,0	
ASHC	244000,,0	
ASUPPRESS	Pseudo-Op	page 29
ATTSEG	CALLI 400016	SAIL
BEEP	CALLI 400111	SAIL
BEGIN	Pseudo-Op	page 26
BEND	Pseudo-Op	page 26
BLKI	700000,,0	I/O
BLKO	700100,,0	I/O
BLOCK	Pseudo-Op	page 33

BLT	251000,,0	
BUFLEN	CALLI 400042	SAIL
BYTE	Pseudo-Op	page 30
CAI	300000,,0	
CAIA	304000,,0	
CAIE	302000,,0	
CAIG	307000,,0	
CAIGE	305000,,0	
CAIL	301000,,0	
CAILE	303000,,0	
CAIN	306000,,0	
CALL	040000,,0	UUO
CALLI	047000,,0	UUO
CALLIT	CALLI 400074	SAIL
CAM	310000,,0	
CAMA	314000,,0	
CAME	312000,,0	
CAMG	317000,,0	
CAMGE	315000,,0	
CAML	311000,,0	
CAMLE	313000,,0	
CAMN	316000,,0	
CHNSTS	716000,,0	SAIL
CLKINT	717000,,0	SAIL
CLOSE	070000,,0	UUO
CLRBFI	051440,,0	UUO
CLRBFO	051500,,0	UUO
COMMENT	Pseudo-Op	page 34
CONI	700240,,0	I/O
CONO	700200,,0	I/O
CONS	257000,,0	SAIL
CONSO	700340,,0	I/O
CONSZ	700300,,0	I/O
CORE	CALLI 11	UUO
CORE2	CALLI 400015	SAIL
CREF	Pseudo-Op	page 37
CTLV	CALLI 400001	SAIL
DATAI	700040,,0	I/O
DATAO	700140,,0	I/O
DATE	CALLI 14	UUO
DAYCNT	CALLI 400100	SAIL
DDCHAN	CALLI 400067	SAIL
DDTGT	CALLI 5	UUO
DDTIN	CALLI 1	UUO
DDTOUT	CALLI 3	UUO
DDTRL	CALLI 7	UUO
DDUPG	715140,,0	SAIL
DEBREAK	CALLI 400035	SAIL
DEC	Pseudo-Op	page 30
DEFINE	Pseudo-Op	page 38
DEPHASE	Pseudo-Op	page 23
DETSEG	CALLI 400017	SAIL
DEVCHR	CALLI 4	UUO
DEVNUM	CALLI 400104	SAIL
DEVUSE	CALLI 400051	SAIL
DFAD	110000,,0	KI
DFDV	113000,,0	KI
DFMP	112000,,0	KI
DFN	131000,,0	
DFSB	111000,,0	KI
DIAL	CALLI 400117	SAIL
DISMISS	CALLI 400024	SAIL
DIV	234000,,0	

DIVB	237000,,0	
DIVI	235000,,0	
DIVM	236000,,0	
DMOVE	120000,,0	KI
DMOVEM	124000,,0	KI
DMOVN	121000,,0	KI
DMOVNM	125000,,0	KI
DPB	137000,,0	
DPYCLR	701000,,0	SAIL
DPYOUT	703000,,0	SAIL
DPYPOS	702100,,0	SAIL
DPYSIZ	702140,,0	SAIL
DSKPPN	CALLI 400071	SAIL
DSKTIM	CALLI 400072	SAIL
EIOTM	CALLI 400005	SAIL
END	Pseudo-Op	page 34
ENTER	077000,,0	UUO
ENTRY	Pseudo-Op	page 27
EQV	444000,,0	
EQVB	447000,,0	
EQVI	445000,,0	
EQVM	446000,,0	
EXCH	250000,,0	
EXIT	CALLI 12	UUO
EXTERNAL	Pseudo-Op	page 27
FAD	140000,,0	
FADB	143000,,0	
FADL	141000,,0	
FADM	142000,,0	
FADR	144000,,0	
FADRB	147000,,0	
FADRI	145000,,0	
FADRL	145000,,0	
FADRM	146000,,0	
FBREAD	706000,,0	SAIL
FBWAIT	CALLI 400057	SAIL
FBWRT	707000,,0	SAIL
FDV	170000,,0	
FDVB	173000,,0	
FDVL	171000,,0	
FDVM	172000,,0	
FDVR	174000,,0	
FDVRB	177000,,0	
FDVRI	175000,,0	
FDVRL	175000,,0	
FDVRM	176000,,0	
FIX	247000,,0	SAIL
FIX	122000,,0	KI
FIXR	126000,,0	KI
FLTR	127000,,0	KI
FMP	160000,,0	
FMPB	163000,,0	
FMPL	161000,,0	
FMPM	162000,,0	
FMPR	164000,,0	
FMPRB	167000,,0	
FMPRI	165000,,0	
FMPRL	165000,,0	
FMPRM	166000,,0	
FOR	Pseudo-Op	page 43
FSB	150000,,0	
FSBB	153000,,0	
FSBL	151000,,0	

FSBM	152000,,0	
FSBR	154000,,0	
FSBRB	157000,,0	
FSBRI	155000,,0	
FSBRL	155000,,0	
FSBRM	156000,,0	
FSC	132000,,0	
GOPTIM	CALLI 400065	SAIL
GETCHR	CALLI 6	UUO
GETLIN	051300,,0	SAIL
GETLN	CALLI 34	UUO
GETNAM	CALLI 400062	SAIL
GETPPN	CALLI 24	UUO
GETPR2	CALLI 400053	SAIL
GETPRV	CALLI 400115	SAIL
GETSEG	CALLI 40	UUO
GETSTS	062000,,0	UUO
GETTAB	CALLI 41	UUO
GLOBAL	Pseudo-Op	page 26
HALT	254200,,0	
HISEG	Pseudo-Op	page 24
HLL	500000,,0	
HLLE	530000,,0	
HLLEI	531000,,0	
HLLEM	532000,,0	
HLLES	533000,,0	
HLLI	501000,,0	
HLLM	502000,,0	
HLLO	520000,,0	
HLLOI	521000,,0	
HLLOM	522000,,0	
HLLOS	523000,,0	
HLLS	503000,,0	
HLLZ	510000,,0	
HLLZI	511000,,0	
HLLZM	512000,,0	
HLLZS	513000,,0	
HLR	544000,,0	
HLRE	574000,,0	
HLREI	575000,,0	
HLREM	576000,,0	
HLRES	577000,,0	
HLRI	545000,,0	
HLRM	546000,,0	
HLRO	564000,,0	
HLROI	565000,,0	
HLROM	566000,,0	
HLROS	567000,,0	
HLRS	547000,,0	
HLRZ	554000,,0	
HLRZI	555000,,0	
HLRZM	556000,,0	
HLRZS	557000,,0	
HRL	504000,,0	
HRLE	534000,,0	
HRLEI	535000,,0	
HRLEM	536000,,0	
HRLES	537000,,0	
HRLI	505000,,0	
HRLM	506000,,0	
HRLO	524000,,0	
HRLOI	525000,,0	
HRLOM	526000,,0	
HRLOS	527000,,0	
HRLS	507000,,0	
HRLZ	514000,,0	
HRLZI	515000,,0	
HRLZM	516000,,0	
HRLZS	517000,,0	
HRR	540000,,0	
HRRE	570000,,0	
HRREI	571000,,0	
HRREM	572000,,0	
HRRES	573000,,0	
HRRI	541000,,0	
HRRM	542000,,0	
HRRO	560000,,0	
HRROI	561000,,0	
HRROM	562000,,0	
HRROS	563000,,0	
HRRS	543000,,0	
HRRZ	550000,,0	
HRRZI	551000,,0	
HRRZM	552000,,0	
HRRZS	553000,,0	
IBP	133000,,0	
IOIV	230000,,0	
IOIVB	233000,,0	
IOIVI	231000,,0	
IOIVM	232000,,0	
IOPB	136000,,0	
IENBW	CALLI 400045	SAIL
IFAVL	Conditional	page 47
IFOEF	Conditional	page 47
IFOIF	Conditional	page 46
IFE	Conditional	page 46
IFG	Conditional	page 46
IFGE	Conditional	page 46
IFION	Conditional	page 46
IFL	Conditional	page 46
IFLE	Conditional	page 46
IFMAC	Conditional	page 47
IFN	Conditional	page 46
IFNAVL	Conditional	page 47
IFNOEF	Conditional	page 47
IFNMAC	Conditional	page 47
IFNOP	Conditional	page 47
IFOP	Conditional	page 47
ILDB	134000,,0	
IMSKCL	722000,,0	SAIL
IMSKCR	723240,,0	SAIL
IMSKST	721000,,0	SAIL
IMSTW	723040,,0	SAIL
IMUL	220000,,0	
IMULB	223000,,0	
IMULI	221000,,0	
IMULM	222000,,0	
IN	056000,,0	UUO
INBUF	064000,,0	UUO
INCHRS	051100,,0	UUO
INCHRW	051000,,0	UUO
INCHSL	051240,,0	UUO
INCHWL	051200,,0	UUO
INIT	041000,,0	UUO
INPUT	066000,,0	UUO
INSKIP	051540,,0	SAIL
INTACM	CALLI 400027	SAIL
INTOEJ	723000,,0	SAIL
INTOMP	723140,,0	SAIL
INTEGER	Pseudo-Op	page 33
INTENB	CALLI 400025	SAIL
INTENS	CALLI 400030	SAIL
INTERNAL	Pseudo-Op	page 27
INTGEN	CALLI 400033	SAIL
INTIIP	CALLI 400031	SAIL
INTIPI	723200,,0	SAIL
INTIRQ	CALLI 400032	SAIL
INTJEN	723000,,0	SAIL
INTMSK	720000,,0	SAIL
INTORM	CALLI 400026	SAIL
INTUUO	723000,,0	SAIL
INWAIT	051600,,0	SAIL
IOPOL	726000,,0	SAIL
IOPOP	725000,,0	SAIL
IOPUSH	724000,,0	SAIL
IOR	434000,,0	
IORB	437000,,0	
IORI	435000,,0	
IORM	436000,,0	
IOWO	Pseudo-Op	page 31
IWAIT	CALLI 400040	SAIL
IWKMSK	723100,,0	SAIL
JBTSTS	CALLI 400013	SAIL
JCRY	255300,,0	
JCRY0	255200,,0	
JCRY1	255100,,0	
JEN	254500,,0	
JFCL	255000,,0	
JFFO	243000,,0	
JFOV	255040,,0	
JOBRD	CALLI 400050	SAIL
JOV	255400,,0	
JRA	267000,,0	
JRST	254000,,0	
JRSTF	254100,,0	
JSA	266000,,0	
JSP	265000,,0	
JSR	264000,,0	
JUMP	320000,,0	
JUMPA	324000,,0	
JUMPE	322000,,0	
JUMPG	327000,,0	
JUMPGE	325000,,0	
JUMPL	321000,,0	
JUMPLE	323000,,0	
JUMPN	326000,,0	
LALL	Pseudo-Op	page 36
LOB	135000,,0	
LEYPOS	702300,,0	SAIL
LINK	Pseudo-Op	page 27
LINKENO	Pseudo-Op	page 27
LINKUP	CALLI 400023	SAIL
LIOTM	CALLI 400006	SAIL
LIST	Pseudo-Op	page 36
LIT	Pseudo-Op	page 25
LOC	Pseudo-Op	page 22
LOCK	CALLI 400076	SAIL
LOGIN	CALLI 15	UUO
LOGOUT	CALLI 17	UUO
LOOKUP	076000,,0	UUO

LSH	242000,,0		
LSHC	246000,,0		
MAIL	710000,,0	SAIL	
MAP	257000,,0	KI	
MOVE	200000,,0		
MOVEI	201000,,0		
MOVEM	202000,,0		
MOVES	203000,,0		
MOVM	214000,,0		
MOVMI	215000,,0		
MOVMM	216000,,0		
MOVMS	217000,,0		
MOVN	210000,,0		
MOVNI	211000,,0		
MOVNM	212000,,0		
MOVNS	213000,,0		
MOVS	204000,,0		
MOVSI	205000,,0		
MOVSM	206000,,0		
MOVSS	207000,,0		
MSTIME	CALLI 23	UUO	
MTAPE	072000,,0	UUO	
MUL	224000,,0		
MULB	227000,,0		
MULI	225000,,0		
MULM	226000,,0		
NAMEIN	CALLI 400043	SAIL	
NOLIT	Pseudo-Op	page 36	
NOSYM	Pseudo-Op	page 37	
OCT	Pseudo-Op	page 30	
OPDEF	Pseudo-Op	page 26	
OPEN	050000,,0	UUO	
OR	434000,,0		
ORB	437000,,0		
ORCA	454000,,0		
ORCAB	457000,,0		
ORCAI	455000,,0		
ORCAM	456000,,0		
ORCB	470000,,0		
ORCBB	473000,,0		
ORCBI	471000,,0		
ORCBM	472000,,0		
ORCM	464000,,0		
ORCMB	467000,,0		
ORCMI	465000,,0		
ORCMM	466000,,0		
ORG	Pseudo-Op	page 22	
ORI	435000,,0		
ORM	436000,,0		
OUT	057000,,0	UUO	
OUTBUF	065000,,0	UUO	
OUTCHR	051040,,0	UUO	
OUTFIV	051740,,0	SAIL	
OUTPUT	067000,,0	UUO	
OUTSTR	051140,,0	UUO	
PAGE	Pseudo-Op	page 37	
PEEK	CALLI 33	UUO	
PGACT	715040,,0	SAIL	
PGCLR	715100,,0	SAIL	
PGINFO	715200,,0	SAIL	
PGIOT	715000,,0	SAIL	
PGSEL	715000,,0	SAIL	
PHASE	Pseudo-Op	page 23	
PJOB	CALLI 30	UUO	
PNAME	CALLI 400007	SAIL	
POINT	Pseudo-Op	page 31	
POINTS	712000,,0	SAIL	
POP	262000,,0		
POPJ	263000,,0		
PORTAL	254040,,0	KI	
PPACT	702040,,0	SAIL	
PPHLD	702340,,0	SAIL	
PPINFO	702240,,0	SAIL	
PPIOT	702000,,0	SAIL	
PPREL	702200,,0	SAIL	
PPSEL	702000,,0	SAIL	
PPSPY	CALLI 400107	SAIL	
PRGEND	Pseudo-Op	page 34	
PRINTX	Pseudo-Op	page 37	
PTGETL	711540,,0	SAIL	
PTIFRE	711100,,0	SAIL	
PTJOBX	711700,,0	SAIL	
PTLOAD	711640,,0	SAIL	
PTOCNT	711140,,0	SAIL	
PTRD1S	711200,,0	SAIL	
PTRD1W	711240,,0	SAIL	
PTRDS	711400,,0	SAIL	
PTSETL	711600,,0	SAIL	
PTWR1S	711300,,0	SAIL	
PTWR1W	711340,,0	SAIL	
PTWRS7	711440,,0	SAIL	
PTWRS9	711500,,0	SAIL	
PTYGET	711000,,0	SAIL	
PTYREL	711040,,0	SAIL	
PTYUUO	711000,,0	SAIL	
PURGE	Pseudo-Op	page 28	
PUSH	261000,,0		
PUSHJ	260000,,0		
PZE	000000,,0		
RADIX	Pseudo-Op	page 35	
RADIX50	Pseudo-Op	page 32	
REASSI	CALLI 21	UUO	
RELEAS	071000,,0	UUO	
RELOC	Pseudo-Op	page 22	
REMAP	CALLI 37	UUO	
RENAME	055000,,0	UUO	
REPEAT	Pseudo-Op	page 45	
RESCAN	051400,,0	UUO	
RESET	CALLI 0	UUO	
RLEVEL	CALLI 400054	SAIL	
ROT	241000,,0		
ROTC	245000,,0		
RUN	CALLI 35	UUO	
RUNMSK	CALLI 400046	SAIL	
RUNTIM	CALLI 27	UUO	
SEARCH	Pseudo-Op	page 29	
SEGNAM	CALLI 400037	SAIL	
SEGNUM	CALLI 400021	SAIL	
SEGSIZ	CALLI 400022	SAIL	
SEND	710000,,0	SAIL	
SET	Pseudo-Op	page 22	
SETA	424000,,0		
SETAB	427000,,0		
SETACT	051640,,0	SAIL	
SETAI	425000,,0		
SETAM	426000,,0		
SETCA	450000,,0		
SETCAB	453000,,0		
SETCAI	451000,,0		
SETCAM	452000,,0		
SETCM	460000,,0		
SETCMB	463000,,0		
SETCMI	461000,,0		
SETCMM	462000,,0		
SETCRD	CALLI 400073	SAIL	
SETDDT	CALLI 2	UUO	
SETLIN	051340,,0	SAIL	
SETM	414000,,0		
SETMB	417000,,0		
SETMI	415000,,0		
SETMM	416000,,0		
SETNAM	CALLI 43	UUO	
SETNM2	CALLI 400036	SAIL	
SETO	474000,,0		
SETOB	477000,,0		
SETOI	475000,,0		
SETOM	476000,,0		
SETPOV	CALLI 32	UUO	
SETPR2	CALLI 400052	SAIL	
SETPRO	CALLI 400020	SAIL	
SETPRV	CALLI 400066	SAIL	
SETSTS	060000,,0	UUO	
SETUWP	CALLI 36	UUO	
SETZ	400000,,0		
SETZB	403000,,0		
SETZI	401000,,0		
SETZM	402000,,0		
SIXBIT	Pseudo-Op	page 32	
SKIP	330000,,0		
SKIPA	334000,,0		
SKIPE	332000,,0		
SKIPG	337000,,0		
SKIPGE	335000,,0		
SKIPL	331000,,0		
SKIPLE	333000,,0		
SKIPN	336000,,0		
SKPHIM	710200,,0	SAIL	
SKPME	710140,,0	SAIL	
SKPSEN	710240,,0	SAIL	
SLEEP	CALLI 31	UUO	
SLEVEL	CALLI 400044	SAIL	
SNEAKS	CALLI 400064	SAIL	
SNEAKW	CALLI 400063	SAIL	
SOJ	360000,,0		
SOJA	364000,,0		
SOJE	362000,,0		
SOJG	367000,,0		
SOJGE	365000,,0		
SOJL	361000,,0		
SOJLE	363000,,0		
SOJN	366000,,0		
SOS	370000,,0		
SOSA	374000,,0		
SOSE	372000,,0		
SOSG	377000,,0		
SOSGE	375000,,0		

SOSL	371000,,0	
SOSLE	373000,,0	
SOSN	376000,,0	
SPCWAR	043000,,0	SAIL
SPCWGO	CALLI 400003	SAIL
SPWBUT	CALLI 400000	SAIL
SRCV	710100,,0	SAIL
STATO	061000,,0	UUO
STATZ	063000,,0	UUO
SUB	274000,,0	
SUBB	277000,,0	
SUBI	275000,,0	
SUBM	276000,,0	
SUBTTL	Pseudo-Op	page 36
SUPPRESS	Pseudo-Op	page 29
SWAP	CALLI 400004	SAIL
SWITCH	CALLI 20	UUO
TDC	650000,,0	
TDCA	654000,,0	
TDCE	652000,,0	
TDCN	656000,,0	
TDN	610000,,0	
TDNA	614000,,0	
TDNE	612000,,0	
TDNN	616000,,0	
TDO	670000,,0	
TDOA	674000,,0	
TDOE	672000,,0	
TDON	676000,,0	
TDZ	630000,,0	
TDZA	634000,,0	
TDZE	632000,,0	
TDZN	636000,,0	
TIMER	CALLI 22	UUO
TITLE	Pseudo-Op	page 34
TLC	641000,,0	
TLCA	645000,,0	
TLCE	643000,,0	
TLCN	647000,,0	
TLN	601000,,0	
TLNA	605000,,0	
TLNE	603000,,0	
TLNN	607000,,0	
TLO	661000,,0	
TLOA	665000,,0	
TLOE	663000,,0	
TLON	667000,,0	
TLZ	621000,,0	
TLZA	625000,,0	
TLZE	623000,,0	
TLZN	627000,,0	
TMPCOR	CALLI 44	UUO
TMPCRD	CALLI 400103	SAIL
TRC	640000,,0	
TRCA	644000,,0	
TRCE	642000,,0	
TRCN	646000,,0	
TRN	600000,,0	
TRNA	604000,,0	
TRNE	602000,,0	
TRNN	606000,,0	
TRO	660000,,0	
TROA	664000,,0	
TROE	662000,,0	
TRON	666000,,0	
TRZ	620000,,0	
TRZA	624000,,0	
TRZE	622000,,0	
TRZN	626000,,0	
TSC	651000,,0	
TSCA	655000,,0	
TSCE	653000,,0	
TSCN	657000,,0	
TSN	611000,,0	
TSNA	615000,,0	
TSNE	613000,,0	
TSNN	617000,,0	
TSO	671000,,0	
TSOA	675000,,0	
TSOE	673000,,0	
TSON	677000,,0	
TSZ	631000,,0	
TSZA	635000,,0	
TSZE	633000,,0	
TSZN	637000,,0	
TTCALL	051000,,0	UUO
TTREAD	051700,,0	SAIL
TTYIOS	CALLI 400014	SAIL
TTYJOB	CALLI 400113	SAIL
TTYMES	CALLI 400047	SAIL
TTYSKP	CALLI 400116	SAIL
TTYUUO	051000,,0	UUO
TWOSEG	Pseudo-Op	page 24
UFA	130000,,0	
UFBCLR	CALLI 400012	SAIL
UFBERR	CALLI 400060	SAIL
UFBGET	CALLI 400010	SAIL
UFBGIV	CALLI 400011	SAIL
UFBPHY	CALLI 400055	SAIL
UFBSKP	CALLI 400056	SAIL
UGETF	073000,,0	UUO
UINBF	704000,,0	SAIL
UNIVERSAL	Pseudo-Op	page 29
UNLOCK	CALLI 400077	SAIL
UNPURE	CALLI 400102	SAIL
UOUTBF	705000,,0	SAIL
UPGIOT	703000,,0	SAIL
UPGMVE	713000,,0	SAIL
UPGMVM	714000,,0	SAIL
USE	Pseudo-Op	page 22
USETI	074000,,0	UUO
USETO	075000,,0	UUO
USKIP	CALLI 400041	SAIL
UTPCLR	CALLI 13	UUO
UUOSIM	CALLI 400106	SAIL
UWAIT	CALLI 400034	SAIL
VAR	Pseudo-Op	page 25
VDSMAP	CALLI 400070	SAIL
WAIT	CALLI 10	UUO
WAKEME	CALLI 400061	SAIL
WHO	CALLI 400112	SAIL
WRCV	710040,,0	SAIL
XALL	Pseudo-Op	page 36
XCREF	Pseudo-Op	page 37
XCT	256000,,0	
XGPUUO	CALLI 400075	SAIL
XLIST	Pseudo-Op	page 36
XLIST1	Pseudo-Op	page 36
XOR	430000,,0	
XORB	433000,,0	
XORI	431000,,0	
XORM	432000,,0	
XPUNGE	Pseudo-Op	page 29
XWD	Pseudo-Op	page 31

Stanford Character Set

The *Stanford Character Set* is displayed in the following table. The three-digit octal code for a character is composed of the number at the left of its row plus the digit at the top of its column. For example, the code for A is 100+1 or 101.

	0	1	2	3	4	5	6	7
000	NUL	↓	α	β	∧	¬	ε	π
010	λ	TAB	LF	VT	FF	CR	∞	∂
020	⊂	⊃	∩	∪	∀	∃	⊗	↔
030	_	→	~	≠	≤	≥	≡	∨
040	SP	!	"	#	$	%	&	'
050	(	)	*	+	,	-	.	/
060	0	1	2	3	4	5	6	7
070	8	9	:	;	<	=	>	?
100	@	A	B	C	D	E	F	G
110	H	I	J	K	L	M	N	O
120	P	Q	R	S	T	U	V	W
130	X	Y	Z	[	\	]	↑	←
140	`	a	b	c	d	e	f	g
150	h	i	j	k	l	m	n	o
160	p	q	r	s	t	u	v	w
170	x	y	z	{	|	ALT	}	BS

NUL	Null
TAB	Horizontal Tab
LF	Line Feed
VT	Vertical Tab
FF	Form Feed
CR	Carriage Return
SP	Space
ALT	Altmode
BS	Back Space

Summary of Character Interpretations

The characters listed below have special meaning in the contexts indicated. These interpretations do not apply when these characters appear in text strings or in comments.

000	NUL	null	ignored on input
001	↓	down-arrow	makes a symbol available in a lower block
002	α	alpha	
003	β	beta	
004	∧	logical and	boolean AND
005	¬	logical not	boolean NOT
00[illegible]	ε	epsilon	delimiter in FOR
[illegible]	π	pi	
[illegible]0	λ	lambda	
011	TAB	tab	same as space (040)
012	LF	line feed	line delimiter
013	VT	vertical tab	
014	FF	form feed	line delimiter; causes new listing page
015	CR	carriage return	statement terminator
016	∞	infinity	
017	∂	partial	
020	⊂	containment	delimiter in FOR
021	⊃	implication	
022	∩	set intersection	
023	∪	set union	
024	∀	for all	
025	∃	there exists	
026	⊗	circle times	arithmetic shift operator
027	↔	double-arrow	statement terminator; remainder of line is interpreted as another statement
030	_	underbar	same as . (056) in identifiers
031	→	right-arrow	same as backslash (134)
032	~	tilde	same as up-arrow (136); illegal as the delimiter in ASCIZ, COMMENT, etc.
033	≠	not equal	boolean XOR
034	≤	less or equal	
035	≥	greater or equal	same as not equal (033)
036	≡	equivalence	
037	∨	logical or	boolean OR
040	SP	space	general delimiter
041	!	exclamation	same as logical or (037)
042	"	double quote	delimits ascii constants
043	#	number sign	declares a variable; illegal as the delimiter in ASCIZ, COMMENT, etc.
044	$	dollar sign	may be used in identifiers
045	%	percent	may be used in identifiers
046	&	ampersand	same as logical and (004)
047	'	close single quote	delimits sixbit constants
050	(	left parenthesis	encloses macro arguments, expressions, and index fields
051	)	right parenthesis	see left parenthesis (050)
052	*	asterisk	integer multiply
053	+	plus	integer addition
054	,	comma	general argument separator
055	-	minus	integer subtraction or negation
056	.	point	may be used in identifiers, floating point numbers, or predefined symbol
057	/	slash	integer division
060	0	digits	used to form number, parts of identifiers
...			
071	9		
072	:	colon	used to define labels; illegal as the delimiter in ASCIZ, COMMENT, etc.
073	;	semicolon	forces remainder of line to be a comment
074	<	left broket	delimits complex atoms; same as left brace (173) to the macro processor
075	=	equal	denotes decimal number; alternate to left-arrow (137) in assignment statements
076	>	right broket	see left broket (074)
077	?	question mark	same as down-arrow (001)

100 @	at-sign	sets indirect bit in instructions; precedes concatenation character in FOR
101 A	upper case letters	used for identifiers; B and E are special in numbers; E is special in FOR
...		
132 Z		
133 [	left bracket	delimits literals, value part of OPDEF, size in ARRAY, PPN in .LOAD
134 \	backslash	evaluate a macro argument and converts the result to a digit string
135]	right bracket	see left bracket (133)
136 ↑	up-arrow	moves a symbol definition to an outer block; makes a symbol INTERNAL or EXTERNAL; illegal as the delimiter in ASCIZ, COMMENT, etc.
137 ←	left-arrow	denotes assignment statement; arithmetic FOR; illegal as the delimiter in ASCIZ, COMMENT, etc.
140 '	open single quote	same as at-sign (100)
141 a	lower case letters	same as upper case letters, except in text constants
...		
172 z		
173 {	left brace	delimits macro bodies, IF-bodies, FOR-bodies, macro arguments
174 |	vertical bar	
175 ALT	altmode	same as right brace (176)
176 }	right brace	see left brace (173)
177 BS	backspace	illegal in input

Index