PERKIN-ELMER

COMMON ASSEMBLY LANGUAGE/32
(CAL/32)

Reference Manual

SPECIAL NOTICE TO
CAL/32 USERS

The CAL/32 assembler is not compatible with the CAL assembler in the handling of unreferenced externals. In order to support weak externals, object code will be generated for unreferenced externals. Programs assembled by CAL/32 and containing unreferenced externals to nonexistent modules will receive an error message at Link time. For compatibility with CAL, the NUREX start option/assembler instruction has been introduced in CAL/32.

TABLE OF CONTENTS

CHAPTERS (Continued)

CHAPTERS (Continued)

APPENDIXES

FIGURES

TABLES

TABLES (Continued)

PREFACE

This manual describes the Perkin-Elmer Common Assembly Language/32 (CAL/32). Chapter 1 is an introduction to the basic concepts of the assembler, central processing unit (CPU), and main memory. Also described are the instruction formats for 16- and 32-bit machines, as well as variations in the formats. Chapter 2 introduces assembly language symbolic representation and describes symbolic values. Chapter 3 defines the source program and contains a list of machine instructions, mnemonics, and a detailed description of assembler instructions. Common mode programming and common mode operations are explained in Chapter 4.

CAL/32 operating instructions are listed in Appendix A. Appendix B contains CAL/32 error codes. Appendix C describes the Perkin-Elmer 32-bit object code format and the Perkin-Elmer 16-bit object code format.

R00 of this manual can be used only with CAL/32 R09 and OS/32 R06 and higher. The following features are incorporated:

- Support of forward references in the operand of EQU, ORG, DS, and DO

- Ease of use (via DEL and temp files)

- Support of lowercase and underscore

- Faster symbol table paging

- Enhanced Copy

- Externals with offsets

- New start options

The CAL/32 user should be familiar with the following 16- and 32-bit manuals:

MANUAL NAME	PUBLICATION NUMBER
Model 8/32 Processor User's Manual	29-428
Model 7/32 Processor User's Manual	C29-405

MANUAL NAME	PUBLICATION NUMBER
16-Bit Processor User's Manual	C29-509
Series Sixteen Processor User Manual	C29-674
Model 3240 Processor User's Manual	C29-685
Model 3220 Processor User's Manual	C29-693
OS/16 MT2 Programmer's Reference Manual	S29-429
OS/16 MT2 Operator's Manual	S29-430
OS/32 Operator Reference Manual	48-030
OS/32 Supervisor Call (SVC) Reference Manual	48-038
OS/32 Application Level Programmer Reference Manual	48-039
32-Bit System User Documentation Summary	50-003

For further information on the contents of all Perkin-Elmer 32-bit manuals, see the 32-Bit Systems User Documentation Summary.

CHAPTER 1
BASIC CONCEPTS

1.1 INTRODUCTION

Like all assemblers, Common Assembly Language/32 (CAL/32) simplifies the direct control of the processor by providing the programmer with a way of representing actual machine operations in an easily understood symbolic form. Assemblers translate symbolic representations of machine instructions into binary form to be executed by the processor. CAL/32 also includes such features as relocation, segmentation, complex data definitions, and expression analysis. CAL/32 can run on any Perkin-Elmer processor and produce machine code for any Perkin-Elmer processor.

Because assembly language programming is so close to actual machine operations, it is essential that the assembly language programmer have a good understanding of the system architecture. This chapter contains introductory architectural descriptions, and the programmer should read the appropriate processor reference manuals for more information.

1.2 THE PROCESSOR

All Perkin-Elmer processors are stored-program, multi-register, two-address machines. The major components are the central processing unit (CPU) and main memory.

1.2.1 Central Processing Unit (CPU)

To many, the CPU is something of a black box that performs arithmetic and logical operations in a precise and controlled way. To the assembly language programmer, the CPU is not completely opaque. Three of its components--local memory or registers, the program status word (PSW), and the I/O interface--are accessible through assembly language programming.

1.2.1.1 Local Memory (Registers)

All Perkin-Elmer processors have some amount of local memory that can be used as temporary storage, accumulators, or indices. There are three types of local memory:

• General purpose registers

• Single precision floating point registers

• Double precision floating point registers

All processors have at least one set of 16 general purpose registers. In the 16-bit processors, each general purpose register holds 16 binary digits; in the 32-bit processors, 32 binary digits. General purpose registers can be used for integer arithmetic, address arithmetic, logical operations, and character operations.

Floating point registers are used only for floating point arithmetic operations. Processors with floating point registers have either eight single precision registers, or eight single precision registers and eight double precision registers. The single precision registers hold 32 binary digits. The double precision registers hold 64 binary digits.

1.2.2 Program Status Word (PSW)

The PSW defines the current state of the processor. It consists of three major parts:

• Status descriptor

• Condition code

• Location counter

Individual bits and bit fields within the status descriptor portion of the PSW define the current state of interrupts and various hardware features of the processor. By setting or resetting bits within the status descriptor, the programmer can enable or disable such interrupts as I/O, arithmetic fault, and machine malfunction. On those processors with multiple sets of general purpose registers, a field in the status descriptor defines which set is currently in use. Programmers writing user level programs, as opposed to operating system or stand-alone programs, cannot directly access the status descriptor. In this case, the operating system maintains control of interrupts and registers.

The condition code provides a means of controlling program flow, based on the results of instruction execution. As certain instructions are executed, the value in the condition code changes to indicate the nature of the result. For example, if an operation produces a zero result, the condition code may be changed to a zero value. With branch instructions, the programmer can test the value in the condition code and branch or not, depending on that value. Not all instruction executions affect the condition code. See the appropriate processor reference manual for more details.

The location counter controls the order of instruction execution. Normally, the processor executes instructions sequentially and uses the location counter to keep track of where the instructions are in main memory, then fetches the instruction from the memory location specified by the address contained in the location counter. It executes that instruction, increments the location counter by the length of the instruction, and fetches the next instruction. Branch instructions, when executed, change the contents of the location counter and thereby affect the branch.

In 32-bit machines, the PSW contains 64 binary digits, the last 24 of which are reserved for the location counter. In 16-bit machines, the PSW contains 32 bits; the last 16 are reserved for the location counter.

1.2.3 I/O Interface

The execution of certain machine instructions allows the programmer to control external devices and to cause the transfer of data between external devices and main memory or registers. The actual programming of I/O operations is very much hardware dependant, both in the processor and the peripherals. I/O instructions are restricted to operating systems and stand-alone programs. User programs can communicate with I/O devices through facilities provided by the operating system.

1.2.3.1 Main Memory

To the assembly language programmer, main memory appears as a block of contiguous storage locations. The smallest unit of memory the programmer can reference is the byte (eight bits). The programmer can also reference halfwords (two bytes), fullwords (four bytes), and double words (eight bytes). Each byte in memory is referenced by a unique address. Memory addresses start with zero and are incremented by one for each succeeding byte. Memory addresses in the 32-bit processors always consist of 24 bits. In the 16-bit processors, memory addresses consist of 16 bits. When accessing bytes, any memory address within the limits of the particular hardware configuration is considered valid. Halfwords must be referenced with halfword addresses. Fullwords must be referenced with addresses that are multiples of four. Double words must be referenced with addresses that are multiples of eight.

1.2.4 Software Relocation

Programs written in CAL/32 can be absolute or relocatable. An absolute program is one whose origin (starting location) is specified at assembly time as a fixed halfword location in memory. Subsequent addresses within the program, whether referencing instructions or data, are fixed at assembly time. For execution, absolute programs always must be loaded into memory at the location specified as the origin. This type of programming is useful in stand-alone applications and some operating system situations.

Relocatable programs can be loaded for execution into any halfword location in memory. The origin of a relocatable program is assumed to be relocatable zero. The CAL/32 output for this type of program specifies all addresses in the program as relative displacements from the origin. At link time, the linkage editor resolves all addresses within the program by adding a relocation value (the actual memory address for the start of the program) to the relative addresses supplied by CAL/32. Relocation applies only to addresses within the program. Relocatable programs can contain absolute data.

1.2.5 Hardware Relocation

Some Perkin-Elmer processors and their operating systems support hardware relocation and segmentation. Programs prepared for these systems start out as relocatable. A linkage editor processes the relocatable output from CAL/32 to link in any needed subprograms. The output of this process is an absolute program that, because of the relocating hardware, can be loaded at any memory address that is a multiple of 256 for memory access controller (MAC) machines, or 2048 for memory address translator (MAT) machines. At run time, the relocating hardware adds the required relocation value to all addresses supplied by the program. This relocating hardware also provides for program segmentation, where the program is divided into pieces that can be loaded into noncontiguous blocks of memory.

CAL/32 supports segmentation by allowing the programmer to divide the program into pure and impure segments. The pure segment of a program consists of machine instructions and constant data and cannot be modified at run time. (The operating system and the hardware prevent modification.) The impure segment consists of the data base which can be modified at run time. Programs prepared as pure and impure segments can be shared (executed concurrently) by several users. Only one copy of the pure segment resides in memory during execution while there is one copy of the impure segment for each user.

1.3 INSTRUCTION FORMATS (16-BIT)

The 16-bit processors have four types of machine instructions: register-to-register, register and indexed storage, register and immediate, and short form. The following abbreviations illustrate the instruction formats:

    OP   Operation
    R1   First operand register
    R2   Second operand register
    N   A 4-bit immediate value
    X2   Second operand index register
    A2   Second operand direct address
    I2   Second operand immediate value

Most instructions require two operands, the first of which is contained in a register. The result usually replaces the contents of the first operand register. Exceptions to these rules are noted in section 1.5.

1.3.1 Register-to-Register (RR) Instructions

Register-to-register instructions cause operations to take place between operands contained in registers. Register-to-register instructions are 16 bits long, as shown in Figure 1-1.

```
---------------------------------------------------------------------
|            OP                  |      R1       |      R2          |
---------------------------------------------------------------------
Bits:
0                               7 8            11 12              15
```

Figure 1-1   RR Format (16-Bit)

The first eight bits of the instruction define the operation. The next four bits identify the first operand register. The final four bits identify the second operand register. In most register-to-register instructions, the specified operation takes place between the contents of the first operand register and the contents of the second operand register. The result of the operation replaces the contents of the first operand register.

1.3.2 Register and Indexed Storage (RX) Instructions

Register and indexed storage instructions cause an operation to take place between a first operand, contained in a register, and a second operand, located in main memory. These instructions require 32 bits, as shown in Figure 1-2.

```
-----------------------------------------------------------------/ /---
|              OP               |    R1    |     X2     | I2    /  /     |
-----------------------------------------------------------------/ /---
Bits:
0                              7 8       11 12        15 16           31
```

Figure 1-2 RX Format (16-Bit)

The first eight bits define the operation. The next four bits identify the first operand register, and the next four bits identify an optional index register. The remaining 16 bits specify an address in main memory. The operation takes place between the contents of the first operand register and the contents of the memory location specified. The actual address of the second operand is determined by adding the contents of the index register to the contents of the address field. If the index field of the instruction contains zero, no indexing takes place. In most cases, the result of the operation replaces the contents of the first operand register.

1.3.3 Register and Immediate (RI) Instructions

These instructions cause operations to take place between the contents of a register and the contents of an immediate field imbedded in the instruction itself. They are 32 bits long, and are shown in Figure 1-3.

```
-----------------------------------------------------------------/ /---
|              OP               |    R1    |     X2     | I2   /  /     |
-----------------------------------------------------------------/ /---
Bits:
0                              7 8       11 12        15 16           31
```

Figure 1-3 RI Format (16-Bit)

The first eight bits specify the operation. The next four bits identify the first operand register. The next four bits identify an optional index register. The final 16 bits are the immediate value. The first operand is the contents of the first operand register. The second operand is obtained by adding the contents of the index register to the contents of the immediate field. If the index field contains zero, no addition takes place. The result of the operation usually replaces the contents of the first operand register.

1.3.4 Short Form (SF) Instructions

These instructions are variations on the register and immediate instructions in which the second operand is small enough to be expressed in four bits. Short form instructions require 16 bits, as shown in Figure 1-4.


```
 ------------------------------------------------------------------
|           OP               |     R1      |        N           |
 ------------------------------------------------------------------
Bits:
0                           7 8          11 12                 15
```


Figure 1-4 SF Format (16-Bit)

The first eight bits indicate the operation. The next four bits identify the first operand register. The next four bits contain the immediate value. Operations take place between the contents of the first operand register. The result of the operation usually replaces the contents of the first operand register.

1.4 INSTRUCTION FORMATS (32-BIT)

The 32-bit processors recognize seven different types of instructions. These are: register-to-register, three variations on register and indexed storage, two variations on register and immediate, and short form. The following abbreviations are used to illustrate instruction formats:

    OP   Operation
    R1   First operand register
    R2   Second operand register
    N   A 4-bit immediate value
    X2   Second operand single index register
    D2   Second operand displacement
    FX2   Second operand first index register
    SX2   Second operand second index register
    A2   Second operand direct address
    I2   Second operand immediate value

Most instructions require two operands, of which the first is the contents of a register. The result of the operation usually replaces the contents of the first operand register. Exceptions to these rules are noted in Section 1.5.

1.4.1 Register-to-Register (RR) Instructions

The format and function of these instructions are the same as for the 16-bit processors. They cause operations to take place between operands contained in registers, and they require 16 bits. These instructions are shown in Figure 1-5.

```
---------------------------------------------------------------------
|             OP                 |      R1       |      R2        |
---------------------------------------------------------------------
Bits:
0                                7 8            11 12            15
```

Figure 1-5 RR Format (32-Bit)

The first eight bits specify the operation. The next four bits identify the first operand register, and the last four bits identify the second operand register. The processor performs the indicated operation between the contents of the first operand register and the contents of the second operand register. In most RR instructions, the result replaces the contents of the first operand register.

1.4.2   Register and Indexed Storage One (RX1) Instructions

These instructions define an operation between the contents of a register and the contents of a main memory location. They require 32 bits, as shown in Figure 1-6.

```
-------------------------------------------------------------------/ /-----
|             OP             |   R1   |   X2    | 0 | 0 | D2     / /        |
-------------------------------------------------------------------/ /-----
Bits:
0                           7 8      11 12     15 16  17  18              31
```
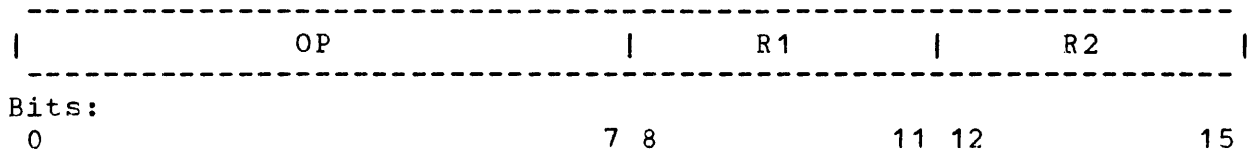
Figure 1-6 RX1 Format (32-Bit)

The first eight bits define the operation. The next four bits
identify the first operand register, and the next four bits
identify the index register. The next two bits, bits 16 and 17,
must be zeros. The next 14 bits constitute a direct program
address in a range from 0 to 16383.

The address of the second operand is obtained by adding the
contents of the index register to the contents of the 14-bit
address field. If the index register field contains zero, this
addition does not take place, and the contents of the address
field are used as the address. The operation takes place between
the contents of the first operand register and the contents of
the specified memory location. The result usually replaces the
contents of the first operand register.

1.4.3 Register and Indexed Storage Two (RX2) Instructions

These instructions define operations between the contents of a
register and the contents of a location in main memory. RX2
instructions are like the RX1 instructions; they require 32 bits.
They differ from the RX1 instructions in the method of
calculating the second operand address. See Figure 1-7.

```
-------------------------------------------------------------------5 5---
|            OP              |     R1     |     X2     | 1 |A2          |
-------------------------------------------------------------------5 5---
Bits:
0                           7 8         11 12        15 16 17       31
```
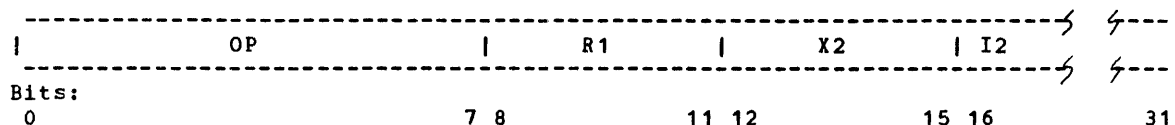
Figure 1-7 RX2 Format (32-Bit)

The first eight bits define the operation. The next four bits
identify the first operand register, and the next four bits
identify the index register. The next bit, bit 16, must be a
one. The remaining 15 bits are treated as a signed integer in
two's complement notation. Bit 17 is the sign bit which, if one,
indicates a negative quantity, and if zero, indicates a positive
quantity.

The address of the second operand is obtained in two steps.

1. The signed integer, with sign bit extended to produce a
    32-bit integer, is added to the contents of the index
    register.

2. This intermediate result is added to the value in the
    incremented location counter. The result is truncated to 24
    bits.

If the index register field is zero, the first addition does not take place. The indicated operation takes place between the contents of the first operand register and the contents of the specified memory location. The result usually replaces the contents of the first operand register.

1.4.4 Register and Indexed Storage Three (RX3) Instructions

These instructions are analogous to the RX instructions in the 16-bit processors. They call for operations between the contents of a register and the contents of an indexed memory location and require 48 bits. See Figure 1-8.

```
------------------------------------------------------------------------/ /---
|        OP        |   R1   |   FX2   | 0 | 1 | 0 | 0 |  SX2   | A2   /  /    |
------------------------------------------------------------------------/ /---
Bits:
0                 7 8     11 12    15 16  17  18  19  20      23 24      47
```
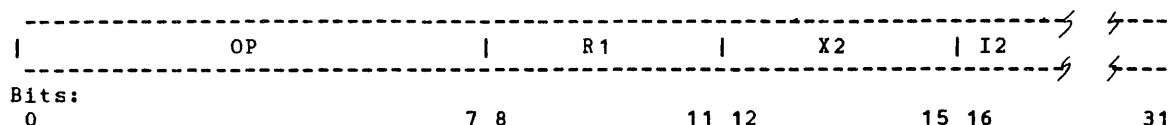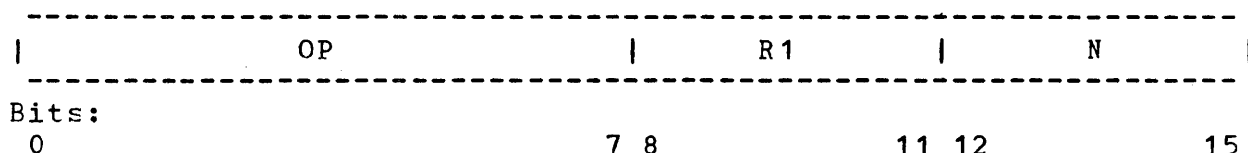
Figure 1-8   RX3 Format (32-Bit)

The first eight bits specify the operation. The next four bits identify the first operand register, and the next four bits identify the first index register. Bit 16 must be zero. Bit 17 must be one. Bits 18 and 19 must be zero. The next four bits identify the second index register. The next 24 bits contain a direct memory address.

The address of the second operand is obtained by adding the contents of the first index register to the contents of the second index register. This intermediate result is then added to the contents of the direct address field, and the final result is truncated to 24 bits.

If either of the index register fields contains zero, that level of indexing does not take place. If both are zero, no indexing takes place. In most RX3 instructions, the operation takes place between the contents of the first operand register and the contents of the specified memory location. The result usually replaces the contents of the first operand register.

1.4.5 Register and Immediate One (RI1) Instructions

These instructions are similar to the RI instructions in the 16-bit processors. They specify operations that take place between the contents of a register and the contents of a field that is part of the instruction. They require 32 bits, as shown in Figure 1-9.

```
 ----------------------------------------------------------------------/ /---
|         OP              |    R1     |     X2      | I2              / /---  |
 ----------------------------------------------------------------------/ /---'
Bits:
0                        7 8        11 12        15 16              31
```
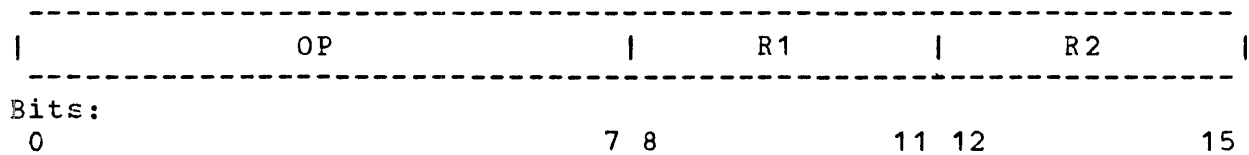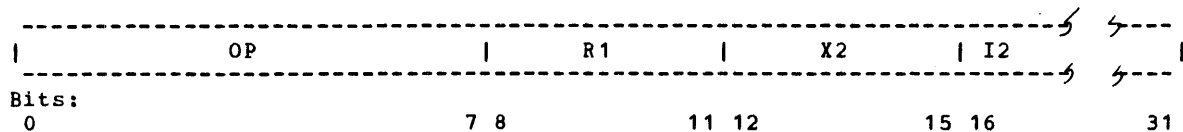
Figure 1-9 RI1 Format (32-Bit)

The first eight bits indicate the operation. The next four bits
identify the first operand register, and the next four bits
identify an index register. The second operand is obtained by
extending the contents of the immediate field to 32 bits, by
propagating the sign bit, and then adding this quantity to the
contents of the index register. If the index register field is
zero, no addition takes place, and the extended immediate value
is the second operand.   The operation takes place between the
contents of the first operand register and the immediate value.
The result usually replaces the contents of the first operand
register.

1.4.6 Register and Immediate Two (RI2) Instructions

These instructions are similar to the RI1 instructions, except
that the field contains a 32-bit value, and the instruction
itself requires 48 bits. See Figure 1-10.

```
 ----------------------------------------------------/ /--------/ /---
|         OP              |    R1     |     X2      | I2/ /        |  / /--- |
 ----------------------------------------------------/ /--------/ /---'
Bits:
0                        7 8        11 12        15 16        23 24       47
```
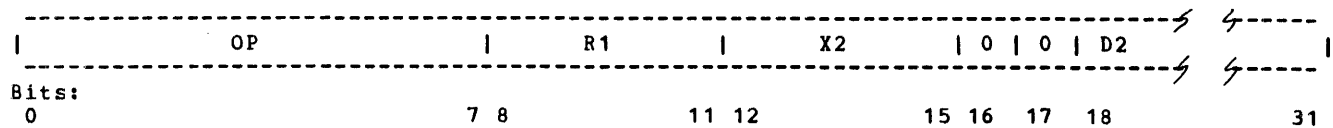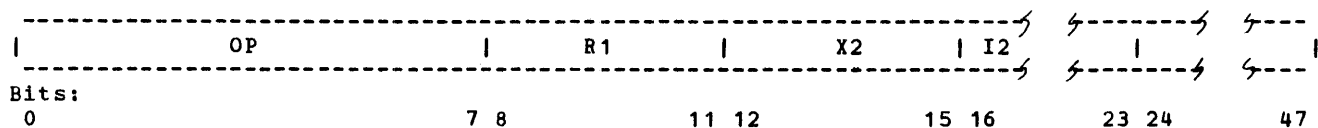
Figure 1-10 RI2 Format (32-Bit)

The first eight bits define the operation. The next four bits
identify the first operand register. The next four bits identify
the index register. The second operand is obtained by adding the
contents of the index register to the contents of the immediate
field.   If the index register field is zero, no addition takes
place, and the immediate value is the second operand.   The
operation takes place between the contents of the first operand
register and the immediate value. The result usually replaces
the contents of the first operand register.

1.4.7 Short Form (SF) Instructions

Short form instructions are similar to the SF instructions in the
16-bit processors. They specify operations between the contents
of a register and the contents of an immediate field, whose value
is small enough to be expressed in four bits. These instructions
require 16 bits, as shown in Figure 1-11.

```
-------------------------------------------------------------------
|              OP               |      R1      |      N          |
-------------------------------------------------------------------
Bits:
0                              7 8           11 12              15
```
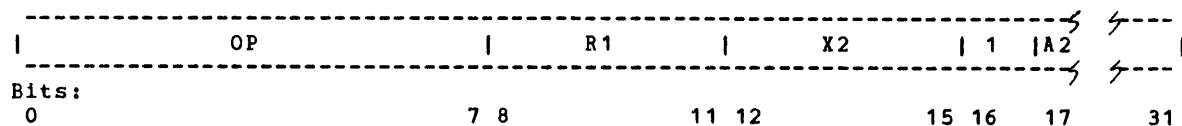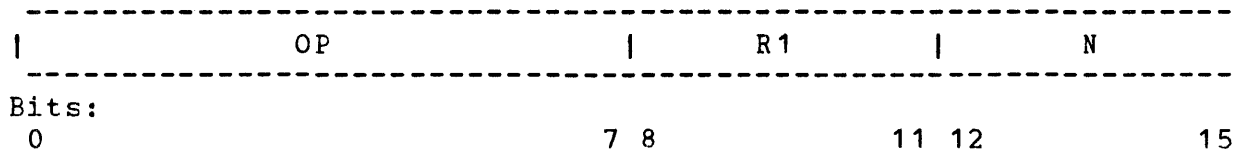
Figure 1-11 SF Format (32-Bit)

The first eight bits define the operation. The next four bits
identify the first operand register. The next four bits are the
immediate field. The operation then takes place between this
value and the contents of the first operand register. The result
usually replaces the contents of the first operand register.

1.4.8 Register and Indexed Storage/Register and Indexed
       Storage (RXRX) Instructions

Register and indexed storage/register and indexed storage (RXRX)
instructions resemble a pair of adjacent RX instructions, but
represent one cohesive string-processing instruction.   An RXRX
instruction is comprised of two instruction members. Each member
can be any one of the RX1, RX2, or RX3 machine formats,
independent of the other member's format. For example, the first
instruction member might be of the RX1 format, and the second
instruction member might be of the RX3 format, yielding a 10-byte
RXRX instruction.   Thus, an RXRX instruction length might range
from 8, 10, or 12 bytes.

The first eight bits of the first instruction member, OP, specify
the operation class. The particular RXRX operation is specified
by the contents of the Operation-Modifier (OP-MOD) field in the
first eight bits of the second instruction member.   OP-MOD is
actually generated by the assembler according to the specific
RXRX operation mnemonic and the R1/L1 or R2/L2 fields programmed
by the user in source code. Refer to Figure 1-12.

Figure 1-12 diagram:

```
|------------------ FIRST MEMBER ------------------|      |------------------ SECOND MEMBER ------------------|
|                  4 TO 6 BYTES                    |      |                  4 TO 6 BYTES                     |

0      7  8   11 12                     31/47|0          7  8   11 12                          31/47
                         OPN1                         OP-MOD                     OPN2
+--------+------+------+--+--+------------+         +--------+---+---+----+--------+------+------+-----+-------------+
|        | R1/  |      |  |  |            |         |     F  | R2/|    |       |      |      |     |             |
|   OP   | /-L1 |  X2  |0 |0 |      D     |         | I   U  | /-L2| FX2| 0100  | SX2  |      |  A  |             |
|        |      +------+--+--+------------+         +--------+---+---+----+        |      |      |     |             |
|        |      |  X2  |1 |    D2         |         | L L C  N                                                    |
|        |      |      |  |               |         | 1 2   C                                                    |

0             12-15 16 17              31|0  1 2 3      7   12-15 16-19 20-23 24      -              47

|------------ RX1/RX2 SAMPLE MEMBER -----------|      |------------ RX3 SAMPLE MEMBER -----------|
                              8, 10, 12 BYTES
```
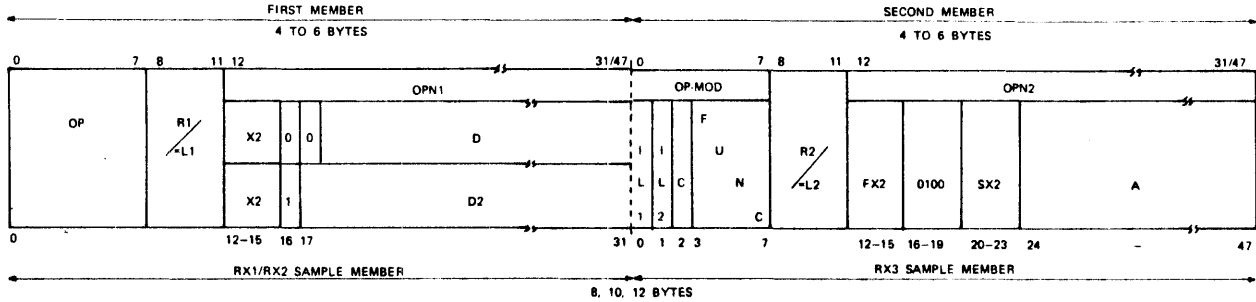
Figure 1-12 RXRX Format (32-Bit)

The next four bits in the first instruction member, R1/L1, identify either R1, the string's length-specifying register, or L1, the string's actual length. The user specifies to the assembler whether the value in the R1/L1 field is a register or an immediate value.

The R1/L1 field is assumed to be a register, unless an equal sign (=) precedes the L1 source expression. In machine format, the IL1 field is set when the =I1 source field specifies an immediate value as length. The IL2 field in machine format is reset when the R1 field is used to specify a register that contains the string's length. When the length is an immediate value, its value may range from 0 through 15. When the length is in a register, the register may contain a length that ranges from 0 through 224-1. A length of 0 indicates a null string.

The remaining bits, bits 12 through 31 or 12 through 47, of the first instruction member, OPN1, contain the address/location of the lowest addressable byte of a string or its storage location. The field, OPN1, is then similar to the indexed address portion of an RX1, RX2, or RX3 machine format. See OPN2 below.

The first eight bits of the second instruction member, OP-MOD, are an operation-modifier field containing OPN1's length indicator, IL1, in bit 0; OPN2"s length indicator, IL2, in bit 1; a special circumstances bit, C, in bit 2; and in bits 3 through 7, FUNC, the specific function code of the general operation class, OP. As described above, IL1 and IL2 are determined by the assembler. The special circumstances bit, C, and function code, FUNC, are determined by the assembler from the operation-mnemonic. The C bit is used by some RXRX instructions to indicate that the result of the operation will be forced positive.

The next four bits, bits 8 through 11, of the second instruction member, R2/L2, identify either R2, this string's length-specifying register; or L2, the string's actual length. Again, the user specifies in source format to the assembler whether the value in the R2/L2 field is a register or an immediate value. The R2/L2 source format field is assumed to be a register, unless an equal sign (=) precedes the L2 source expression. In machine format, IL2 is set when the =L2 field is used to specify an immediate value. IL2 is reset when R2 is used to specify a register. When the length is an immediate value, expressed as =L2, its value may range from 0 through 15. When the length is in a register, its value may range from 0 through $2^{24}-1$. A zero length indicates a null string.

The remaining bits, bits 12 through 31 or 12 through 47, of the second instruction member, OPN2, contain the address/location of the lowest addressable byte of a second member's string. Both OPN1 and OPN2 are similar in format to the indexed address portion of an RX1, RX2, or RX3 machine format. The particular format of either OPN1 or OPN2 is selectively generated by the assembler, independently according to the user source program.

In RX1 machine format, bits 16 and 17 are zero. Bits 12 through 15 identify the index register, X2, the contents of which are added to the absolute 14-bit value, D, to formulate the string's address.

In RX2 machine format, bit 16 is set. Bits 12 through 15 identify the index register, X2, the contents of which are added to the 15-bit displacement value, D2, to formulate the string's address.

In RX3 machine format, bits 16 through 19 are 0100 binary. Bits 12 through 15 identify the first index register, FX2; and bits 20 through 23 identify the second index register, SX2. The contents of both are added to the 24-bit address value, A, to formulate the string's address.

NOTES

1. When the first member's OPN1 represents the string's address in RX2 format, the displacement value, D2, is relative to the end address of the first instruction member, not to the end of the full RXRX instruction.

2. When the second member's OPN2 represents the string's address in RX2 format, the displacement value is relative to the end of the second instruction member, which is also the end of the full RXRX instruction.

1.5 VARIATIONS ON INSTRUCTION FORMATS

Not all instructions follow the above instruction formats. In some instructions the fields are redefined. Some instructions do not require two operands. Some instructions do not change the first operand, some instructions change the second operand, and some instructions change neither operand.

1.5.1 Conditional Branch Instructions

Conditional branch instructions use formats that resemble register-to-register, register and indexed storage, and short form instructions. However, the interpretation of the fields differs from the standard, as does the actual operation. In all conditional branch instructions, the first operand identification is interpreted as a mask that is ANDed with the condition code. If the result of this test indicates that the branch is to be taken, then the second operand address is the location to which the processor must go to obtain the next instruction.

In the RR instructions, the second operand register contains the branch address. In the RX instructions, the branch address is obtained by one of the standard methods for obtaining second operand addresses. In the SF instructions, the immediate field is interpreted as a halfword displacement, either forward or backward, from the current location counter. The branch address is obtained by adding or subtracting this quantity from the current location counter.

1.5.2 Branch and Link Instructions

These instructions facilitate branching to and returning from subroutines. They use formats similar to RR and RX where the first operand register is the link register. Before the branch is taken, the address of the next memory location following the branch instruction is placed in this register. In the RR instructions, the branch location is the contents of the second operand register. In the RX instruction, the branch address is obtained by one of the usual methods for obtaining second operand addresses.

1.5.3 Other Variations

Some instructions change the second operand rather than the first. Most notable among these are the store instructions and the instructions that add the contents of a register to the contents of a memory location.

Test instructions and compare instructions change neither operand. The indicated operation takes place between the two operands, but neither is changed. The result of the operation is indicated by the condition code.

Certain other instructions, such as load PSW and simulate interrupt, do not always require a first operand. In addition, all of the I/O instructions do not follow the general rules. For detailed information on how these and other anomalous instructions work, refer to the appropriate processor reference manual.

CHAPTER 2
SYMBOLIC REPRESENTATION

2.1 INTRODUCTION

When writing assembly language programs, the programmer uses meaningful symbols to represent the binary language interpreted by both Common Assembly Language/32 (CAL/32) and the processor. Symbols consist of printable ASCII characters, either singly or in combination. CAL/32 recognizes the complete set of printable ASCII characters. However, depending on the context, there can be restrictions on the use of the complete set. See Chapter 3.

2.2 SYMBOLS AND EXPRESSIONS

Symbols represent addresses, register identifiers, absolute values, operation identifiers, and constants. Examples of symbols are:


```
A
LOOP
BXLE
PART1
REG5
16
```


Symbols can be combined to form expressions. The arithmetic operators: plus, minus, multiply, divide are represented in CAL/32 by the symbols: +, -, *, /. They combine with other symbols to form arithmetic expressions. Examples of these arithmetic expressions are:


```
A+B
LAST-FIRST*TWO
A-16
```


Blanks and parentheses are not permitted within an expression. For example, the sequence:


```
A - B * (C + D)
```

would not be interpreted by CAL/32 as an expression. Depending on the context, CAL/32 might misinterpret the symbols, generate incorrect code, and fail to detect the error. Where CAL/32 can recognize the error, it generates an error message.

The evaluation of expressions takes place from left to right with no rules of precedence. Thus, CAL/32 evaluates the expression:

    LAST-FIRST*TWO

by subtracting the value of First from the value of Last, and multiplying this result by the value of Two.

Logical expressions consist of symbols joined by the logical operators AND and inclusive OR. They are represented in CAL/32 by the symbols & and !. Examples of logical expressions are:

    X&Y!A
    CHAR&NULL

Logical expressions are evaluated from left to right with no rules of precedence. Blanks and parentheses are not permitted in logical expressions.

Mixed expressions are formed by combining logical and arithmetic operators. For example:

    A-B!TWO

CAL/32 evaluates this expression by first subtracting the value of B from the value of A, and then ORing the result with the value of Two. Mixed expressions are like arithmetic and logical expressions in that blanks and parentheses are not allowed, and the method of evaluation is from left to right with no rules of precedence.

Symbols represent either absolute or relocatable quantities. At assembly time, relocatable quantities have a value equal to their displacement from some fixed point within the program, usually but not necessarily, the origin or starting location. At load time, the relocatable quantity is replaced by an absolute quantity whose value is calculated by adding the relocation value to the relocatable quantity. Absolute quantities are known to the assembler at assembly time and are not changed at load time.

The operations: multiply, divide, AND, and OR are permitted only between absolute data. The plus and minus operators can be used on mixed data. The results of such operations are:

```
          CPERATION                    RESULT

     Absolute + Absolute          Absolute
     Absolute - Absolute          Absolute
     Relocatable + Relocatable    Invalid
     Relocatable - Relocatable    Absolute
     Relocatable + Absolute       Relocatable
     Relocatable - Absolute       Relocatable
     Absolute + Relocatable       Relocatable
     Absolute - Relocatable       Invalid
```

2.3 SYMBOLS AND THEIR VALUES

By definition, certain symbols used in CAL/32 programming have
implicit values; that is, the value of the symbol is determined
by the way in which it is expressed and used. Examples of this
kind of symbol are the decimal, hexadecimal, and character
symbols used as operands in instructions. There are also global
symbols in CAL/32. These symbols have preset values that cannot
be redefined by the programmer. The programmer also can define
the value of a symbol explicitly by using the equate statement.
This section covers the use of implicit and global symbols.
Chapter 3 covers the explicit use and definition of symbols.

2.3.1 Implicit Symbols

When used in the correct context, a string of decimal digits is
automatically assigned the actual value of the number represented
by the string. For example, the expression:

    A+14

has a value that the assembler determines by adding the quantity
14 to the value A, which must be defined by some other means.

CAL/32 also recognizes the implicit value of special character
strings the programmer uses to represent decimal, hexadecimal,
and character values. These strings are made up of a single
letter that indicates the particular type, followed by a group of
characters enclosed in apostrophes that represents the value.
The code characters are:


```
CODE CHARACTER              TYPE

     H            Halfword decimal
     F            Fullword decimal
     X            Halfword hexadecimal
     Y            Fullword hexadecimal
     C            Character
```

Decimal numbers consist of an optional sign (+ or -) followed by decimal digits representing the actual value. Commas are not allowed in the representation. Halfword decimal values can be represented by from 1 to 5 decimal digits, with a range from -32,768 to +32,767. Fullword values can be represented by from 1 to 10 decimal digits, with a range from -2,147,483,648 to +2,147,483,647. CAL/32 converts these decimal numbers into two's complement binary integers. Examples of decimal symbols, with their internal representation expressed in hexadecimal notation are:

```
      SYMBOL          VALUE

      H'125'          007D
      H'32765'        7FFD
      H'+32765'       7FFD
      H'-15'          FFF1
      F'123123'       0001 E0F3
      F'1'            0000 0001
      F'-2'           FFFF FFFE
```

Hexadecimal symbols consist of the X or Y type code followed by a string of hexadecimal digits enclosed in apostrophes. Halfword symbols can use from one to four digits. Fullword symbols can use from one to eight digits. Leading zeros are not required, and the value is right justified. Examples of hexadecimal symbols are:

```
      SYMBOL          VALUE

      X'F'            000F
      X'D4E'          0D4E
      Y'030'          0000 0030
      Y'A'            0000 000A
      Y'0'            0000 0000
```

Character symbols consist of from one to four ASCII characters enclosed in apostrophes and preceded by the type code C. Characters are right justified, with zero fill. Depending on the context, either a halfword or a fullword results. Examples of character symbols are:

```
      SYMBOL          VALUE          VALUE
                      (HALFWORD)     (FULLWORD)

      C'*'            002A           0000  002A
      C'12'           3132           0000  3132
      C'AB'           4142           0000  4142
      C'1234'         3334           3132  3334
```

In the last example, where a halfword value was generated, only the rightmost two characters were used. Where the context dictates a halfword, and a longer string is used, a truncation error results. One final type of implicit assignment occurs in the use of symbols as statement identifiers. Where a symbol is used in the name field of a statement, it is automatically assigned a value equal to the value of the current location counter. This type of assignment is covered in Chapter 4.

2.3.2 Global Symbols

Six symbols recognized by CAL/32 have predetermined values. They are:


```
ADC
LADC
PURETOP
IMPTOP
ABSTOP
*
```


The use of these symbols is somewhat restricted, and they cannot be redefined by the programmer.

In programs written for 32-bit processors, the address length constant (ADC) always has a value of 4, the length of an address constant in bytes. (In 32-bit processors, addresses must be contained in fullwords, even though the actual address is only 24 bits in length.) In programs for which CAL/32 is to generate 16-bit code, ADC has the value of 2. In programs written for 32-bit processors, the log (base 2) of the address length constant (LADC) always has a value of 2. In programs for 16-bit processors, LADC always has a value of 1. Both of these symbols, ADC and LADC, are used most frequently in common mode programming. See Chapter 4.

The symbols PURETOP, IMPTOP, and ABSTOP have values equal to:


```
PURETOP    The next available location in the pure segment
IMPTOP     The next available location in the impure segment
ABSTOP     The next available location in the absolute segment
```


Because these values change during assembly, the symbols must be used carefully. They can be used as second operand identifiers in machine instructions and as operands in assembler instructions where they are treated as address values. They cannot be used in assembler instructions that control the location counter.

The asterisk symbol (*), used as an operand rather than as an operator in an expression, always has a value equal to the value of the current location counter. Throughout the assembly process, CAL/32 maintains a location counter analogous to the hardware location counter contained in the central processing unit (CPU). Depending on the organization of the program, this location counter can contain any one of several values. For 32-bit programs the location counter may point to the current location in the absolute segment, the pure segment, or the impure segment. For 16-bit assemblies, the location counter may point to the current absolute location or the current relocatable location.

CHAPTER 3
THE SOURCE PROGRAM

3.1 INTRODUCTION

The source program consists of a set of assembly language
statements that specify the operations to be performed by the
processor, define the constants and storage areas used by the
program, and control the assembly process to produce the desired
output. Source statements for Common Assembly Language/32
(CAL/32) are of two types: comment statements and instruction
statements. Instruction statements are further divided into
machine instructions and assembler instructions. Each statement
consists of an 80-character record, in which symbols and
expressions identify the statement, and where necessary, indicate
the operation and locate the operands.

3.2 COMMENT STATEMENTS

Comment statements can appear anywhere in the source program.
They allow the programmer to include easy-to-read documentation
in the source program listing. They produce no object code. The
assembler does not process comment statements except to check for
proper sequencing and scan for invalid characters.

Comment statements must always start with an asterisk (*) in the
first character position. Positions 2 through 71 can contain any
printable ASCII character, including lowercase alphabetic
characters. Blanks are considered to be "printable" characters.
If a nonprintable character turns up in a comment statement,
CAL/32 replaces it with a pound sign (#). Position 72 of a
comment statement must contain a blank character. Positions 73
through 80 can, at the programmer's option, be used for sequence
identification. The sequence field can contain any printable
ASCII character other than lowercase alphabetic characters.
Where sequence checking is requested, each successive sequence
identifier must be greater, in the ASCII collating sequence, than
the previous identifier. Examples of comments are:


```
POSITION
1                                              72 73
*   THIS IS A COMMENT
*   IT MAY APPEAR ANYWHERE IN THE PROGRAM
*   SUBROUTINE GETCHAR                           GET10000
*MOVES A CHARACTER FROM THE INPUT BUFFER         GET10010
*AND RETURNS IT IN GENERAL REGISTER THREE        GET10020
```

3.3 INSTRUCTION STATEMENTS

Instruction statements can be written in fixed format or in free
format. For either format, there are five distinct fields in
each statement. In fixed format, these fields are:

CHARACTER POSITIONS	DEFINITION
1 through 8	Name field
10 through 14	Operation field
16 through n	Operand field
n+2 through 71	Comment field
73 through 80	Sequence field

Positions 9, 15, and 72 always must contain blank characters.
The operand field and the comment field are variable in length,
and the first blank character after position 16 serves as a
delimiter between the operand field and the comment field.
Because of the way the output listing is tabulated, the comment
field cannot contain more than 37 characters. If more than 37
characters appear, only the first 37 are printed on the output
listing.

CAL/32 does not require source statements to be written in fixed
format. It accepts free format source, in which blank characters
serve as delimiters. If, for example, the name field is not
used, a blank character in the first position indicates that the
next nonblank character is the start of the operation field.
Similarly, if the operaticn field requires fewer than five
characters, the first blank character following the operation
code indicates that the next nonblank character is the first
character cf the operand field. As in the fixed format
statement, the first blank character following the operand field
indicates the end of that field and the beginning of the comment
field. There are three restrictions on the use of free format:

1. Comment length is limited to 37 characters, including blanks.

2. Position 72 must contain a blank character.

3. The sequence field must start in position 73.

The last restriction is because CAL/32 cannot distinquish between
a blank character as part of a comment and a blank character
intended to separate the comment from the sequence field.

If there are no nonblank characters in positions 1 through 20,
CAL/32 assumes that the statement is a comment and lists it as
such, with a warning note. If more than 15 blanks separate the
name field from the operation field, CAL/32 assumes that the

operation field is not present. Similarly, if more than 15
blanks separate the operation field from the operand field,
CAL/32 assumes that the operand field is not present. In both
cases, CAL/32 generates an error message.

When writing CAL/32 instruction statements, the programmer uses
symbolic representation in the name field, the operation field,
and the operand field. The following paragraphs describe the use
of symbols and expressions in these fields.

3.4 NAME FIELD

Where a symbol appears in the name field, it represents the value
of the current location counter for that particular instruction.
This allows the programmer to refer to specific locations
symbolically, without having to know the actual value of the
location counter. The following five restrictions apply to the
formation of names:

1. The first character of a name must be an uppercase or
    lowercase alphabetic character or one of the special
    characters:

    - at sign (@)

    - dot (.)

    - dollar sign ($)

    - underscore (_)

                           NOTE

           Lowercase   letters   are   internally
           converted to uppercase except in string
           constants.

2. The remaining characters can be made up of any combination of
    valid first characters, plus the numeric characters 0 through
    9.

3. The name must consist of from one to eight characters.

4. The name must start in the first character position of the
    source record.

5. Imbedded blanks are nct permitted.

Examples of valid names are:

    LABEL
    LOOP1
    .SIN
    @GOTO
    $$GET5

Examples of incorrect names are:

    1LOOP   First character is numeric
    LOOPCOUNTER   More than eight characters
    AB?C   Question mark is illegal

As a general rule, a given symbolic string can appear only once
in the program where it defines a location. That is, the same
symbol may not appear in the name field of more than one
instruction. The exception to this is the equate instruction.
This is covered in the section on assembler instructions.

3.5 OPERATION FIELD

The use of symbols in the operation field is severely restricted.
Only previously defined symbols can appear in this field. The
symbols that appear in the operation field are called mnemonics;
they represent operations to be performed by the processor at run
time, or operations to be performed by the assembler. CAL/32
recognizes mnemonics that represent all machine operations for
all Perkin-Elmer processors. It also recognizes a large set of
assembler mnemonics that allows the programmer to control the
assembly process.

Mnemonics can consist of no more than five characters. They are
formed in the same way as names and use the same character set.
CAL/32 permits users to define mnemonics. This process is
described in the section that deals with the equate instruction.
Specific mnemonics that define machine operations and assembler
operations are described later in this chapter. Examples of
operation mnemonics are:

    MNEMONIC   TYPE   MEANING

       AR   Machine   Add register
       S   Machine   Subtract
       CLI   Machine   Compare logical immediate
       ORG   Assembler   Set location counter

3.6 OPERAND FIELD

CAL/32 permits the use of both symbols and expressions in the operand field of instructions. Symbols used in the operand field can be implicitly defined or can be explicitly defined. The rules for forming operands for assembler instructions vary from instruction to instruction, and each is described individually later in this chapter.

Most machine instructions require two operands while some require only one. Where two operands are required, the first is separated from the second by a comma. Following are the general rules for forming operands for machine instructions.

3.6.1 Register-to-Register Instructions

Both the first and the second operand must be represented by symbols or expressions with values between 0 and 15 inclusive. If the value is greater than 15 or less than 0, the assembler sets it to 0, and generates an error message. For example, if the symbols 1 and 2 appear in the operand field of the add register instruction:

        AR   1,2

CAL/32 generates the machine code to add the contents of register 2 to the contents of register 1 and store the result in register 1. The use of 1 and 2 here is an example of how decimal numbers have an implicit value when used in the proper context. Another example:

        AR   X'1',X'2'

shows how hexadecimal symbols can be used as register identifiers. This is an exception to the previously stated rule that hexadecimal symbols generate halfword or fullword values. Where used as register identifiers, decimal, hexadecimal, and character symbols cause the assembler to generate 4-bit values.

Expressions can be used in identifying registers, as in:

        AR   A+B,C'A'-X'40'

where CAL/32 evaluates the expressions and uses the results as the register identifiers. This is not a universally useful feature of the language, although it has some applications in common mode programming.

A more useful way to identify registers is to use explicitly defined symbols. Suppose the symbols SUM and INC are defined to have values of 1 and 2, respectively. Then the instruction:

```
AR      SUM,INC
```

has the same effect as:

```
AR      1,2
```

yet the notation is more meaningful because registers 1 and 2 are identified as the SUM and INC registers.

3.6.2 Register and Indexed Storage Instructions

If the first operand is required, it must be a valid register identifier as described for register-to-register instructions. The second operand, separated from the first by a comma, can be:

● a symbol,

● an expression, or

● a symbol or an expression followed by an index register identifier enclosed in parentheses.

Where indexing is used, identification of the registers follows the same rules as those for specifying first or second operand registers. In double-indexed instructions, the first and second index identifiers are separated by a comma. An example of how register and indexed storage instructions are written is:

```
S       1,A
```

where the first operand is the contents of general register 1, and the second operand is the value at location A in memory. Another example:

```
S       SUM,TABLE(PTR)
```

shows how single indexing is expressed. In this case, the first operand is the value contained in the register identified by the symbol SUM, and the second operand is the value at memory location table plus the contents of the index register PIR.

Another example:

```
S       SUM,LAST-FIRST(BASE,PTR)
```

shows the use of double indexing along with the use of an expression in the operand field. A final example:

```
S       SUM,O(ADDR)
```

illustrates where an address of a second operand is contained in the index register. Here, there must be a symbol in the address field even if it is equal to zero.

3.6.3 Register and Immediate Instructions

The first operand must be specified by a valid register identifier. The second operand can be:

● a symbol,

● an expression, or

● a symbol or an expression followed by an index register identifier enclosed in parentheses.

Example:

```
CLI     STRNG,C'ABCD'
```

causes the character string ABCD, represented internally as the fullword character value 4142 4344, to be compared with the contents of the register identified by the symbol STRNG. In another example:

```
CLI     ADDR,LAST-FIRST(PTR)
```

the expression LAST-FIRST is evaluated by CAL/32 at assembly time. At run time this value is added to the contents of the index register before the comparison takes place. In another example:

```
CLI     ADDR,Y'2000'(PTR)
```

the fullword, hexadecimal quantity 0000 2000, is added to the contents of the index register. The result is then compared with the contents of the register identified by the symbol ADDR.

3.6.4 Register and Indexed Storage/Register and Indexed Storage (RXRX) Instructions

The RXRX instructions have four basic source operand fields, each of which is separated from the other by a comma. The first operand field can be:

- a valid register identifier, symbol, or expression with a defined absolute value in the range 0 to 15

- an equal sign (=) preceding a symbol or an expression with a defined absolute value in the range 0 to 15

The second source operand field, separated from the first by a comma, can be:

- a symbol or an expression

- a symbol or an expression, optionally followed by an index register identifier enclosed in parentheses

- a symbol or an expression, optionally followed by a pair of index register identifiers, separated by a comma, with the pair enclosed in parentheses

The third source operand field, separated from the second by a comma, can be:

- a valid register identifier, symbol, or expression with a defined absolute value in the range 0 to 15

- an equal sign (=) preceding a symbol or an expression with a defined absolute value in the range 0 to 15

The fourth source operand field, separated from the third by a comma, can be:

- a symbol or an expression

- a symbol or an expression, optionally followed by an index register identifier enclosed in parentheses

- a symbol or an expression, optionally followed by a pair of index register identifiers, separated by a comma, with the pair enclosed in parentheses

Examples of how these instructions are written are:

    MOVE =LENGTH2,HERE,=LENGTH1,THERE

which moves the string of length, LENGTH1, at location THERE to
the location HERE up to the number of bytes indicated by LENGTH2.
If LENGTH1 is less than LENGTH2, this instruction pads the extra
bytes with the right-justified character in general register
zero.

In the preceding example, the first operand field is the
immediate value of symbol LENGTH2. The equal sign that specifies
LENGTH2's value is an immediate value and not a register
identifier. The second operand field is the storage address at
location HERE. The third operand field is the immediate value of
symbol LENGTH1 (its immediacy is again indicated by the equal
sign). The fourth operand field is the string at location THERE.

Another example is:

    MOVEP R7,PRINTOUT(LINE,COL2),R8,MESSAGE(CLASSX,ERRINDX)

which moves the string of the length specified in general
register R8, found at the memory location computed by summing the
address value of MESSAGE with the contents of both index
registers CLASSX and ERRINDX. The string is moved to a storage
location whose address value is computed by summing the address
value of PRINTOUT plus the contents of both index registers, LINE
and COL2. The number of bytes to be filled is the length
specified in general register R7. If the length in R8 is less
than that in R7, the MOVEP instruction, by definition, pads the
extra bytes with the default character, a space.

In the preceding example, the first operand field is the register
identifier, R7; the second operand field is the storage address
at location PRINTOUT, as double indexed by the register
identifiers, LINE and COL2; the third operand field is the
register identifier, R8; and the fourth operand field is the
string's location MESSAGE, as double indexed by the register
identifiers, CLASSX and ERRINDX.

Another example is:

    PMV   =8,DECSUMS(SALESID),5,TOTALS(ORDERX)

which packs and moves the unpacked decimal data digit string
whose length is indicated in general register 5. Note that the
5 means a general register because no equal sign precedes it.
The unpacked decimal data digit string is found at the memory
location computed by summing the address value of TOTALS with the

contents of the single index register identifier ORDERX. For
details on how this conversion takes place, refer to the
instruction definitions in the appropriate processor manuals.
Generally, the unpacked decimal data is converted to packed
decimal data up to the number of digits that may occupy the
reserved byte length, indicated by the =8 expression. In this
case, 8 bytes are reserved, providing storage for 15 decimal
packed digits and a position for the sign-indicator. The PMV
instruction, by definition, has various safeguards for illegal
digit cases and overflow, and provides leading zeros as needed,
when the number of positions available for either the unpacked
digits and the packed digits is of unequal length. The memory
location to which the converted digit data is moved is computed
by summing the address value of DECSUMS with the contents of the
single index register SALESID.

In the preceding example, the first operand field is the
immediate value =8. Note that the equal sign specifies that 8 is
an immediate value and not a register identifier. The second
operand field is the address location DECSUMS as singly indexed
by the register identifier, SALESID. The third operand field is
the register identifier 5; and the fourth operand field is the
address location TOTALS, as indexed by the single index register
identifier ORDERX.

3.7 MACHINE INSTRUCTIONS

Table 3-1 lists the mnemonics for all machine instructions.
Where there is no entry in the format column, that instruction is
not available for that particular line of processors.

TABLE 3-1 SUMMARY OF MACHINE INSTRUCTIONS
AND MNEMONICS

INSTRUCTION	MNEMONIC	32-BIT FORMAT	16-BIT FORMAT
Add	A	RX	RX*
Add DP Floating Point	AD	RX	RX
Add DP Floating Point Register	ADR	RR	RR
Add to Bottom of list	ABL	RX	RX
Add to Bottom of list Flagged	ABLF		RX**
Add with Carry Halfword	ACH		RX
Add with Carry Halfword Register	ACHR		RR
Acknowledge Interrupt	ACK		RX
Acknowledge Interrupt Register	ACKR		RR
Add Floating Point	AE	RX	RX

TABLE 3-1   SUMMARY OF MACHINE INSTRUCTIONS
AND MNEMONICS (Continued)

INSTRUCTION	MNEMONIC	32-BIT FORMAT	16-BIT FORMAT
Add Floating Point Register	AER	RR	RR
Add Halfword	AH	RX	RX
Add Halfword Immediate	AHI	RI1	RI
Add Halfword to Memory	AHM	RX	RX
Add Halfword Register	AHR	RR*	RR
Acknowledge Interrupt	AI		RX*
Add Immediate	AI	RI2	RI*
Acknowledge Interrupt Register	AIR		RR
Add Immediate Short	AIS	SF	SF
Autoload	AL	RX	RX
Add to Memory	AM	RX	RX
Add Register	AR	RR	RR
Add to Top of List	ATL	RX	RX
Add to Top of List Flagged	ATLF		RX
Branch and Link	BAL	RX	RX
Branch and Link Register	BALR	RR	RR
Branch to Control Storage	BDCS	RX	RI
Branch on Equal Status High Speed	BESHS		RX**
Branch on False Condition Backward Short	BFBS	SF	SF
Branch on False Condition	BFC	RX	RX
Branch on False Condition Register	BFCR	RR	RR
Branch on False Condition Forward Short	BFFS	SF	SF
Branch on Not Equal Status High Speed	BNSHS		RX**
Branch on True Condition Backward Short	BTBS	SF	SF
Branch on True Condition	BTC	RX	RX
Branch on True Condition Register	BTCR	RR	RR
Branch on True Condition Forward Short	BTFS	SF	SF
Branch on Index High	BXH	RX	RX
Branch on Index Low or Equal	BXLE	RX	RX

TABLE 3-1   SUMMARY OF MACHINE INSTRUCTIONS
AND MNEMONICS (Continued)

INSTRUCTION	MNEMONIC	32-BIT FORMAT	16-BIT FORMAT
Compare	C	RX	RX*
Complement Bit	CBT	RX	
Compare DP Floating Point	CD	RX	RX
Compare DP Floating Point Register	CDR	RR	RR
Compare Floating Point	CE	RX	RX
Compare Floating Point Register	CER	RR	RR
Compare Halfword	CH	RX	RX
Compare Halfword Immediate	CHI	RI1	RI
Compare Halfword Register	CHR	RR*	RR
Change to Halfword Value	CHVR	RR	
Compare Immediate	CI	RI2	RI*
Compare Logical	CL	RX	RX*
Compare Logical Byte	CLB	RX	RX
Compare Logical Halfword	CLH	RX	RX
Compare Logical Halfword Immediate	CLHI	RI1	RI
Compare Logical Halfword Register	CLHR	RR*	RR
Compare Logical Immediate	CLI	RI1	RI*
Compare Logical Register	CLR	RR	RR*
Compare Register	CR	RR	RR*
Cyclic Redundancy Check Module 12	CRC12	RX	RX**
Cyclic Redundancy Check Modulo 16	CRC16	RX	RX**
Decrement Counter High Speed	DCHS		RX**
Divide	D	RX	RX*
Divide DP Floating Point	DD	RX	RX
Divide DP Floating Point Register	DDR	RR	RR
Divide Floating Point	DE	RX	RX
Divide Floating Point Register	DER	RR	RR
Divide Halfword	DH	RX	RX
Divide Halfword Register	DHR	RR*	RR
Divide Register	DR	RR	RR*

TABLE 3-1 SUMMARY OF MACHINE INSTRUCTIONS
AND MNEMONICS (Continued)

INSTRUCTION	MNEMONIC	32-BIT FORMAT	16-BIT FORMAT
Enter Control Storage	ECS	RI1	SF
Exchange Program Status Register	EPSR	RR	RR
Exchange Byte Register	EXBR	RR	RR
Exchange Halfword Register	EXHR	RR	
Float DP Floating Point Register	FLDR	RR	RR
Float Register	FLR	RR	RR
Fix DP Floating Point Register	FXDR	RR	RR
Fix Register	FXR	RR	RR
Generate Inter-Process Interrupt	GIPI		RR**
Load	L	RX	RX*
Load Address	LA	RX	RI*
Load Byte	LB	RX	RX
Load Byte High Speed	LBHS		RI**
Load Byte High Speed Indirect	LBHSI		RX**
Load Byte Register	LBR	RR	RR
Load Complement Short	LCS	SF	SF
Load DP Floating Point	LD	RX	RX
Load DP Floating Point Register	LDR	RR	RR
Load Floating Point	LE	RX	RX
Load Floating Point Register	LER	RR	RR
Load Halfword	LH	RX	RX
Load Halfword Immediate	LHI	RI1	RI
Load Halfword Logical	LHL	RX	RX*
Load Halfword Register	LHR	RR*	RR
Load Halfword and Set	LHS	RX	
Load Immediate	LI	RI2	RI*
Load Immediate Short	LIS	SF	SF
Load Multiple	LM	RX	RX
Load Multiple DP Floating Point	LMD	RX	RX
Load Multiple Floating Point	LME	RX	RX
Load Program Status	LPS		RX
Load Program Status Register	LPSR		RR
Load PSW	LPSW	RX	RX
Load PSW Register	LPSWR	RR	
Load Real Address	LRA	RX	

TABLE 3-1   SUMMARY OF MACHINE INSTRUCTIONS
AND MNEMONICS (Continued)

INSTRUCTION	MNEMONIC	32-BIT FORMAT	16-BIT FORMAT
Load Register	LR	RR	RR*
Multiply	M	RX	RX*
Multiply DP Floating Point	MD	RX	RX
Multiply DP Floating Point Register	MDR	RR	RR
Multiply Floating Point	ME	RX	RX
Multiply Floating Point Register	MER	RR	RR
Multiply Halfword	MH	RX	RX
Multiply Halfword Register	MHR	RR*	RR
Multiply Halfword Unsigned	MHU		RX
Multiply Halfword Unsigned Register	MHUR		RR
Move and Process Byte String Register	MPBSR	RR	
Multiply Register	MR	RR	RR*
AND	N	RX	RX*
AND Halfword	NH	RX	RX
AND Halfword Immediate	NHI	RI1	RI
AND Immediate	NI	RI2	RI*
AND Halfword Register	NHR	RR*	RR
AND Register	NR	RR	RR*
OR	O	RX	RX*
Output Command	OC	RX	RX
Output Command Register	OCR	RR	RR
OR Halfword	OH	RX	RX
OR Halfword Immediate	OHI	RI1	RI
OR Halfword to Memory	OHM		RX**
OR Halfword Register	OHR	RR*	RR
OR Immediate	OI	RI2	RI*
OR Register	OR	RR	RR*
Process Byte	PB	RX	
Process Byte Register	PBR	RR	
Read Block	RB	RX	RX
Remove from Bottom of List	RBL	RX	RX
Remove from Bottom of List Flagged	RBLF		RX**
Read Block Register	RBR	RR	RR
Reset Bit	RBT	RX	
Read Data	RD	RX	RX

TABLE 3-1   SUMMARY OF MACHINE INSTRUCTIONS
AND MNEMONICS (Continued)

INSTRUCTION	MNEMONIC	32-BIT FORMAT	16-BIT FORMAT
Read DCS	RDCS	RR	RR
Read Data High Speed	RDHS		RX**
Read Data High Speed Register	RDRHS		RR**
Read Data Register	RDR	RR	RR
Read Halfword	RH	RX	RX
Read Halfword Register	RHR	RR	RR
Rotate Left Logical	RLL	RI1	RI
Rotate Left Logical Short	RLLS		SF**
Read Process Data High Speed	RPDHS		RX**
Replace PSW	RPSW		RR**
Rotate Right Logical	RRL	RI1	RI
Rotate Right Logical Short	RRLS		SF**
Remove from Top of List	RTL	RX	RX
Remove from Top of List Flagged	RTLF		RX**
Subtract	S	RX	RX*
Store Byte High Speed Indirect	SBHSI		RI**
Set Bit	SBT	RX	
Subtract with Carry Halfword	SCH		RX
Subtract with Carry Halfword Register	SCHR		RR
Simulate Channel Program	SCP	RX	
Subtract DP Floating Point	SD	RX	RX
Subtract DP Floating Point Register	SDR	RR	RR
Subtract Floating Point	SE	RX	RX
Subtract Floating Point Register	SER	RR	RR
Set Program Mask	SETM		RX
Set Program Mask Register	SETMR		RR
Subtract Halfword	SH	RX	RX
Subtract Halfword Immediate	SHI	RI1	RI
Subtract Halfword from Memory	SHM		RX**

TABLE 3-1   SUMMARY OF MACHINE INSTRUCTIONS
AND MNEMONICS (Continued)

INSTRUCTION	MNEMONIC	32-BIT FORMAT	16-BIT FORMAT
Subtract Halfword Register	SHR	RR*	RR
Subtract Immediate	SI	RI2	RI*
Simulate Interrupt	SINT	RI1	RI
Subtract Immediate Short	SIS	SF	SF
Shift Left Arithmetic	SLA	RI1	RI
Shift Left Halfword Arithmetic	SLHA	RI1	RI
Shift Left Halfword Logical	SLHL	RI1	RI
Shift Left Logical	SLL	RI1	RI
Shift Left Halfword Logical Short	SLHLS	SF	RI
Shift Left Logical Short	SLLS	SF	SF
Store PSW	SPSW		RR**
Subtract Register	SR	RR	RR*
Shift Right Arithmetic	SRA	RI1	RI
Shift Right Halfword Arithmetic	SRHA	RI1	RI
Shift Right Halfword Logical	SRHL	RI1	RI
Shift Right Logical	SRL	RI1	RI
Shift Right Halfword Logical Short	SRHLS	SF	SF
Shift Right Logical Short	SRLS	SF	SF
Sense Status	SS	RX	RX
Sense Status Register	SSR	RR	RR
Store	ST	RX	RX*
Store Byte	STB	RX	RX
Store Byte High Speed	STBHS		RX**
Store Byte Register	STBR	RR	RR
Store DP Floating Point	STD	RX	RX
Store Floating Point	STE	RX	RX
Store Halfword	STH	RX	RX
Store Multiple	STM	RX	RX
Store Multiple DP Floating Point	STMD	RX	RX
Store Multiple Floating Point	STME	RX	RX
Supervisor Call	SVC	RX	RX

TABLE 3-1   SUMMARY OF MACHINE INSTRUCTIONS
AND MNEMONICS (Continued)

INSTRUCTION	MNEMONIC	32-BIT FORMAT	16-BIT FORMAT
Test Bit	TBT	RX	
Test Halfword Immediate	THI	RI1	RI
Test Immediate	TI	RI2	RI*
Translate	TLATE	RX	RX**
Test and Set	TS	RX	
Unchain	UNC		RR**
Write Block	WB	RX	RX
Write Block Register	WBR	RR	RR
Write Data	WD	RX	RX
Write DCS	WDCS	RR	RR
Write Data Register	WDR	RR	RR
Write Data High Speed	WDHS		RX**
Write Data High Speed Register	WDRHS		RR**
Write Halfword	WH	RX	RX
Write Halfword Register	WHR	RR	RR
Write Processed Data High Speed	WPDHS		RX**
Exclusive OR	X	RX	RX*
Exclusive OR Halfword	XH	RX	RX
Exclusive OR Halfword Immediate	XHI	RI1	RI
Exclusive OR Halfword Register	XHR	RR*	RR
Exclusive OR to Memory	XHM		RX**
Exclusive OR Immediate	XI	RI2	RI*
Exclusive OR Register	XR	RR	RR*

* The indicated mnemonic operation code is generated, and the listing is flagged with a question mark to indicate a potential error.

** Model 50 instruction set

Table 3-2 lists the mnemonics for machine formats extended for the Perkin-Elmer Model 3220 Processors. The 16-bit format is not applicable.

TABLE 3-2   SUMMARY OF MACHINE INSTRUCTIONS AND
MNEMONICS EXTENDED FOR THE PERKIN-ELMER
3200 SERIES PROCESSORS

INSTRUCTIONS	MNEMONIC	32-BIT FORMAT
Breakpoint	BRK	RR
Compare alphanumeric (R0=pad)	CPAN	RXRX
Compare alphanumeric and default pad	CPANP	RXRX
Load interruptible state	ISRST	RX
Save interruptible state	ISSV	RX
Load complement SP register	LCER	RR
Load complement DP register	LCDR	RR
Load DP register from SP memory	LDE	RX
Load DP register from SP register	LDER	RR
Load DP register from general register pair	LDGR	RR
Load process state	LDPS	RX
Load SP register from DP memory	LED	RX
Load SP register from DP register	LEDR	RR
Load SP register from general register	LEGR	RR
Load general register pair from DP register	LGDR	RR
Load general register from SP register	LGER	RR
Load packed decimal string as binary	LPB	RX
Load positive DP register	LPDR	RR
Load positive SP register	LPER	RR
Load process segment table descriptor	LPSTD	RX
Load system segment table descriptor	LSSTD	RX
Move and pad (R0=pad)	MOVE	RXRX
Move and pad default pad	MOVEP	RXRX
Move translated until	MVTU	RXRX
Pack and move	PMV	RXRX
Pack and move absolute	PMVA	RXRX
Read error logger	REL	RX1 ***
Reset memory voltage failure	RMVF	RX1 ***

INSTRUCTIONS	MNEMONIC	32-BIT FORMAT
Store DP register in SP memory	STDE	RX
Store binary as packed decimal string	STPB	RX
Store process state	STPS	RX
Unpack and move	UMV	RXRX
Unpack and move absolute	UMVA	RXRX
Store byte with no ECC	XSTB	RX

*** No register or other operands allowed in source
    format

In addition to the set of mnemonics listed in Tables 3-1 and 3-2,
CAL/32 recognizes a complete set of extended branch mnemonics.
These instructions allow the programmer to call for conditional
branch instructions without having to state explicitly the
condition code mask. Table 3-3 lists these instructions.

TABLE 3-3   EXTENDED BRANCH MNEMONICS

INSTRUCTION	MNEMONIC
Branch on carry	BC
Branch on carry register	BCR
Branch on carry short	BCS
Branch on no carry	BNC
Branch on no carry register	BNCR
Branch on no carry short	BNCS
Branch on equal	BE
Branch on equal register	BER
Branch on equal short	BES
Branch on not equal	BNE
Branch on not equal register	BNER
Branch on not equal short	BNES
Branch on low	BL
Branch on low register	BLR
Branch on low short	BLS

TABLE 3-3 EXTENDED BRANCH MNEMONICS
(Continued)

```
-----------------------------------------------------
|              INSTRUCTION              | MNEMONIC  |
|=======================================|===========|
| Branch on not low                     | BNL       |
| Branch on not low register            | BNLR      |
| Branch on not low short               | BNLS      |
|                                       |           |
| Branch on minus                       | BM        |
| Branch on minus register              | BMR       |
| Branch on minus short                 | BMS       |
|                                       |           |
| Branch on not minus                   | BNM       |
| Branch on not minus register          | BNMR      |
| Branch on not minus short             | BNMS      |
|                                       |           |
| Branch on plus                        | BP        |
| Branch on plus register               | BPR       |
| Branch on plus short                  | BPS       |
|                                       |           |
| Branch on not plus                    | BNP       |
| Branch on not plus register           | BNPR      |
| Branch on not plus short              | BNPS      |
| Branch on overflow                    | BO        |
| Branch on overflcw register           | BOR       |
| Branch on overflcw short              | BOS       |
|                                       |           |
| Branch on no overflow                 | BNO       |
| Branch on no overflow register        | BNOR      |
| Branch on no overflow short           | BNOS      |
|                                       |           |
| Branch on zero                        | BZ        |
| Branch on zero register               | BZR       |
| Branch on zero short                  | BZS       |
|                                       |           |
| Branch on not zero                    | BNZ       |
| Branch on not zero register           | BNZR      |
| Branch on not zero short              | BNZS      |
|                                       |           |
| Branch unconditional                  | B         |
| Branch unconditional register         | BR        |
| Branch unconditional short            | BS        |
|                                       |           |
| No operation                          | NOP       |
| No operation register                 | NOPR      |
-----------------------------------------------------
```

The extended branch instructions are essentially single operand instructions where the first operand mask value is included in the operation mnemonic. The programmer supplies only the second

operand or branch location. For short branches, the programmer does not have to specify the forward or backward direction. CAL/32 determines the direction of the branch and generates the appropriate machine code. For example:

```
LOOP1    L      STRNG,TABLE(PTR)        LOAD STRING FROM TABLE
         CLR    STRNG,INPUT             COMPARE WITH INPUT
         BES    END                     EQUIVALENT FOUND
         AIS    PTR,4                   NOT FOUND INCREMENT PTR
         BNZS   LOOP1                   GET NEXT STRING
         XR     STRNG,STRNG             NOT FOUND END OF TABLE
END      ST     STRNG,RETURN            RETURN VALUE
```

In this program, CAL/32 determines the locations of LOOP1 and END and generates the required forward and backward branch instructions.

Two more CAL/32 instructions that do not have direct machine equivalents are:

```
         INSTRUCTION                     MNEMONIC

    Branch on true condition short       BTCS
    Branch on false condition short      BFCS
```

With these instructions, the programmer must specify the mask value and the branch location. CAL/32 determines the direction, forward or backward, and the appropriate machine operation is generated.

3.8 ASSEMBLER INSTRUCTIONS

Assembler instructions control the assembly process. Although they may resemble machine instructions in form, they do not generate any machine executable code. They are used to define symbols, reserve storage, generate data constants, and control the final output.

3.8.1   Symbol Definition Instructions

Symbol definition instructions allow the programmer to assign values to symbols and set up communication paths between separately assembled programs. The latter operation facilitates the use of subroutines because they can be written and assembled separately from the main program. At load time, a linking loader uses information supplied by CAL/32 to resolve addresses between main programs and subroutines to set up the correct linkage.

3.8.1.1 Equate (EQU) Instruction

This is one of the most commonly used assembler instructions. It assigns values to symbols and it has the form:

```
        NAME            OPERATION           OPERAND

    A symbolic name     EQU             An expression
```

Examples of EQU instructions are:

```
    LOOP        EQU     LOOP1
    TOP         EQU     END-64
    DELTA       EQU     BOTTOM-TOP
    HERE        EQU     *
    START       EQU     X'10FE'
    SUM         EQU     1
    PTR         EQU     2
```

EQU instructions can appear anywhere in the program. CAL/32 requires that each EQU instruction have a symbol in the name field and treats the absence of this symbol there as an error. The value assigned to a symbol by an EQU instruction is absolute or relocatable, depending on the type of expression in the operand field. If the operand of an EQU statement contains a forward reference, CAL/32 will perform any additional passes required to define all symbols. CAL/32 does not reserve storage for symbols defined by an EQU instruction. Wherever it encounters the symbol in the program, CAL/32 replaces the symbol with the value defined in the EQU instruction. For example:

```
    STRNG       EQU     1
    PTR         EQU     2
    INPUT       EQU     3
                  .
                  .
                  .
    LOOP1       L       STRNG,TABLE(PTR)     LOAD STRING FROM TABLE
                CLR     STRNG,INPUT          COMPARE WITH INPUT
```

In this case, CAL/32 generates the code to load register 1 with four bytes located at the address specified by TABLE, indexed by register 2. The next instruction causes CAL/32 to generate the code to compare the four bytes in register 1 with the contents of register 3. The use of the EQU instruction here allows the programmer to assign meaningful names to the registers that hold the character strings, and index into the table. It also

provides a simple way to redefine the values assigned to these symbols. By changing the EQU instructions and reassembling, it is possible to change the values assigned to the symbols without doing extensive editing to change each individual statement where these registers are used.

It is also possible, although not recommended, to redefine a symbol within a program. For example:

```
LOOP1       EQU     *
              .
              .
              .
LOOP1       EQU     *
```

When the symbol LOOP1 is encountered in the first EQU instruction, CAL/32 assigns it the value of the current location counter. Subsequent references to LOOP1 receive this value. Following the second EQU instruction, the value of LOOP1 is changed to the value of the new location counter. Because such redefinitions might not be intentional, CAL/32 issues a warning message wherever a symbol is redefined by an EQU instruction. (In the example, the programmer might have intended to write LOOP2 instead of LOOP1 in the second EQU instruction.)

The user must guard against circular location counter dependency, as shown in the following example:

```
A           EQU     *
            DS      1
            DS      B-A
B           EQU     *
            END
```

CAL/32 will flag an "M001 xxxTOP" error where xxx is PURE, IMP, or ABS, depending upon the current location counter.

As stated earlier, CAL/32 permits the user to define operation mnemonics within the program. To do this, the user defines the new mnemonic in an EQU instruction in which the new operation mnemonic is in the name field, and the operand field contains a hexadecimal constant of the form X'nnxy'. Here, nn is the machine language operation code, and x and y are descriptors that tell CAL/32 how to handle the new mnemonic. The values of x and y inform CAL/32 of the instruction format. The values are defined as follows:

```
x = 0, y = 8    RR or SF format
x = 0, y = 2    RX or RI format
x = 0, y = 4    RI1 format
x = 0, y = 1    RI2 format
```

To define extended branch mnemonics, x gets a value equal to the R1 field (mask) and y gets one of the following values:

```
y = 3    RX format
y = C    RR format
y = D    SF format
```

For example, in the instruction:

```
BTC         15,ADDR
```

the branch on true condition mnemonic and the mask field 15 can be combined into an extended branch instruction as follows:

```
BTCF        EQU  X'42F3'
```

in which BTCF is the new mnemonic; 42 is the machine code for the branch on true condition instruction; F is the mask value (15); and 3 specifies RX format. Once this new mnemonic is defined, the programmer can write:

```
BTCF    ADDR
```

instead of:

```
BTC         15,ADDR
```

The new mnemonic definition remains in effect only for the program in which it is defined. The new mnemonic must be different from all other mnemonics recognized by CAL/32.

There are three things to remember in using equate statements:

1. The name field must always contain a valid symbol.

2. The operand field must always contain a defined symbol or expression.

3. The symbol that appears in the name field of an equate instruction must not appear in the name field of any other instruction, except another equate instruction.

If any of these rules are violated, CAL/32 generates an appropriate error message.

3.8.1.2 External, Entry, Weak External, Weak Entry, and Data Entry (EXTRN, ENTRY, WXTRN, WNTRY, and DNTRY) Instructions

These instructions are listed together since they perform corresponding functions to establish links between main programs and subroutines, and between programs with a common data base. These instruction forms are:

NAME	OPERATION	OPERAND
Not used (illegal)	EXTRN	One or more symbols separated by commas
Not used (illegal)	ENTRY	One or more symbols separated by commas
Not used (illegal)	WXTRN	One or more symbols separated by commas
Not used (illegal)	WNTRY	One or more symbols separated by commas
Not used (illegal)	DNTRY	One or more symbols separated by commas

The EXTRN instruction identifies symbols referenced by the program but defined outside the program. The ENTRY instruction identifies symbols defined within the program and referenced externally. (They can be referenced internally as well.)

For example, consider two programs: one calculates the sine and cosine of an angle, the other uses the sine and cosine. The first is a general purpose program that could be used by many other programs. The ENTRY and EXTRN instructions make this possible without having to assemble the sine and cosine program every time it is needed. The sine and cosine program would contain an ENTRY instruction and entry points such as:

```
          ENTRY SIN,COS
             .
             .
             .
SIN       EQU    *
             .
             .
             .
COS       EQU    *
             .
             .
             .
```

The symbols SIN and COS appear as operands in the ENTRY instruction and as names in the EQU instructions. When CAL/32 assembles this program, CAL/32 informs the linking loader that the locations identified by the names SIN and COS are entry points into the program.

The program that uses sine and cosine would contain an external statement and branch instructions such as:

```
        EXTRN SIN,COS
          .
          .
          .
        BAL    LINK,SIN
          .
          .
          .
        BAL    LINK,COS
          .
          .
          .
```

At assembly time, CAL/32 generates object data to inform the linkage editor that the symbols SIN and COS are externally defined. At link time, the linkage editor uses this information, along with the information generated by the entry instruction in the other program, to provide the necessary linkage.

The WXTRN instruction is essentially equivalent to the EXTRN instruction. However WXTRN symbols are subject to the following exception processing by Link:

● An error condition does not arise if the symbol is not resolved.

● Object libraries are not searched in order to satisfy a weak external.

● If a module containing an entry point referenced by a WXTRN symbol is included, then the entry point will be used to satisfy WXTRN references to it in the normal fashion.

The WNTRY instruction is essentially equivalent to the ENTRY instruction. However, WNTRY symbols are subject to the following exception processing by Link.

● Weak entry points are not examined when searching an object library. Therefore, a program module containing a weak entry point is not included to satisfy an external reference.

● If a program module containing a weak entry point is included from a module, the weak entry point will be used to satisfy external references in the normal fashion.

The DNTRY instruction is essentially equivalent to the ENTRY instruction. However, symbols nominated by DNTRY are resolved directly when building overlaid modules rather than resolved in an SVC instruction. This instruction identifies a symbol defined local to the program containing the DNTRY instruction.

To help protect references to data in higher level nodes, Link automatically loads the entire path of overlays starting at the overlay containing data and ending with the overlay making the reference to a data entry point (DNTRY). A reference to a program section positioned in a higher level node, via the POSITION command, is treated the same way. A reference to data or a program section in the root will not cause a path of overlays to be loaded.

If a DNTRY is referenced in a lower level node, an SVC 5 manual overlay load might be required to ensure that the overlay is in memory at the time of the reference.

Restrictions on the use of external and entry instructions are:

● The operand field of an external instruction must not contain an expression, such as SIN+4.

● Expressions involving externally defined symbols must be of the form:

 - External symbol + absolute expression

 - External symbol - absolute expression

        BAL   LINK,SIN+4

   is a legal use of an externally defined symbol.

● Externally defined symbols cannot be used internally as instruction identifiers.

● Any symbol identified as an entry must appear internally in the name of an instruction.

● Symbols identified as entries cannot be redefined by multiple equate instructions.

3.8.1.3 Include (INCLD) Instruction

This information provides Link with a mechanism to guarantee the inclusion of object modules without other linkage references to it. It has the form:

```
     NAME          OPERATION       CPERAND

Not used          INCLD           One or more
(illegal)                         symbols
                                  separated by
                                  commas
```

The INCLD is used in the same fashion as the EXTRN to linking references. However, this instruction is used to nominate program modules rather than external symbols.

NOTE

CAL/32 will generate the same object as in the past, provided none of the following instructions are used: external with offset, DCMD, DNTRY, WNTRY, WXTRN, or INCLD. The assembly of any of these instructicns will produce an object that TET will reject. Link is required to process modules containing this extended object. These instructions are only valid in a Target 32 assembly and have no effect on 16-bit object generation.

3.8.2 Data Definition Instructions

These instructions allow the programmer to reserve areas of memory to be used at run time. Some of these instructions allow the programmer to specify values with which these areas can be initialized at load time. Other data definition instructions provide easy access to complex data structures.

3.8.2.1   Define Storage (DS, DSH, DSF) Instruction

This instruction causes CAL/32 to reserve a block of storage within the program without initializing the reserved locations to any value. It has the form:

```
     NAME          OPERATION       OPERAND

A symbol          DS              An absolute
(optional)                        expression
A symbol          DSH             An absolute
(optional)                        expression
A symbol          DSF             An absolute
(optional)                        expression
```

The DS mnemonic causes CAL/32 to reserve the specified block of storage starting from the value of the current location counter. In the DSH form, CAL/32 first aligns the location counter on a

halfword boundary and then reserves the storage. In the DSF
form, CAL/32 first aligns the location counter on a fullword
boundary. Examples of the define storage instruction are:

```
    BUF1      DS    100
    BUF2      DSH   125
    BUF3      DSF   16
```

In the first example, CAL/32 reserves 100 bytes of storage by
simply adding 100 to the location counter. In the second
example, CAL/32 reserves 125 halfwords (250 bytes) of storage.
CAL/32 does this by aligning the location counter on a halfword
boundary, if it is not already properly aligned, and then adding
250 to it. In the third example, CAL/32 ensures that the
location counter is aligned on a fullword boundary and then adds
64 (the byte equivalent of 16 fullwords) to it. If the operand |
contains a forward reference, CAL/32 will perform any additional |
passes required to define all symbols. |

Define storage instructions are commonly used to reserve storage
areas for transient data. Examples of this are I/O buffers and
register save areas. For example:

```
              ENTRY RSAVE
              EXTRN SIN,CCS
    LINK      EQU   15
                .
                .
                .
    RSAVE     DSF   16
                .
                .
                .
              BAL   LINK,SIN
                .
                .
                .
```

shows how a main program might set up a register save area within
itself. The code for the called program might look like:

```
              ENTRY SIN,CCS
              EXTRN RSAVE
    RO        EQU   0
                .
                .
                .
    SIN       EQU   *
              STM   RO,RSAVE
                .
                .
                .
```

where the subroutine stores the general registers in the externally defined area, RSAVE. When using define storage instructions remember that:

● The DSH and DSF forms cf the instruction ensure halfword and fullword alignment.

● The define storage instructions do not initialize memory to any particular value.

● Cnly one operand is allowed in a define storage instruction, and it must be a defined, absolute symbol or expression.

3.8.2.2 Define Constant (DC, DCF) Instruction

The define constant instruction allcws the programmer to reserve areas of memory and at the same time specify the initial value to be loaded into them. The define constant instruction has two forms:

NAME	OPERATION	OPERAND
A symbol (optional)	DC	One or more operands separated by commas
A symbol (optional)	DCF	One or more operands separated by commas

The DC mnemonic ensures that the first of the operands is aligned on a halfword boundary. The DCF mnemonic ensures that the first of the operands is aligned on a fullword boundary. Operands of different types can be used in the same define constant instruction. However, where alignment is a concern, the programmer must be careful in mixing operands of different types. Types of operands are described below.

A single character code indicates the type of constant. This character code is not always required, and the exceptions are noted as they occur. The assembler determines from the character code how it is to interpret the constant and translate it into the proper object format. Table 3-4 lists the character codes recognized by CAL/32, their meanings, and the types of constants generated.

TABLE 3-4 CONSTANT TYPES

CODE	MEANING	MACHINE FORMAT
======	=========	================
X	Hexadecimal	16-bit binary
Y	Hexadecimal	32-bit binary
H	Integer	16-bit signed binary
F	Integer	32-bit signed binary
A	Address	32-bit value of address
Z	Address	16-bit value of address
T	Address	One half of 16-bit address
E	Single precision floating point	32-bit floating point format
D	Double precision floating point	64-bit floating point format
C	Character	An 8-bit code per character (7-bit ASCII)
P	Packed decimal string	Fixed point sign-coded integer of binary encoded 4-bit decimal digits in a string of variable byte length.
U	Unpacked decimal string	Fixed point sign-coded integer of 7-bit ASCII encoded decimal digits (8-bits per digit) in a string of variable byte length.

3.8.2.3 Hexadecimal Constants

A hexadecimal constant consists of one or more hexadecimal digits, 0 through 9 and A through F, enclosed in apostrophes and preceded by the type code X or Y. Where the X type is used, CAL/32 reserves two bytes of storage and generates the loader information that will cause those two bytes to be initialized at load time with the binary representation of the hexadecimal number. The Y type causes four bytes to be reserved and initialized. Examples of hexadecimal constants are:

```
CONSTANT          VALUE

DC    X'1234'      1234
DC    Y'1234'      0000 1234
DCF   X'20'        0020
DCF   Y'0064'      0000 0064
DC    X'1234ABC'   4ABC
```

The first example shows a halfword hexadecimal constant which, because of the DC operation code, is aligned on a halfword boundary. The second example shows a fullword hexadecimal constant. In this case, fullword alignment is not guaranteed. The third example shows a halfword constant aligned on a fullword boundary. The fourth example shows how to force fullword alignment for a fullword constant. The last example shows what happens when too many digits are given. CAL/32 truncates the constant to the least significant digits and generates an error message. The maximum number of digits for an X type constant is four. The maximum number for a Y type constant is eight.

NOTE

> Where fewer than the maximum number of
> digits are given, CAL/32 right justifies
> the value in the location and fills in
> the missing digits with zeros.

Two special mnemonics facilitate the building of hexadecimal tables by eliminating the need to specify the X or Y type code. They have the form:

NAME	OPERATION	OPERAND
A symbol (optional)	DCX	One or more operands separated by commas
A symbol (optional)	DCY	One or more operands separated by commas

Operands for these instructions consist of from one to four hexadecimal digits for the DCX instruction and from one to eight hexadecimal digits for the DCY instruction. Examples of these constants are:

```
DCX    1,0,14AE,20,4040
DCY    1,2FFFE,64,80000000
```

The DCX instruction generates five halfword constants as follows:

```
0001
0000
14AE
0020
4040
```

The DCY instruction generates four fullword constants as follows:

```
0000 0001
0002 FFFE
0000 0064
8000 0000
```

Before generating the constants, CAL/32 ensures that they are properly aligned with halfword constants aligned on halfword boundaries and fullword constants aligned on fullword boundaries.

3.8.2.4 Integer Constants

Integer constants can be either halfword or fullword. Halfword constants are expressed by the character code H followed by a string of from 1 to 5 decimal digits enclosed in apostrophes. Fullword constants are expressed by the character code F followed by a string of from 1 to 10 decimal digits enclosed in apostrophes. The range of halfword constants is from -32,768 to +32,767. The range of fullword constants is from -2,147,483,648 to +2,147,483,647. The decimal strings used in these constants must not include commas or blanks. A sign, + or -, can precede the string.

The internal representation of integer constants is two's complement binary. In this notation, positive numbers and zero have their true binary form. For example, a halfword integer with a value of 25 is represented internally (hexadecimal notation) as:

    00

Negative numbers are expressed in accordance with the formula:

    $Value = 2 - x$

where n is the number of bits used to express the value, and x is the absolute value of the number. For example, to represent minus 10 in a halfword constant:

    $n = 16 \ (10_{16})$
    $x = 10 \ (A_{16})$
    $Value = 10000_{16} - A_{16} = FFF6_{16}$

Examples of integer constants are:

```
        CONSTANT          VALUE

    DC   H'32767'         7FFF
    DC   H'-32768'        8000
    DC   F'1'             0000 0001
    DC   H'-2'            FFFE
    DCF  F'25'            0000 0019
```

The H and F codes themselves do not guarantee correct alignment.
To ensure that a fullword integer is aligned on a fullword
boundary, the programmer should use the DCF instruction.

CAL/32 does not require that integer constants be defined with
the character codes and decimal strings enclosed in apostrophes.
A simple decimal string can be used. For example:

```
    DC   1
    DC   -7
```

The length of the integer constants generated by these
instructions depends on the processor on which the program is to
run. For 32-bit processors such instructions generate fullword
constants, such as:

```
    CONSTANT              VALUE

    DC   1                0000 0001
    DCF  -7               FFFF FFF9
```

For 16-bit processors, these instructions generate halfword
constants, such as:

```
    CONSTANT              VALUE

    DC   1                0001
    DC   -7               FFF9
```

It is possible to force a fullword alignment by using the DCF
mnemonic with a simple decimal string. The use of a DCF
instruction affects only the alignment of the first of the
integer constants; the length of the constant is determined
solely by the processor on which the program is to be run. Thus,
when using these instructions with operands which are simple
decimal strings, it is not possible to generate a halfword
constant for a 32-bit processor.

3.8.2.5 Address Constants

Address constants consist of a single character type code followed by a symbol or an expression enclosed in parentheses. The three types of address constants are A, Z, and T. Type A constants generate fullword address constants in programs intended to be run on 32-bit processors; they generate halfword address constants in programs intended to be run on 16-bit processors. Types Z and T address constants always generate halfword values. Examples of address constants are:

```
DC     A(LOOP+2)
DC     A(TABLE)
DC     A(TOP-BOTTOM)
DC     Z(IOVECTOR)
DC     T(ALPHATAB)
```

For 32-bit processors, the first three examples cause CAL/32 to reserve a fullword of storage, initialized at load time to contain the value of the expression or symbol enclosed in parentheses. This value can be absolute or relocatable, depending on the nature of the expression. The address quantity is right justified in the least-significant 24 bits of the fullword, and the most-significant 8 bits are forced to zero. However, it is possible to use the most-significant bits for some purpose. They might be used as flag bits as in the example:

```
PARAM     DS     4
ADDR      DC     A(PARAM+Y'A0000000')
          EXTRN  SIN
LINK      EQU    15
ADREG     EQU    14
            .
            .
            .
          STE    R0,PARAM
          L      ADREG,ADDR
          BAL    LINK,SIN
            .
            .
            .
```

At the time of the branch and link instruction, register 14 contains the address of the location PARAM in the least-significant 24 bits. The most-significant 8 bits contain the value X'A0'. The subroutine can use the address portion and the flag portion independently, as:

```
                .
                .
                .
SIN        EQU    *
                .
                .
                .
           LE     R4,0(ADREG)              GET PARAMETER
           TI     ADREG,Y'A0000000'        TEST FLAGS
                .
                .
                .
```

The Z type address constants generate halfword values. They can
be used in programs for 32-bit processors if the programmer is
certain that the actual address cannot exceed 65,535, the maximum
unsigned value that can be expressed in a halfword.

The T type address constants are used as entries in translation
tables. These instructions cause CAL/32 to reserve a halfword of
storage initialized with one half of the actual address, right
justified. The most significant bit is zero. These constants
are intended for use with the translation tables associated with
the translate instruction and with the auto driver channel.

Address constants can be written without the type code and
parentheses, as in:

```
      TABLE      DS     16
      BUFF1      DS     64
                 .
                 .
                 .
      ADD1       DC     TABLE                ADDRESS OF TABLE
      ADD2       DC     BUFF1                ADDRESS OF BUFFER ONE
                 .
                 .
                 .
```

Where this convention is used, the size of the generated constant
depends on the processor for which the program is written. For
32-bit assemblies, CAL/32 generates fullword constants. For
16-bit assemblies, CAL/32 generates halfword constants. The
programmer can force halfword constants to be generated by using
the mnemonic DCZ, as:

```
      DCZ        TABLE,BUFF1
```

which causes a series of halfword address constants to be
generated.

3.8.2.6 Floating Point Constants

The source form for floating point constants consists of a decimal number enclosed in apostrophes and preceded by the letter E for single precision, or the letter D for double precision. The decimal number consists of:

- an optional plus sign or minus sign,

- one or more decimal digits that may include a decimal point, and

- an optional E character followed by an optional plus sign or minus sign, followed by one or two decimal digits denoting a power of 10.

Single precision floating point constants require a fullword of storage. Double precision floating point constants require a doubleword of storage. Internally, floating point constants are represented in excess 64 notation. In this kind of notation, each floating point number consists of a sign, an exponent, and a fraction. The first bit of the number is the sign bit. If this bit is a 1, the number is negative; if it is a 0, the number is positive. The next 7 bits represent the exponent, expressed in excess 64 notation. This field can contain any value between 0 and 127 inclusive. The remaining bits, 24 for single precision and 56 for double precision, represent the fraction with an implied radix point before the first bit.

The true value of the floating point number is obtained by multiplying the fraction by 16 raised to the power indicated by the exponent field. In excess 64 notation, this power is determined by subtracting 64 from the value in the exponent field. In this way, values equal to or greater than 64 produce a 0 or positive power. Raising 16 to this power and then multiplying by the fraction produces values between .0625 and 7.5 x 10. Exponent field values that are less than 64 produce a negative power and values between .06249... and 5.4 x 10- . Floating point 0 is represented by a fullword or a double word of zeros.

Examples of floating point constants are:

CONSTANT		INTERNAL REPRESENTATION
DC	E'1'	4110 0000
DC	E'0.0'	0000 0000
DC	E'7.2E74'	7F19 7817
DC	D'10.5'	41A8 0000 0000 0000
DC	D'5.4E-79'	0010 01D1 33A9 49F6
DC	D'7.2E+75'	7FFE B0E3 AD97 8760

In the internal representation of floating point constants, the fractional part can consist of from 1 to 6 hexadecimal digits for single precision, and up to 14 hexadecimal digits for double precision. If the decimal number exceeds this degree of accuracy, the magnitude of the number is preserved but the precision is lost. In performing the conversion from decimal to internal floating point, CAL/32 carries guard digits to ensure 6 hexadecimal digit accuracy for single precision and 14 hexadecimal digit accuracy for double precision. The programmer must ensure proper alignment.

3.8.2.7 Character Constants

Character constants consist of the single letter code C followed by a string of ASCII characters enclosed in apostrophes. All characters are translated into 7-bit ASCII, in which the most significant bit is always 0. Examples of character constants are:

```
DC      C'NAME'
DC      C'APOSTROPHE = '' '
```

The second example shows how an apostrophe is included in a character constant. Between enclosing apostrophes, a double apostrophe is treated as a single character. The maximum number of characters that can be defined in a single character constant is 64. If the number of characters in a constant is odd, CAL/32 appends a blank character at the end to maintain halfword alignment.

3.8.2.8 Decimal String Constants

The source format for decimal string constants consists of a decimal number enclosed in apostrophes and preceded by the letter P for packed decimal string constants, or by the letter U for unpacked decimal string constants. The decimal number is an integer and consists of an optional plus sign or minus sign, followed by 1 to 31 decimal digits.

The machine internal representation of the packed decimal string constant is a fixed-point, sign-coded integer, where each digit occupies 4 bits and each byte holds 2 digits. That is, each decimal digit, 0 through 9, is binary encoded in a 4-bit hexadecimal digit. As the number of decimal digits varies from 1 to 31, the length in bytes of the decimal string varies from 1 to 16 bytes. The last hexadecimal digit contains a 4-bit code for sign; a hexadecimal C for plus, or a hexadecimal D for minus. The integer representation is right-justified within the variable length string, so the least-significant digit of the decimal number occupies the hexadecimal digit just preceding the sign code. Each digit is thus consecutively packed, with the

most-significant digit (zero or nonzero) in bit positions 0 through 3 of the leftmost byte of the string. Refer to the examples that follow for the differences in internal representation, when the packed decimal string constant is defined by either the define constant (DC) instruction or the define byte (DB) instruction.

The machine internal representation of the unpacked decimal string constant is a fixed-point, sign-coded integer, where each zoned digit occupies a byte. That is, each decimal digit, 0 through 9, is encoded in 7-bit ASCII with the leftmost bit 0; providing an 8-bit byte with the left hexadecimal digit containing a zone code of 3 and the right hexadecimal digit containing the binary encoded decimal digit. As the number of decimal digits varies from 1 to 31, the length in bytes of the decimal string varies from 1 to 31 bytes. The integer representation is right-justified within the variable length string. The rightmost byte contains the least significant digit in its rightmost hexadecimal digit and the sign code in its leftmost hexadecimal digit. The sign code is a 4-bit code, described above with a hexadecimal C for plus, and a hexadecimal D for minus. Each digit is thus consecutively coded into bytes, with the most-significant digit (zoned zero or zoned nonzero). Refer to the following examples for the differences in internal representation, when the unpacked decimal string constant is defined by either the DC instruction or the DB instruction.

The address of the string is the address of the leftmost byte containing the most-significant digit (zero or nonzero). The address generated for either the packed decimal string constant or the unpacked decimal string constant is that associated with the label of the source statements and the current location counter. Examples of the PDS constants are:

SOURCE FORMAT		INTERNAL REPRESENTATION (HEXADECIMAL)			
DB	P'1'	1C			
DB	P'+50'	050C			
DB	P'-879'	879D			
DB	P'+1234'	0123	4C		
DB	P'-12345'	1234	5D		
DB	P'1234567890123456789012345678901'	1234	5678	9012	3456
		7890	1234	5678	901C
DC	P'1'	001C			
DC	P'+50'	050C			
DC	P'-879'	879D			
DC	P'+1234'	0001	234C		
DC	P'12345'	0012	345C		
DC	P'1234567890123456789012345678901'	1234	5678	9012	3456
		7890	1234	5678	901C

Note that as string-processing instructions are intended to operate at the lowest addressable level, on byte-addressable locations these constants are most efficiently generated by the DB instructions, described in the define byte instruction section. If the DC instruction is used, an extra byte of leading zeros is generated, when the number of digits is a multiple of 4, or is an odd number of digits not divisible by 3. Examples of unpacked decimal string (zoned) constants are:

```
                                         INTERNAL
                                         REPRESENTATION
      SOURCE FORMAT                      (HEXADECIMAL)

      DB   U'1'                          C1
      DB   U'+50'                        35C0
      DB   U'-879'                       3837 D9
      DB   U'+1234'                      3132 33C4
      DB   U'12345'                      3132 3334 D5
      DB   U'12345678901234567890123456789 01'   3132 3334 3536 3738
                                         3930 3132 3334 3536
                                         3738 3930 3132 3334
                                         3536 3738 3930 C1

      DC   U'1'                          30C1
      DC   U'+50'                        35C0
      DC   U'-879'                       3038 37D9
      DC   U'+1234'                      3132 33C4
      DC   U'-12345'                     3031 3233 34D5
      DC   U'12345678901234567890123456789 01'   3031 3233 3435 3637
                                         3839 3031 3233 3435
                                         3637 3839 3031 3233
                                         3435 3637 3839 30C1
```

As string processing instructions require programmed length attributes, familiarization with the internal storage requirements for both packed decimal string and unpacked decimal string constants is advisable. In the previous examples, the relationship of number of digits to byte length is as follows:

```
CONSTANT DEFINED BY        BYTE LENGTH

Packed      DB             (integer of n/2) + 1

Packed      DC             2*(integer of n/4) + 2

Unpacked    DB             n

Unpacked    DC             n, for n even
                           n + 1, for n odd
```

where n is the number of decimal digits in the source formats of the decimal constants.

3.8.3 Define Byte (DB) Instruction

This instruction defines consecutive 8-bit bytes of data. It has the form:

NAME	OPERATION	OPERAND
A symbol (optional)	DB	One or more operands separated by commas

The symbol used in the name field of the DB instruction is assigned the value of the current location counter. There is no automatic alignment. The programmer must ensure proper alignment where the symbolic name of a DB instruction is used as an operand identifier in an instruction requiring its operand to be located on a halfword, fullword, or double word boundary.

The operand field can contain one or more operands, separated by commas. There can be an even or an odd number of operands. The operands can be any symbol or expression value. For any operand, other than character or decimal string expressions, the least significant eight bits of the operand value are used to generate one byte of data. Examples of the DB instructions are:

```
DB      X'F7'
DB      128
DB      -1
DB      C'A'
DB      C'ABCDEFG'
```

As shown in the examples, the operand of a DB instruction can be a signed integer. In this case, the integer can have any value between -128 and +127, inclusive.

A special form of the DB instruction:

```
DB      *
```

forces alignment of the location counter to a halfword boundary. If, when this instruction is encountered, the location counter contains an odd value, one byte of zero value is generated, and the location counter is made even. If the location counter is already even, this instruction has no effect.

3.8.4 Define List (DLIST) Instruction

This instruction provides a simple means for defining circular lists used by the machine instructions:

- Add to top of list

- Add to bottom of list

- Remove from top of list

- Remove from bottom of list

The define list instruction has the form:

```
    NAME       OPERATION      OPERAND

  A symbol      DLIST        An absolute
  (optional)                 expression
```

The absolute expression in the operand field specifies the number of slots in the list. For 32-bit assemblies, CAL/32 reserves four halfwords of storage for list pointers, followed by the specified number of fullwords (slots). The first halfword list pointer is initialized with a value equal to the number of slots in the list. The remaining three pointers are initialized to zero. For 16-bit assemblies, CAL/32 reserves four bytes of storage for list pointers, followed by the specified number of halfwords. The first byte pointer is initialized to a value equal to the number of slots in the list. The remaining byte pointers are initialized to zero. An example of the DLIST instruction is:

```
  LIST1      DLIST 100
```

In a 32-bit assembly, this has the same effect as:

```
  LIST1      DCF    X'64',X'0',X'0',X'0'
             DS     400
```

The DLIST instruction forces alignment to a fullword boundary in 32-bit assemblies. It forces alignment to a halfword boundary for 16-bit assemblies.

3.8.5 Define Command (DCMD) Instruction

This instruction causes the string within the set of apostrophes
to be passed directly to the object code.

```
    NAME          OPERATICN        OPERAND

  A symbol          DCMD         C 'command
  (optional)                     string'
```

The operand of the DCMD instruction is subject to the same
syntactic rules as any other character string. CAL/32 performs
no syntax checking on the command string.

CAL/32 will generate the same object as in the past, provided the
DCMD instruction is not used. The assembly of this instruction
will produce an object that TET will reject. Link is required to
process modules containing this extended object. The DCMD
instruction is valid only in a Target 32 assembly and has no
effect on the 16-bit object generation.

3.8.6 Location Counter Instructions

These instructions allow the programmer to select the current
location counter and set its value. For 32-bit assemblies,
CAL/32 maintains three location counters: pure, impure, and
absolute. For 16-bit assemblies, it maintains two location
counters: relocatable and absolute. At any given time, only one
location counter can be in use. With these instructions, the
programmer can control the program segmentation and relocation.

3.8.6.1 Pure (PURE) Instruction

This instruction causes all subsequent machine instructions to be
assembled as part of the pure segment. It has the form:

```
    NAME        OPERATION        OPERAND

  A symbol        PURE         None
  (optional)                   (ignored)
```

The current location counter is saved, and the new location
counter is set to point to the next halfword boundary beyond the
most recently used location in the pure segment. If a PURE
instruction occurs in a relocatable 16-bit program, it has no
effect. If it occurs in an absolute 16-bit program, it causes a
switch to the relocatable location counter.

3.8.6.2 Impure (IMPUR) Instruction

This instruction causes all subsequent instructions to be assembled as part of the impure segment. It has the form:

```
NAME        OPERATION     OPERAND

A symbol     IMPUR        None
(optional)                (ignored)
```

The current location counter is saved, and the new halfword boundary is set beyond the most recently used impure address. In 16-bit assemblies, this instruction has no effect if the program is already in relocatable mode. If it is in absolute mode, the location counter is switched to relocatable.

NOTE

Unless otherwise specified by the programmer, impure mode is assumed.

3.8.6.3 Origin (ORG) Instruction

This instruction selects a location counter and sets it to a defined value. It has the form:

```
NAME         OPERATION     OPERAND

A symbol       ORG         A symbol or
(optional)                 expression
```

The operand of the origin instruction determines which location counter is selected and the value it is given. If the value of the operand is pure, impure, absolute, or relocatable, the corresponding location counter is selected and set to the operand value. If the operand contains a forward reference, CAL/32 will perform any additional passes required to define all symbols.

The user must guard against circular location counter dependency, as in the following example:

```
            ORG    A
            LIS    4,4
    A       EQU    B
            LIS    4,4
    B       EQU    *
            END
```

CAL/32 will flag an "M001 xxxTOP" error, where xxx is PURE, IMP, |
or ABS depending on the current location counter. |

NOTE

> If no ORG instruction appears at the
> beginning of a program, CAL/32 assumes it
> to be relocatable starting at relocatable
> zero. For 32-bit programs it also
> assumes the impure segment.

3.8.6.4 Absolute (ABS) Instruction

This instruction causes the location counter to be put in the
absolute mode. It has the form:

```
NAME          OPERATION        OPERAND

A symbol        ABS             None
(optional)                      (ignored)
```

The current location counter is saved, and the new location
counter is set to point to the next halfword boundary beyond the
most recently used absolute location. If the absolute location
counter was not previously used, it is set to zero.

3.8.6.5 Align (ALIGN) Instruction

This instruction aligns the current location counter to a power |
of 2. It has the form: |

```
NAME          OPERATION        OPERAND

A symbol        ALIGN          A symbol or
(optional)                     expression
```

The value contained in the operand field determines the type of
alignment.   Symbols used in the operand field must be previously
defined. The value in the operand field must be absolute and
equal to either two, four, or eight. If the operand value is
two, CAL/32 adjusts the location counter to ensure that it
contains a halfword address. CAL/32 forces fullword alignment if
the operand value is four, and double word alignment if the value
is eight.   If at the time of this instruction, the location
counter is already properly aligned, CAL/32 does not change it.

If it is not properly aligned, CAL/32 increments it by the minimum amount necessary to force proper alignment. A symbol, if used in the name field, receives the value of the location counter after the alignment is performed.

NOTE

If the value of the operand is not absolute, or if it is not correctly defined, CAL/32 forces fullword alignment, and generates an error message.

3.8.6.6 Conditional No Operation (CNOP) Instruction

This instruction is similar to the ALIGN instruction in that it aligns the location counter to a power of 2. It has the form:

```
NAME         OPERATION      OPERAND

A symbol       CNOP         A symbol or
(optional)                  expression
```

The CNOP differs from the ALIGN instruction in that instead of merely incrementing the location counter, it actually inserts no operation instructions into the program stream. The value of the operand must be absolute and equal to two, four, or eight. Symbols used in the operand field must have been previously defined. If at the time this instruction is encountered, the location counter is on an odd boundary, CAL/32 increments it by one to make it even, inserts the required number of CNOP instructions to force alignment, and generates an error message. This instruction has no effect if the location counter is already properly aligned. A symbol, if used in the name field, receives the value of the location counter associated with the first CNOP instruction generated.

3.8.7 Assembler Control Instructions

These instructions allow the programmer to control the assembly process itself, identify the type of processor on which the program is to be run, halt the assembly operation temporarily, and request a limited amount of optimization.

3.8.7.1 Target (TARGT) Instruction

This instruction identifies the type of processor on which the program is to be run. It has the form:

```
    NAME          OPERATION        OPERAND

A symbol          TARGT          A symbol or
(optional)                       expression
```

The value of the operand expression must be either 16 or 32, absolute. Symbols used in the operand field must be previously defined. If the operand value is 16, CAL/32 generates object code for 16-bit processors. If the value is 32, it generates object code for 32-bit processors. If the value is anything else, CAL/32 generates a warning message and generates code for the same type of processor on which it is running. If there is no TARGT instruction in the program, CAL/32 assumes the target machine to be the same as the machine on which the assembly is running.

NOTE

> The TARGT instruction must precede any
> PURE or IMPUR instructions or any
> instruction that generates machine code.

3.8.7.2 End (END) Instruction

The END instruction indicates the end of the source input. It has the form:

```
    NAME        OPERATION            OPERAND

A symbol        END            A symbol or
(optional)                     expression (optional)
```

Because of its function, this statement must be the last instruction in the source input file. The optional operand, if used, identifies the starting location of the program. For example:

```
    MAIN      EQU    *
               .
               .
               .
    LAST      END    MAIN
```

The END instruction, with the operand MAIN, causes CAL/32 to output information identifying the location MAIN as the starting location of the program. The loader and the operating system use this information to ensure that the program starts at the requested location. If there is no operand, the END instruction merely terminates the assembly process without outputting any loader information. The END instruction is required in all CAL/32 programs.

3.8.7.3 Copy Library (CLIB) Instruction

This instruction allows the user to specify or change library files from within a program. It has the form:

    CLIB vol:fname.ext

Each CLIB statement logically concatenates the new library file (operand of CLIB) to any existing library file. If the new library file cannot be assigned, CAL/32 will log an error message and pause.

3.8.7.4 Copy (COPY) Instruction

This instruction allows the programmer to insert source code from library files into the source code received from the regular source input file. It has the form:

```
NAME            OPERATION       OPERAND

A symbol        COPY            A symbol [,vol:fname.ext]
(optional)                      (required)
```

CAL/32 assumes that the library file was assigned to lu 7 (see Appendix A). CAL/32 also assumes that the file is made up of 80-character records. It searches through the logical file, looking only at the first 10 characters of each record until it finds a file label of the form:

```
RECORD POSITION                     CONTENTS

1 and 2                 **
3 through 10            A valid symbolic name of
                       from 1 to 8 characters
```

in which the symbolic name exactly matches the symbol in the operand field. If the search is unsuccessful, CAL/32 logs the message:

    COPY ERROR: xxxxxxxx

in which xxxxxxxx is replaced by the name of the file being
sought. This might happen in the case of incorrect file
assignment. The operator can change the assignment and resume
the assembly process from the location of the COPY instruction.
The COPY instruction allows only one operand. The programmer
must provide one COPY instruction for each file to be copied into
the source stream.

If the optional second operand is supplied, CAL/32 will assign |
and search only that physical file and ignore any files logically |
attached by CLIB. If the file cannot be assigned, CAL/32 will |
log an error message and pause. |

The copy process terminates when an END statement is encountered
in the file, or when a record with either /* or /& in the first
two character positions is encountered. Where an END instruction
is encountered in the copy file, it does not mean the end of the
source file but only the end of the copy file. At this point,
CAL/32 resumes reading from the source input file. COPY
instructions may not appear in files which are themselves being
included in a source program by means of a COPY instruction.

3.8.7.5 File Copy (FCOPY) Instruction |

The assembler instruction FCOPY allows the user to copy an entire |
library file. It has the form: |

    FCOPY vol:fname.ext |

When FCOPY is in effect, a /* starting in column 1 or an END in |
the op code field will be skipped, and copying will continue |
until an end of file is reached. If the file cannot be assigned, |
CAL/32 will log an error message and pause. |

3.8.7.6 Pause (PAUSE) Instruction

The PAUSE instruction allows the programmer to halt the assembly
process. It has the form:

        NAME   OPERATION   OPERAND

     A symbol   PAUSE   None
     (optional)   (ignored)

The PAUSE instruction temporarily halts the assembly process.
When the assembler encounters a PAUSE instruction, the assembler
requests the operating system under which it is running to
suspend execution. The system notifies the operator. The
operator can resume execution of the assembler at the instruction

immediately following the PAUSE instruction by using the operating system command CONTINUE. For example, the PAUSE instruction can be used by the operator to reassign a copy file, such as:

```
COPY      REGEQUS   COPY REGISTER EQUATES
PAUSE
COPY      COMBLKS   COPY COMMON BLOCKS
```

3.8.7.7 Squeeze (SQUEZ) Instruction

The SQUEZ instruction puts CAL/32 into a mode in which it performs a limited amount of space optimization. It has the form:

```
NAME         OPERATION        OPERAND

A symbol     SQUEZ       A symbol or
(optional)               expression (optional)
```

When in optimization mode, CAL/32 makes multiple passes over the source input. During each pass, it attempts to reduce long instructions (48 and 32 bits) to shorter forms (32 and 16 bits). The value of the operand expressions sets the maximum number of passes. If CAL/32 can complete the optimization in fewer passes, it stops the optimization process and completes the assembly.

The value of the operand expression must be an absolute number between 1 and 99. Any symbols used in the expression must have been previously defined. If the operand value is 0, or if there is no operand, CAL/32 assumes a maximum of 9 passes.

NOTE

If there are user induced errors in the source stream (illegal mnemonics or undefined symbols), CAL/32 terminates the squeeze operation and goes on to produce the final assembler output. Some instructions in this output may have been squeezed, depending on where in the process the errors were discovered.

CAL/32 performs three types of space optimization:

1. Changes RX3 instructions to RX2 or RX1

2. Changes operation codes to allow the use of an equivalent, but shorter, instruction

3. Eliminates unconditional branch instructions to the next halfword location

An example of the first type of optimization is the forward reference instruction. In this instruction, the operand is defined in the program at some point beyond the instruction to which it refers.

Example:

```
            •
            •
            •
            A        R1,VALUE
            •
            •
            •
VALUE       DCF      F'125'
            •
            •
            •
```

When CAL/32 processes the ADD instruction, it cannot tell if the location of the second operand, identified by the symbol VALUE, is within the range of either an RX1 or RX2 instruction. It has to assume that an RX3 instruction is necessary. By making additional passes over the source input after all addresses have been resolved, CAL/32 has the needed information to determine if the reference to VALUE is within the range of either an RX1 or an RX2 instruction and make the substitution.

An example of the second type of optimization is:

```
            LI       R3,-1
```

In the optimization mode, CAL/32 reduces this instruction to:

```
            LCS      R3,1
```

which reduces the length of the instruction from 48 bits to 16 bits, without changing the effect. Depending on the processor, the substituted instruction might be faster or slower than the original instruction.

NOTE

CAL/32 changes an operation code only in the object output. The original instruction remains in the listing, flagged with an asterisk.

The third type of optimization does not occur in normal programming, but it does sometimes appear in compiler generated CAL/32. For example:

```
            •
            •
            •
            ST      R1,SAVE
            B       CONTINUE
CONTINUE    L       R1,TEMP
            •
            •
            •
```

In this case, CAL/32 simply eliminates the unnecessary branch instruction, although the branch instruction does appear in the assembly listing, flagged with an asterisk.

More than one SQUEZ instruction can appear in the program. The first SQUEZ instruction sets the number of additional passes. Subsequent SQUEZ instructions put CAL/32 back into optimization mode after a NO SQUEZ instruction (described below) took it out of the optimization mode. Operands may appear in the subsequent SQUEZ instructions, but they are ignored.

Because CAL/32 looks at only one instruction at a time, and because its global data is limited to the symbol table, squeezing might introduce errors into the program. This is most likely to happen when data and instructions are mixed.

Example:

```
            •
            •
            •
            BTC     8,LOOP1
            •
            •
            •
LOOP1       EQU     *
            •
            •
            •
            BFC     0,LOOP2
            DS      26
            ALIGN   4
CONST       DC      F'256'
LOOP2       EQU     *
            •
            •
            •
```

If on one pass, CONST is already aligned on a fullword boundary, the branch to LOOP2 can be converted to a short format branch. A subsequent pass may allow the branch to LOOP1 to be shortened. When this happens, CONST is no longer on a fullword boundary, and CAL/32 adds two to the location counter to align it properly. This forces LOOP2 out of the range of a short branch instruction. CAL/32 will recover from this situation by changing the branch instruction back to its original format and marking it internally as unsqueezable.

3.8.7.8 Squeeze Related (NOSQZ, ERSQZ, NORX3) Instructions

There are three additional instructions that can be used to control squeezing and optimization of the source input file. They have the form:

NAME	OPERATION	OPERAND
A symbol (optional)	NOSQZ	Not used (ignored)
A symbol (optional)	ERSQZ	Not used (ignored)
A symbol (optional)	NORX3	Not used (ignored)

The no squeeze instruction (NOSQZ) has the effect of turning off the optimization processes initiated by a previous SQUEZ instruction. Optimization can be restarted by a subsequent squeeze statement. NOSQZ overrides a squeeze start option.

The error squeeze instruction, (ERSQZ) can be used with the SQUEZ instruction. It forces CAL/32 to continue squeezing even after assembly errors are detected.

The no RX3 instruction (NORX3) provides a simpler form of optimization during a normal 2-pass assembly. Once this instruction is encountered, CAL/32 forces RX instructions to the RX1 or RX2 format. RX3 instruction formats are still generated if double indexing is specified, or if the instruction references an element of a common block or an externally defined symbol. This instruction can be safely used in programs that are smaller than 16kb. It must not be used in segmented (pure and impure), programs.

3.8.7.9 Sequence Checking (SQCHK, NOSEQ) Instructions

The sequence checking instructions enable and disable the sequence checking of source. They have the form:

```
    NAME          OPERATION     OPERAND

A symbol          SQCHK         Not used
(optional)                      (ignored)
A symbol          NOSEQ         Not used
(optional)                      (ignored)
```

The sequence check instruction (SQCHK) causes CAL/32 to compare each source statement sequence number with the number of the preceding statement. Each successive number must be greater in the ASCII collating sequence than the preceding one. CAL/32's initial sequence value is equal to eight spaces, so that numbers can be right justified in the field without leading zeros. If a source statement contains a value equal to or less than the preceding statement, CAL/32 generates an error message. The sequence fields of statements included in the program by a COPY instruction are not checked.

The no sequence check instruction (NOSEQ) disables the sequence checking process. The sequence field of this instruction is checked, if sequence checking was in effect at the time. The default mode of CAL/32 is NOSEQ.

3.8.7.10 Scratch (SCRAT) Instruction

The scratch instruction causes CAL/32 to copy the source input file to a scratch device during pass one. It has the form:

```
    NAME          OPERATION     OPERAND

A symbol          SCRAT         Not used
(optional)                      (ignored)
```

Subsequent passes over the source input file are read from the scratch device. Since no statement preceding the SCRAT instruction can be copied, the SCRAT instruction should be the first statement in the program.

3.8.7.11 Pass Pause (PPAUS) Instruction

This instruction causes CAL/32 to issue a pause request to the operating system at the end of each pass. It has the form:

```
    NAME          OPERATION     OPERAND

A symbol          PPAUS         Not used
(ignored)                       (ignored)
```

The purpose of the PPAUS instruction is to allow the operator to reset the source input file to the beginning for the next pass. This is useful in situations where no scratch file is available, and the source input file is not rewindable.

<div align="center">NOTE</div>

> Where neither the SCRAT instruction nor the PPAUS instruction is used, CAL/32 issues a rewind command to the source input lu at the end of each pass.

3.8.7.12 Message (MSG) Instruction

The message instruction allows the programmer to log a message to the system console. It has the form:

NAME	OPERATION	OPERAND
A symbol (optional)	MSG	Text

The operand field contains the text of the message. All characters following the operation field, up to and including position 71, are sent to the system console as a message. This instruction can appear anywhere in the program, and the message is logged on every pass.

3.8.7.13 Batch Assembly (BATCH, BEND) Instructions

The batch assembly instructions provide a means for assembling more than one complete program in a batch stream. They have the form:

NAME	OPERATION	OPERAND
None (illegal)	BATCH	Not used (ignored)
None (illegal)	BEND	Not used (ignored)

The batch instruction (BATCH) initiates the batch stream. It has the effect of redefining the END instruction so CAL/32 does not terminate itself at the end of the required number of passes. Rather, CAL/32 terminates the assembly of that particular program, reinitializes itself, and starts reading the next program from the source input file. The BATCH instruction must

be the first statement in the stream of programs. If it is used,
CAL/32 assumes that there is a scratch device. Options specified
in the operating system START command remain in effect for the
entire batch assembly (see Appendix A).

The batch end instruction (BEND) terminates the batch assembly.
It must appear immediately following the END instruction in the
last program of the stream. The BEND instruction tells CAL/32 to
go to end of task when final assembly is completed. The end of
task code returned is equal to the highest code generated during
the batch assemblies.   CAL/32 will also terminate a batch
assembly normally if end of file or end of medium status is
detected when attempting to read the first statement after the
END of an assembly.

3.8.7.14 Unreferenced Externals (UREX, NUREX) Instructions

These instructions permit or suppress the output of object code
for unreferenced externals. The default state is UREX. They
have the form:

| NAME   OPERATION   OPERAND

| Not used   UREX   Not used
| (ignored)   (ignored)

| Not used   NUREX   Not used
| (ignored)   (ignored)

3.8.8 Conditional Assembly Instructions

These instructions allow the programmer to include in the program
code sequences that may or may not be assembled, depending on
some condition.   By . simply reassembling the program and
redefining the conditions, a single program can be made to serve
more than one purpose.

3.8.8.1 Compound Conditional (IFx, ELSE, ENDC) Instructions

There are three instructions in this set. They have the form:

      NAME   OPERATION   OPERAND

      A symbol   IFx   A symbol or
      (optional)   expression
      A symbol   ELSE   A symbol or
      (optional)   expression
                                     (ignored)
| A symbol   ENDC   A symbol or
      (optional)   expression
| (ignored)

The compound conditional instructions are used to provide complete conditional assembly capability. A symbol used in the name field of an IF instruction is defined if the condition described by the instruction is true. A symbol used in the name field of an ELSE instruction is defined if the correspondng if condition is false. Symbols used in the name fields of end condition instructions are always defined.

In the first instruction, the compound IF instruction, x represents the actual condition. Following is a list of the various mnemonics for these instructions:

MNEMONIC	MEANING	MNEMONIC	MEANING
IFZ	If zero	IFNM	If nonminus
IFNZ	If nonzero	IFE	If even
IFP	If plus	IFO	If odd
IFNP	If nonplus	IFU	If undefined
IFM	If minus	IFD	If defined

CAL/32 tests the value of the operand when processing compound IF instructions. If the operand meets the condition specified by the operation, the instructions immediately following the IF instruction are assembled. If the operand does not meet the specified condition, the instructions immediately following the IF instruction are not assembled.

The ELSE instruction reverses the state of the assembler as set by a previous compound IF statement. If the assembler was not assembling code because a previous IF statement turned off the assembly process, the appearance of an ELSE instruction would cause the assembler to resume assembling, starting with the instruction immediately fcllowing the ELSE instruction. If the assembler was assembling code because a previous if condition was met, the appearance of the ELSE instruction would cause the instructions immediately following the else instruction not to be assembled. An ELSE instruction is not required to appear in a block of conditionally assembled code.

The third instruction of this set is the end condition instruction (ENDC) which terminates the presently active condition. Normal assembly process resumes with the next instruction. Any compound IF instruction used in the program must have a corresponding ENDC instruction. If the end of the source file is reached before an existing condition terminates, CAL/32 terminates the condition, generates an error message, and resumes normal assembly cn the next pass. If the operand of the IFx contains a forward reference, CAL/32 will perform any additional passes required to define all symbols. As an example of conditional assembly, consider a subroutine that can receive its parameters in either of two ways: first, the parameters are located by referencing a list of addresses immediately following the branch and link instruction in the main

program; second, the address of the actual parameter list is contained in register 14. The subroutine could handle both of these situations with conditional assembly, as follows:

```
          .
          .
          .
          IFZ    CALL1
SUB       LH     R1,0(RF)           GET FIRST PARAMETER ADDRESS
          LH     R1,0(R1)           GET FIRST PARAMETER
          LH     R2,2(RF)           GET SECOND PARAMETER ADDRESS
          LH     R2,0(R2)           GET SECOND PARAMETER
          AIS    RF,4               ADJUST RETURN ADDRESS
          ELSE                      LIST NOT IN LINE
SUB       LH     R1,0(RE)           GET FIRST PARAMETER
          LH     R2,2(RE)           GET SECOND PARAMETER
          ENDC
          .
          .
          .
RETURN    BR     RF                 RETURN TO CALLER
          .
          .
          .
```

If at assembly time, the value of CAL1 is zero, the instructions between the IF instruction and the ELSE instruction are assembled, and the instructions between the ELSE instruction and ENDC instruction are not assembled. If the value of CAL1 is other than zero, only the instructions between the ELSE instruction and the ENDC instruction are assembled.

Another example of conditional assembly shows how conditions can be nested:

```
          .
          .
          .
          IFNP   LGTH               CONDITION #1
*  ERROR  1                         LGTH IS NOT POSITIVE
          ELSE                      CONDITION #1
          IFZ    SRC-DST            CONDITION #2
*  ERROR  2                         SRC IS EQUAL TO DST
          ELSE                      CONDITION #2
          LHI    R1,LGTH
          IFP    SRC-DST            CONDITION #3
          LHI    R2,SRC
          LHI    R3,DST
          ELSE                      CONDITION #3
          LHI    R2,DST
          LHI    R3,SRC
          ENDC                      END CONDITION #3
          ENDC                      END CONDITION #2
          ENDC                      END CONDITION #1
          .
          .
          .
```

This set of nested conditionals depends on the values of three symbols: LGTH, SRC, and DST. If LGTH is negative or zero, only the comment:

```
*   ERROR   1                            LGTH IS NOT POSITIVE
```

is produced. If LGTH is positive, and SRC is equal to DST, only the second comment:

```
*   ERRCR   2                            SRC IS EQUAL TO DST
```

is produced. If LGTH is positive, and SRC is greater than DST, the following instructions:

```
            LHI     R1,LGTH
            LHI     R2,SRC
            LHI     R3,DSC
```

are assembled. If LGTH is positive, and SRC is less than DST, the following instructions are assembled:

```
            LHI     R1,LGTH
            LHI     R2,DST
            LHI     R3,SRC
```

The user must be careful, when using a forward reference in the |
operand field of the IFU instruction, to avoid the following type |
of code: |

```
            IFU     A                                              |
B           EQU     8                                              |
            ENDC                                                   |
A           EQU     1                                              |
            IFNZ    B                                              |
            DS      10                                             |
            ENDC                                                   |
B           EQU     0
            END
```

CAL/32 will flag this code with an "M001 xxxTOP" error where xxx is PURE, IMP, or ABS, depending upon the location counter used.

<div align="center">NOTE</div>

> A condition once set by an IF instruction remains in effect until the corresponding ENDC instruction is encountered. Thus, when the first condition was met, the first comment was produced. The ELSE instruction reversed this state, and no subsequent code was assembled.

3.8.8.2 Simple If (IF) Instruction

The simple IF instruction is retained in CAL/32 to maintain compatibility with previous assemblers. It has the form:

NAME	OPERATION	OPERAND
A symbol (optional)	IF	A symbol or expression

What CAL/32 does on encountering an IF instruction depends on the value of the operand. If the operand has a nonzero value, CAL/32 assembles all statements following the IF instruction, until the end of the source file is reached, or until another IF instruction is encountered in which the operand value is zero. At this point, CAL/32 stops assembling the source input until the END instruction, or another IF instruction with a nonzero operand value, is encountered. If the operand contains a forward reference, CAL/32 will perform any additional passes required to define all symbols.

<div align="center">NOTE</div>

> Do not use simple IF instructions and compound IF instructions in the same program. Simple IF instructions must not be used in nested conditionals.

3.8.8.3 Do (DO) Instruction

The DO instruction provides a form of conditional and multiple assembly capability. It has the form:

NAME	OPERATION	OPERAND
A symbol (optional)	DO	A symbol or expression

The DO instruction causes the statement immediately following it to be assembled as many times as specified by the value of the operand. The value of the operand must be between 0 and 32,767. If the value of the operand is 0, the next instruction is skipped. If the operand contains a forward reference, CAL/32 will perform any additional passes required to define all symbols.

The user must guard against circular location counter dependency, as in the following example:

```
A           EQU     *
            DO      B-A
            DS      2
B           EQU     *
            END
```

CAL/32 will flag an "M001 xxxTOP" error, where xxx is PURE, IMP, or ABS, depending upon the current location counter.

3.8.9 Instructions for Data Structures

These instructions allow the programmer to define complex data structures. Some of these instructions allow the programmer to define and initialize data blocks compatible with FORTRAN common.

3.8.9.1 Structure Definition (COMN, STRUC, ENDS) Instructions

Structure definition instructions are used to define data structures. They have the form:

```
    NAME            OPERATION       OPERAND

A symbol            COMN            Not used
(optional)                          (ignored)
A symbol            STRUC           Not used
(optional)                          (ignored)
A symbol            ENDS            Not used
(optional)                          (ignored)
```

The common instruction (COMN) defines FORTRAN compatible common blocks. The structure instruction (STRUC) defines other types of data structures. The end structure instruction (ENDS) terminates both common definitions and data definitions.

The symbol in the name field of a COMN or STRUC statement contains the absolute value of the length of the structure or common block. The symbol specified with the ENDS instruction is associated with the current value of the offset counter.

A symbol is always required in the name field of a COMN instruction. To define FORTRAN compatible blank common, a special symbol consisting of two slashes (//) must appear in the first two positions of the name field. The remaining positions must be blank. If the name field is blank, CAL/32 will assume (//) was intended for a FORTRAN blank common.

The scope of the common block consists of all the storage definitions between the COMN instruction itself and the next ENDS statement. Only define storage, origin, and equate instructions are permitted between a COMN and its corresponding ENDS instruction. The define storage instructions included within the common block definition do not actually reserve storage; they define offsets within the common block. Origin statements can be used to modify the offset counter. The equate instructions can be used to define symbols relative to elements in the common block. Common blocks cannot be nested within other common blocks or within other structure definitions.

The following is an example of the definition of FORTRAN compatible common blocks:

```
C       FORTRAN PRCGRAM
        INTEGER*2 I,J,K,KK,K2,L
        COMMON A(10), I, J(3,20)
        COMMON/COMONE/B(30), K(4), KK
        CCMMON/COMTWO/X,Y,Z,K2,L(24)
```

The CAL/32 code to define these common blocks is:

```
//        COMN              DEFINE BLANK COMMON
A         DS    40          TEN FLOATING POINT NUMBERS
I         DS    2           ONE TWO-BYTE INTEGER
J         DS    120         SIXTY TWO-BYTE INTEGERS
          ENDS              END OF BLANK COMMON DEFINITION
COMONE    COMN              DEFINE COMMON BLOCK COMONE
B         DS    120         THIRTY FLOATING POINT NUMBERS
K         DS    8           FOUR TWO-BYTE INTEGERS
KK        DS    2           ONE TWO-BYTE INTEGER
          ENDS              END COMMON BLOCK COMONE
COMTWO    COMN              DEFINE COMMON BLOCK COMTWO
X         DS    4           ONE FLOATING POINT NUMBER
Y         DS    4           ONE FLOATING POINT NUMBER
Z         DS    4           ONE FLOATING POINT NUMBER
K2        DS    2           CNE TWO-BYTE INTEGER
L         DS    48          TWENTY FOUR TWO-BYTE INTEGERS
          ENDS
```

Common block definitions must precede any statements that reference the common block. Referencing a common element plus a displacement is permitted in the operand of a machine instruction, in a define constant instruction, or in a block data origin instruction defined below.

STRUC is used to define general purpose data structures. The scope of this data structure consists of all the storage definitions between the structure instruction and its corresponding ENDS instruction. Only define storage, origin, and equate instructions can be used in a structure definition. The define storage instructions do not actually reserve storage; they define offsets within the data structure. Origin statements can be used to modify the value of the offset counter. Equate statements can be used to define names relative to elements in the data structure. Data structures cannot be nested within other data structure definitions or within common block definitions.

To define a linked list structure, each node of which contains a 2-byte forward pointer, a 2-byte backward pointer, six bytes, and a set of values such as: four bytes, one byte, one byte and six bytes, the programmer might write:

```
NODE        STRUC
FWD         DS      2                   DEFINE FORWARD POINTER
BAK         DS      2                   DEFINE BACKWARD POINTER
VALA        DS      4                   DEFINE FOUR-BYTE VALUE
VALB        DS      1                   DEFINE ONE-BYTE VALUE
VALC        DS      1                   DEFINE ONE-BYTE VALUE
VALD        DS      6                   DEFINE SIX-BYTE VALUE
            ENDS
```

The effect of this definition is the same as:

```
NODE        EQU     16
FWD         EQU     0
BAK         EQU     2
VALA        EQU     4
VALB        EQU     8
VALC        EQU     9
VALD        EQU     10
```

Once NODE is defined, it can be used as follows:

```
            .
            .
            .
            LHI     R5,POOL             GET ADDRESS OF POOL
            LB      R0,VALB(R5)         GET VALUE B OF FIRST NODE
            LH      R5,FWD(R5)          GET POINTER TO NEXT NODE
            .
            .
            .
POOL        DS      100*NODE
            .
            .
            .
```

Data structure definitions must precede any references to their elements in RX3 format instructions, unless the NORX3 instruction or the SQUEZ instruction was used.

3.8.9.2 Structure Initialization (BDATA, BORG) Instructions

Structure initialization instructions define FORTRAN compatible block data subprograms that consist of labeled common blocks. They have the form:

```
NAME         OPERATION    OPERAND

A symbol       BDATA       Not used
(optional)                 (ignored)
A symbol       BORG        Not used
(optional)                 (ignored)
```

The block data instruction (BDATA) must precede any statements that generate data, and the block data subprogram must not contain any executable code. The common blocks to be initialized must be defined at the beginning of the block data subprogram. Once they are defined, the block origin instruction (BORG) is used to initialize the data elements of the common blocks. The operand of the block origin instruction consists of the common block name followed immediately by the element name or its displacement enclosed in parentheses. Only one operand is allowed. The following is an example of a block data subprogram:

```
                    BDATA
          *
          *                     COMMON BLOCK DEFINITION
          *
          BLK    CCMN
A         DS     4
B         DS     40
Y         DS     20
Z         DS     4
          ENDS
          *
          *                     INITIALIZE ELEMENTS A, B+8, AND Z
          *
          BORG   BLK(A)         REFERENCE BY NAME
          DC     E'10'
          BORG   BLK(64)        REFERENCE BY DISPLACEMENT
          DC     E'20'
          BORG   BLK(B+8)       REFERENCE BY NAME AND
                                DISPLACEMENT
          DC     E'30'
          END
```

This program initializes A to a floating point value of 10; Z to a floating point value of 20; and the third fullword, B, to a floating point value of 30.

3.8.10 Listing Control Instructions

These instructions allow the programmer to exercise some control over the format and the content of the source listing produced by CAL/32 on the final pass of the assembly.

3.8.10.1 Listing Identification (PROG, TITLE) Instructions

Listing identification instructions are used to force CAL/32 to print header information at the top of each page of the source listing. They have the form:

```
   NAME          OPERATION  OPERAND

   A symbol      PROG       Text
   (optional)
   A symbol      TITLE      Text
   (optional)
```

The program instruction (PROG) specifies the primary heading for each page of the listing. In addition, it causes the symbol in the name field to be placed at the beginning of the object file for program identification. On 16-bit assemblies, only the first six characters of the name field are put in the object file.

All characters in the operand field (a maximum of 56) up to and including position 71 are printed in the primary header line of each page of the listing. If more than one PROG instruction is encountered in a module, the last PROG instruction will override all previous ones.

The title instruction (TITLE) is a way to specify subheadings that can be changed within the program. The text contained in the operand field up to and including position 71, is printed on the line immediately below the heading produced by the PROG instruction. As many TITLE instructions as required can appear in the source input file. Each time a TITLE instruction is encountered, CAL/32 starts a new listing page with the new subheading when the next printable statement is processed. Subsequent pages contain this same subheading, until another TITLE instruction appears. If two or more TITLE instructions occur together in sequence, only the last TITLE instruction affects the subheading content since a new page will be printed only when a printable statement is encountered.

TITLE instructions themselves are not printed although they are included in the statement count.

3.8.10.2 Format Control (LCNT, EJECT, SPACE, WIDTH) Instructions

Format control instructions allow the programmer to control the format of the listing. They have the form:

NAME	OPERATION	OPERAND
A symbol (optional)	LCNT	A symbol or expression
A symbol (optional)	EJECT	A symbol or expression
A symbol (optional)	SPACE	A symbol or expression
A symbol (optional)	WIDTH	A symbol or expression

The operand field of the line count instruction (LCNT) specifies the number of lines to be printed on each page of the listing. The operand value must be an absolute number no greater than 99 and no less than 10. The default value of the line count is 58.

Whenever the eject instruction (EJECT) appears, it overrides the specified or default line count, and causes CAL/32 to start a new page when the next printable statement is processed. The new page starts with whatever headings are in use. This statement is included in the statement count, but it is not printed. If one or more EJECT instructions occur together in sequence, only one page is advanced since the actual advance occurs only when a printable instruction is encountered. EJECT instructions themselves are not printed although they are included in the statement count.

The operand field of the space instruction (SPACE) specifies the number of lines to be skipped in the listing. The value of the operand must be absolute. If the number of lines to be skipped exceeds the number of lines remaining on the page, this instruction has the same effect as an EJECT instruction and is included in the statement count, but not printed.

The operand field of the width instruction (WIDTH) specifies the number of columns to be printed across the page. The value of the operand field must be an absolute number, not greater than 132 and not less than 64. The default value is 132.

3.8.10.3 Content Control (NLIST) Instructions

The content control instructions control the content of the listing. They have the form:

NAME	OPERATION	OPERAND
A symbol (optional)	NLIST	Not used (ignored)
A symbol (optional)	LIST	Not used (ignored)
A symbol (optional)	LSTC	Not used (ignored)
A symbol (optional)	NLSTC	Not used (ignored)
A symbol (optional)	ERLST	Not used (ignored)
A symbol (optional)	LSTM	Not used (ignored)
A symbol (optional)	NLSTM	Not used (ignored)
A symbol (optional)	FREZE	Not used (ignored)
A symbol (optional)	NFREZ	Not used (ignored)
A symbol (optional)	CROSS	Not used (ignored)
A symbol (optional)	NCROS	Not used (ignored)
A symbol (optional)	LSTUR	Not used (ignored)
A symbol (optional)	NLSTU	Not used (ignored)
A symbol (optional)	WARN	Not used (ignored)
A symbol (optional)	NWARN	Not used (ignored)

The no list instruction (NLIST) suppresses listing of the source program. Only those statements that contain errors are printed.

The list instruction (LIST) reverses this situation, and all source statements are printed. The assembler default is to print all source statements.

The list conditionals instruction (LSTC) permits printing of unassembled conditional assembly statements. This is the normal default mode of the assembler.

The no list conditionals instruction (NLSTC) suppresses printing of unassembled conditional statements.

The error list instruction (ERLST) causes CAL/32 to print all assembly errors by type, along with number of each statement on which the error occurred, immediately after symbol table listing.

The list macro instruction (LSTM) permits printing of all macro expansions that are part of the source input file. The macro instruction, the expanded source code, and the generated object code are printed. A plus character (+) precedes each statement number in the expanded source to identify those statements as part of a macro. This is the normal mode of the assembler.

The no list macro instruction (NLSTM) suppresses printing of macro expansions. Only the macro statement itself is printed.

The freeze (FREZE) instruction halts incrementing of the statement counter when a copy file or macro expansion are included in the source input file. All statements in the copy file or macro expansion receive the same statement number as that of the COPY instruction. This is the normal mode of the assembler.

The no freeze (NFREZ) instruction increments the statement counter for every statement encountered in the source input.

The cross reference (CROSS) instruction uses CAL/32 to generate and print a cross reference listing of all the symbols used in the program. Each symbol is listed in alphabetical order, along with identification of the statements in which it is referenced. The statement in which it is defined is flagged with an asterisk. This is the normal mode of the assembler.

The no cross (NCROS) instruction prevents the generation of a cross reference listing.

The list unreferenced symbols (LSTUR) instruction causes unreferenced symbols to be listed in the symbol list. This is the normal mode of the assembler.

The no list unreferenced symbols (NLSTU) instruction suppresses the listing of unreferenced symbols in the symbol list.

The warning (WARN) instruction allows CAL/32 to flag warnings in the listing and tally the number of warnings encountered. This is the normal mode of the assembler.

The no warning (NWARN) instruction suppresses both the warnings and the warning count from the listing.

3.9   ASSEMBLY LISTING

The assembly listing consists of two sections: the source and object program statements and the symbol cross reference table. The format for printing the source and object program statements is basically the same for either 16-bit assemblies or 32-bit assemblies. The only difference is in the number of characters printed for the location counter and the object data.

- In 16-bit assemblies, only four hexadecimal digits are printed for the location counter, and a maximum of eight hexadecimal digits for the data. The letter R is appended to the location counter value if the relocatable location counter is being used.

- In 32-bit assemblies, six hexadecimal digits are printed for the location counter and a maximum of 12 hexadecimal digits for the object data. In addition, the actual second operand address of RX2 and SF instructions is printed next to the object data. This address is preceded by an equal sign (=). The letter I is appended to the location counter if the impure location counter is being used. The letter P is appended to the location counter if the pure location counter is being used.

- In both 16- and 32-bit assemblies, the letter F is appended to the data field to indicate that the statement references an externally defined symbol, a symbol in a common block, or an undefined symbol.

The statement number is a decimal number between 1 and 99999. Each source statement read by the assembler is assigned a unique statement number, beginning with 1, except for source statements from a copy file or macro expansion with the FREZE instruction. The first column of the listing can contain any of the following characters:

CHARACTER   MEANING

!   The name field of this instruction contains a
              symbol that was redefined by an EQUATE
              instruction.

?   A machine instruction not available on the
              target machine was used; an equivalent
              instruction existed and was substituted, or

              a machine dependent instruction was used in
              assembling a common mode program, or

              an assembler instruction was used with an
              improper but assemblable operand, or

              a SCRAT card was encountered as other than the
              first statement or when batch mode is in
              effect, or

              an EXTRN/ENTRY symbol is longer than 6
              characters for target 16, or

              a DS instruction was encountered in a pure
              section.

              * A machine instruction was shortened or
              modified by squeezing.

The following information is printed at the beginning of the cross reference listing:

● Start options in the START command

● The number of errors detected by the macro processor if the program assembled was generated by the macro processor.

● Number of CAL/32 errors and the page number of the last error

● Number of CAL/32 warnings and the page number of the last warning

● Number of passes

● Message indicating the use of symbol table paging to disk

● Message indicating abnormal termination of squeezing because of squeeze-induced errors

● Message indicating the amount of required table space

Following this, each symbol used in the program is listed in alphabetical order along with its value. If a cross reference was requested, the statement number of each statement containing a reference to the symbol is printed following the value. The statement number in which the symbol is defined is printed with an asterisk (*) following. Associated with each symbol is a flag used to indicate one of the following:

| | |
FLAG	MEANING
⌀	Properly defined local symbol
M	Multiply defined symbol
U	Undefined symbol
<	Entry symbol
<U	Undefined entry
>	Externally defined symbol
>M	Multiply defined external
**	Unreferenced external

The flag is printed in the first column of the line containing the symbol.

If an error is detected in a source statement, the following message is printed immediately after the error statement:

Annn

A indicates the general type of error, and nnn is a decimal number that further identifies the error. Appendix B contains a complete list of CAL/32 error codes.

CHAPTER 4
COMMON MODE PROGRAMMING

4.1 INTRODUCTION

A useful feature of Common Assembly Language/32 (CAL/32) is mode programming where a single source file can be used to produce object code for either 16-bit processors or 32-bit processors. In creating a common mode source file, the programmer must be aware of certain restrictions and safeguards and, in some cases, must use special operaticn mnemonics that can be translated into either 16-bit or 32-bit operations.

4.2 ADDRESS OPERATION INSTRUCTIONS

Addresses for 16-bit processors occupy 16 bits, a halfword. For the 32-bit processors, addresses occupy the least-significant 24 bits of a fullword. In normal mode, CAL/32 makes no distinction between operations on address quantities and operations on other data types. However, when writing in common mode, the programmer must use special operation mnemonics for address operations so CAL/32 can translate them into the correct target machine code. Table 4-1 lists these instructions, their mnemonics, and the target machine translations.

TABLE 4-1 COMMON MODE ADDRESS OPERATIONS

INSTRUCTION	MNEMONIC	32-BIT TRANS-LATION	16-BIT TRANS-LATION
Add Address	AA	A	AH
Add Address Immediate	AAI	AI	AHI
Add Address RR	AAR	AR	AHR
Add Address to Memory	AAM	AM	AHM
Compare Address	CA	C	CH
Compare Address Immediate	CAI	CI	CHI
Compare Address RR	CAR	CR	CHR
Compare Logical Address	CLA	CL	CLH
Compare Logical Address Immediate	CLAI	CLI	CLHI
Compare Logical Address RR	CLAR	CLR	CLHR
Immediate	CLAI	CLI	CLHI

TABLE 4-1   COMMON MODE ADDRESS OPERATIONS (Continued)

INSTRUCTION	MNEMONIC	32-BIT TRANS-LATION	16-BIT TRANS-LATION
Load Address	LDA	L	LH
Load Address Immediate	LDAI	LA	LHI
Load Address RR	LDAR	LR	LHR
AND Address	NA	N	NH
AND Address Immediate	NAI	NI	NHI
AND Address RR	NAR	NR	NHR
OR Address	OA	O	OH
OR Address Immediate	OAI	OI	OHI
OR Address RR	OAR	OR	OHR
Subtract Address	SA	S	SH
Subtract Address Immediate	SAI	SI	SHI
Subtract Address RR	SAR	SR	SHR
Shift Left Address Arithmetic	SLAA	SLA	SLHA
Shift Left Address Logical	SLAL	SLL	SLHL
Shift Right Address Arithmetic	SRAA	SRA	SRHA
Shift Right Address Logical	SRAL	SRL	SRHL
Store Address	STA	ST	STH
Test Address Immediate	TAI	TI	THI
Exclusive OR Address	XA	X	XH
Exclusive OR Address Immediate	XAI	XI	XHI
Exclusive OR Address RR	XAR	XR	XHR
Multiply Address	MA	M	MH
Multiply Address RR	MAR	MR	MHR
Divide Address	DA	D	DH
Divide Address RR	DAR	DR	DHR

CAL/32 translates these instructions into halfword or fullword instructions, depending on the target machine. For example:

```
              .
              .
              .
       LDA    R1,ADD1
       AA     R1,DISP
              .
              .
              .
ADD1   DC     A(TABLE)
DISP   DC     2
              .
              .
              .
```

When CAL/32 assembles these instructions for 16-bit execution, it produces object code that would normally correspond to:

```
        .
        .
        .
LH      R1,ADD1
AH      R1,DISP
        .
        .
        .
```

For 32-bit programs, CAL/32 produces code that would correspond to:

```
        .
        .
        .
L       R1,ADD1
A       R1,DISP
        .
        .
        .
```

Translation is at the object code level; CAL/32 prints the original common mode code on the listing.

4.3 COMMON MODE IMMEDIATE OPERATIONS

CAL/32 provides a common mode immediate operation for the load immediate LDI instruction. Depending on the target machine, the LDI is translated into a fullword-referencing LI instruction for the 32-bit machine, or a halfword-referencing LHI instruction for the 16-bit machine, as follows:

INSTRUCTION	COMMON MNEMONIC	32-BIT TRANSLATION	16-BIT TRANSLATION
Load Immediate	LDI	LI	LHI

4.4 COMMON MODE ASSEMBLER INSTRUCTIONS

In addition to all of the regular assembler instructions described in Chapter 3, CAL/32 recognizes four assembler instructions primarily for use in common mode programming. Two of these are data definition type instructions; the other two are assembler control type instructions.

4.4.1 Data Definition Instructions

The common mode data definition instructions are: define address length constant and define address length storage. They have the form:

NAME	OPERATION	OPERAND
A symbol (optional)	DAC	One or more operands separated by commas
A symbol (optional)	DAS	A symbol or expression

4.4.1.1 Define Address Length Constant Instruction

The define address length constant instruction is equivalent to the define constant instruction. It is used in common mode programming to reserve storage to be initialized with address length constants. For 32-bit assemblies, the constants are fullwords aligned on fullword boundaries. For 16-bit assemblies, the constants are halfwords aligned on halfword boundaries.

4.4.1.2 Define Address Length Storage Instruction

The define address length storage instruction is equivalent to the define storage instruction. In 32-bit assemblies, the instruction reserves the specified amount of fullwords aligned on a fullword boundary. In 16-bit assemblies, it reserves the specified amount of halfwords aligned on a halfword boundary. Examples of the use of these instructions are:

```
        .
        .
        .
DAC     A(TABLE)
DAS     16
        .
        .
        .
```

When assembled for 32-bit execution, the define address length constant instruction generates a fullword containing the address of TABLE. The define address length storage instruction reserves 16 fullwords of storage. When assembled for 16-bit execution, these instructions cause CAL/32 to generate a halfword containing the address of TABLE, along with a storage area of 16 halfwords.

NOTE

Define address length storage instructions can be used in common block and structure definitions.

4.4.2 Assembler Control Instructions

Two special assembler instructions control error checking. Their form is:

NAME	OPERATION	OPERAND
A symbol (optional)	CAL/32	Not used (ignored)
A symbol (optional)	NOCAL	Not used (ignored)

The first of these instructions (CAL/32) establishes the common mode and enables common mode error checking. In this mode, any machine dependent instruction causes a nonfatal error, and a warning flag is printed on the assembly listing.

The NOCAL/32 instruction disables the common mode and its error checking mechanisms until the next CAL/32 instruction is encountered. This is the assembler default mode in which an operation code mnemonic, not valid for the targeted processor but for which there is a valid equivalent, is assembled using the valid equivalent. A question mark (?) is then printed in the left hand margin of the listing.

4.5 MIXED MODE COMPUTATIONS

On 32-bit processors, mixed mode computations, such as adding a halfword quantity to an address length quantity contained in a register, can be performed. In general, any halfword arithmetic or logical operation can be performed on address length quantities contained in registers. The exceptions are: shifts, multiply, and divide. The halfword forms of these instructions should never be used with address length quantities. Instead, use the special address operation instructions.

4.6 GLOBAL SYMBOLS

The global symbols, ADC and LADC, are used primarily in common mode programming. In 32-bit assemblies, ADC has a value of four, the length in bytes of an address length constant. LADC has a value of two, the log (base 2) of the address length. In 16-bit assemblies, ADC has a value of two, and LADC has a value of one. Illustrated are these symbol uses in which a main program calls a subroutine and passes parameters to the subroutine in a list of addresses immediately following the branch and link instruction:

```
             .
             .
             .
             BAL     RF,SUB
             DAC     A(PARM1),A(PARM2),A(PARM3)
RETURN       EQU     *
             .
             .
             .
```

The subroutine picks up the parameters and calculates the return address as follows:

```
SUB          AIS     RF,LADC           ADJUST RF FOR
             NAI     RF,-ADC           ALIGNMENT
             LDA     R1,0(RF)          ADDRESS OF FIRST PARAMETER
             LDA     R2,ADC(RF)        ADDRESS OF SECOND PARAMETER
             LDA     R3,2*ADC(RF)      ADDRESS OF THIRD PARAMETER
             .
             .
             .
SUBEND       B       3*ADC(RF)         RETURN TO CALLER
             .
             .
             .
```

The add immediate short instruction and the add address immediate instruction are needed in the subroutine because alignment of address constants in 32-bit assemblies can cause a halfword of filler to be inserted between the branch and link instruction and the first address constant. In this case, the address in register 15 is the address of this halfword, and these instructions increment the address in register 15 to make it point to the first address constant. If no filler is required, because the first constant is naturally aligned on a fullword boundary, register 15 points to the first constant, and these two instructions have no effect.

Another use of LADC is in shift instructions where a byte pointer
must be converted into an address pointer, as:

```
        .
        .
        .
LB      R1,INDEX            GET BYTE POINTER
SLAL    R1,LADC            CONVERT TO ADDRESS POINTER
LDA     R2,TABLE(R1)       GET ADDRESS FROM TABLE
BR      R2
        .
        .
        .
```

In 16-bit assemblies, LADC has a value of one, and the shift left
logical instruction has the effect of doubling the value of the
byte pointer, converting it into a halfword pointer. In 32-bit
assemblies, LADC has a value of two, and the shift instruction
has the effect of quadrupling the value of the byte pointer,
converting it into a fullword pointer.

The LADC symbol can also be used where machine dependent code
must be written within a common mode program. For example:

```
        .
        .
        .
IFNZ    LADC-1             IF TRUE USE 32 BIT CODE
L       RF,A               LOAD FULLWORD IN RF
A       RF,B               ADD FULLWORD B
ST      RF,A               STORE IN A
ELSE                       LADC-1 IS FALSE USE 16 BIT
LM      RE,A               LOAD FULLWORD IN RE AND RF
AH      RF,B+2             ADD LOW ORDER B
ACH     RE,B               ADD HIGH ORDER B
STM     RE,A               STORE IN A
ENDC
        .
        .
        .
```

shows how fullword addition, requiring double registers in 16-bit
assemblies and single registers in 32-bit assemblies, can be
handled in a common mode program.

4.7 SPECIAL INSTRUCTIONS

By definition, the instructions load multiple, store multiple, and load PSW, operate on address length data. This is why there are no address operation mnemonics for these instructions. Where these instructions are used in common mode programming, the data on which they operate must be defined by the define address length constant and the define address length storage instructions. For example:

```
                    .
                    .
                    .
                    LPSW NEWPSW
                    .
                    .
                    .
    START           STM     R0,SAVE
                    LM      R0,PARAM
                    .
                    .
                    .
    NEWPSW          DAC     STATUS,A(START)
    RSAVE           DAS     16
    PARAM           DAC     CON1,CON2,...
                    .
                    .
                    .
```

List processing instructions operate on address length quantities within the list. There is some incompatibility between the 16- and the 32-bit versions of these instructions. The 16-bit list instructions require byte pointers at the head of the list. The 32-bit list instructions require halfword pointers. List instructions can be used in common mode programming as long as the number of slots in the list does not exceed 255.

Lists always should be defined with the define list instruction.
Use byte instructions where it is necessary to refer to the list
pointers in the program. Define displacement into the list
pointer fields in terms of the LADC symbol. For example:

```
             •
             •
             •
SLOTS    EQU    LADC-1                    NUMBER OF SLOTS
USED     EQU    2*LADC-1                  NUMBER USED
CTOP     EQU    3*LADC-1                  CURRENT TOP
NBOT     EQU    4*LADC-1                  NEXT BOTTOM
             •
             •
             •
         LB     R1,LIST+CTOP
             •
             •
             •
LIST     DLIST 32
             •
             •
             •
```

In this example, the load byte instruction is used along with the
value of CTOP to access the current top pointer in the list.

APPENDIX A
COMMON ASSEMBLY LANGUAGE/32
(CAL/32) OPERATING INSTRUCTIONS

The CAL/32 assembler requires a minimum of one lu and up to a maximum of 11 logical units for operation, depending on the options selected and the features invoked by the source program. All of these logical units can be assigned by the user. However, if an lu is needed and not assigned, CAL/32 will allocate temporary system files for logical units 4, 5, 6, 8, 9, 12, and 13. CAL/32 will delete and reallocate permanent files for logical units 2 and 3, provided they were not previously assigned and the DEL start option was specified. The logical units used are:

LU	USE	LOGICAL RECORD	ALLOCATED BY CAL/32	REQUIRED FOR
1	Source input device. The source input to be assembled is read from this device on pass one. This device is re-wound prior to each subsequent pass unless BATCH is specified and the source input is not on a random access device, or SCRAT or PPAUS is specified.	80	No	All
2	Binary output device. Assembled object pro-gram is written to this device on last pass.	108 T=16 126 T=32	If DEL specified	All
3	Assembly listing output device. Assembly list-ing is written to this device on the last pass.	64 - 132	If DEL specified	All
4	Source scratch device. The source input is copied to this device during pass one. The source input is read from this device on all subsequent passes.	80	Yes	SCRAT BATCH

LU	USE	LOGICAL RECORD	ALLOCATED BY CAL/32	REQUIRED FOR
5	Symbol cross reference scratch device. Cross reference information is built on this device during the last pass. A device assigned to this lu must support random access.	256	Yes	CROSS
6	Symbol table paging device. Symbol table information is paged to this device during all passes. A device assigned to this lu must support random access.	512	Yes	Insufficient memory
7	Source library input device. Source information to be included in the main assembly is read from this device on each pass unless SCRAT or BATCH was specified. Then the library is searched and read on pass one only.	80	No	COPY
8	Forward equate scratch device. This lu can be used if forward referenced equates exist in the source input. This device must support random access.	256	Yes	Forward equates
9	Error tabulation device. Error messages and their associated line numbers are written in binary to this device during the last pass and written to lu 3 after completion of the assembly and symbol table listing.	80	Yes	ERLST

LU	USE	LOGICAL RECORD	ALLOCATED BY CAL/32	REQUIRED FOR
12	PCB file directory scratch device. This device must support random access.	256	Yes	CLIB
13	PCB name directory scratch device. This device must support random access.	256	Yes	COPY

When an assembly terminates, an end of task code is passed to the operating system in the operand field of the SVC 3 instruction. The meanings of the possible end of task codes are:

END OF TASK CODE	MEANING
0	Assembly complete without errors.
1	Illegal option passed with the START command. Assembly is aborted after logging the illegal options to the console. The user should retry.
2	One or more errors detected during the assembly. This end of task code is also used if errors are detected in one or more programs of a batch assembly.
3	Misplaced BEND.
4	Symbol table overflow.

When operating under OS/32, CAL/32 accepts certain control options as arguments of the START command. Any combination of spaces and/or commas can separate or follow the options specification:

```
START
OPTION          OPERANDS


PPAUS           None
CROSS           None
NLIST           None
NORXT           None (alias for NORX3)
SQUEZ           Number of passes (1-99)
SQCHK           None
SCRAT           None
TARGT           16 or 32
WIDTH           Width of listing
LCNT            Lines per page (10-99)
ERLST           None
ERSQZ           None
NLSTC           None
CAL             None
NLSTM           None
BATCH           None
FREZE           None
NDISK           None (inhibits symbol table paging to disk)
NFIX            None (prevents CAL/32 from making
                      extra passes to fix squeeze-induced errors)
DEL             None
LIST            None
LSTC            None
LSTM            None
LSTUR           None
UREX            None
WARN            None
NCROS           None
NDEL            None
NFREZ           None
NLSTU           None
NOCAL           None
NOSQZ           None
NOSEQ           None
NUREX           None
NWARN           None
NORX3           None
```

Start options have the following form:

    option [=operand]

A typical start command for a CAL/32 assembly with start options
is:

      ST ,DEL,SQUEZ=99,NCROSS

The delete start options (DEL, NDEL) enable or disable CAL/32
from deleting and reallocating object and listing files when
needed. If the DEL option is in effect and lu 2 and lu 3 are
unassigned, CAL/32 will delete, reallocate, and assign them to
fname.OBJ and fname.LST, respectively. The default option is
NDFL, in which case CAL/32 will simply log an 8100 error to the
console and pause. In the event that lu 1 is not assigned to a
direct access device, the DEL option will have no effect, and
CAL/32 will issue an 8100 error before pausing.

When CAL/32 encounters conflicting start options such as CROSS
and NCROS, it will regard the last option encountered as the
intended option. This allows the user to redefine the default
start options via CSS. For example:

      LO CAL32
      AS 1,SOURCE.CAL
      ST ,NCROSS DEL @1
      $EXIT

The above CSS effectively changes the default options to NCROSS
and delete unless overridden by the parameter @1.

OPERATING INSTRUCTIONS FOR OS/32

CAL/32 will not run on a 16-bit machine, however it will still
produce 16-bit object code if requested.

Before using CAL/32 under OS/32, the relocatable object supplied
must be established as an OS/32 task, using Link. A typical
command sequence using Link to establish CAL/32 is:

      LO .BG,LINK
      T .BG
      ST
      >ES TA
      >OP work=5000, SYS=FFFFF, SEG,ROL
      >IN CAL32
      >BU CAL32
      >END

CAL/32 is segmented into pure and impure code for shared use with OS/32 systems that support this capability. To establish CAL/32 as a nonsharable task, remove the SEG option from the above command sequence. TET can be used instead of Link provided the source does not contain any of the following instructions: external with offset, DNTRY, WNTRY, WXTRN, INCLD, DCMD.

A typical command sequence to establish CAL/32 using TET is:

```
LO  .BG,TET32        Load TET32
AS  5,CON:           Assign interactive device
AS  7,CON:
AL  TETSCRT,IN       Allocate and assign scratch
AS  4,TETSCRT
ST
ES  TA               Establish task command
IN  CAL32.OBJ        Read CAL/32 object
EXP 80               Get 20kb for symbol table
BU  TA,CAL/32        Build CAL/32 task
MAP PR:              Obtain map
END
```

The established CAL/32 task can then be loaded with the LOAD command and devices or files assigned as required. For example:

```
LO  .BG,CAL32
T   .BG
AS  1,PROGRAM.CAL,SRO  Source input
AS  3,PR:              Assign listing device
ST  ,DEL,SQUEZ         Start assembly
```

When assembly is completed, CAL/32 terminates through the operating system, which logs the message:

    END OF TASK n

where n is the end of task code.

The files used for scratch, cross reference, paging, forward equates, PCB file directory, PCB name directory, and error summary will be allocated by CAL/32 as temporary operating system files if they are needed and were not previously assigned by the user.

APPENDIX B
COMMON ASSEMBLY LANGUAGE/32 (CAL/32) ERROR CODES

A001   the address   The address is out of range for the specified instruction format.

A002   the address   The address is out of range for an RX2 instruction.

A003   the operand   The operand of a previously squeezed instruction was changed making the squeezed instruction invalid.

B001   alignment   The address of the operand is on an incorrect boundary for the instruction specified.

B002   alignment   An odd address used in a T constant location counter was not even when the instruction was specified.

C001   common mode   An opcode that is not part of the common mode set is used in a common mode assembly.

D001   data structure   An illegal statement appears in a STRUC or COMN definition.

E001   END placement   An END statement was encountered within a STRUC or COMN definition or within an unterminated conditional.

F001   missing operand

F002   register   A register value is not in the
        specification   range of 0 to 15, or an odd register value is used where an even value is required.

F003   invalid source   Invalid label in the source field,
        field   a label in the name field is not followed by a space, or a required label is missing; e.g., on EQU.

F004   invalid symbol   More than 8 characters were specified in a symbol.

F005	EXTRN	An invalid type for EXTRN; e.g., common block, or EXTRN was used in an expression.
F006	immediate field	The value of data is too large to fit into the immediate field. A fullword EXTRN is used in RI1 instruction. A character string used as an immediate field is too long.
F007	ENTRY	A symbol declared as an ENTRY is undefined. Improper type for ENTRY; e.g., common block name.
F008	delimiter	Operands are not separated by commas. Unrecognizable operator. The last operand is not followed by a CR or a blank. Unbalanced parentheses. Opcode is not followed by a space or a CR.
F009	invalid expression	Expression uses common element names not in the same block.
F010	apostrophe	No ending apostrophe in C,D,E,F, H,P,U,X, or Y constant. Illegal character encountered in C,D,E,F,H,P,U,X, or Y constant prior to the ending apostrophe.
F011	invalid operand	T constant was specified in TARGT 16 assembly. Argument mode of T constant is not ABS, PURE, or IMPURE. Illegal data specified in BDATA program. Fullword EXTRN used as an operand of DCZ. Value of DB operand must be absolute. Value of DS, DSF, DSH. Invalid symbol used for ENTRY name. Symbol used as ENTRY must be ABS, PURE, IMPURE, or Relocatable. Invalid symbol used for EXTRN name. Invalid data in BORG. Operand of CNOP or ALIGN is not absolute. Operand of DLIST is not absolute.
		Improper type for EXTRN operand; e.g., common block name. Transfer address on END statement is an improper type; e.g., EXTRN. Illegal operand on EQU.
F012	improper statement	Label only

F013	file descriptor	Syntax error on fd of a COPY, FCOPY, or CLIB statement
F014	missing string	No characters between apostrophes of C,E,D,F,H,P,U,X or Y constant
F015	invalid character	Illegal character encountered between apostrophes of an E or D constant.
I001	conditional	An ELSE or ENDC statement found without a matching IFx
M001	symbol definition	The symbol in the name field is also used in the name field of another statement. The value or type of a symbol changed from its definition on a previous pass. (This can occur by illegal use of conditionals, ORG, DO, DS, or a misplaced SCRAT statement.)
M002	symbol definition	An attempt was made to redefine a symbol with an EQU that is the name field of a statement.
O001	illegal opcode	The opcode used is totally unrecognizable or illegal for the specified TARGT.
P001	location counter	The location counter exceeded 216 on a TARGT-16 or 224 on a TARGT-32 assembly.
P002	reentrancy check	The instruction attempts to modify PURE code.
R001	relocation error	An invalid combination of relocatable terms in an expression. A relocatable operand follows a unary minus.
S001	sequence check	The value in the sequence numbers field is not greater than the previous sequence number.
S002	COPY	COPY statement appears within a file being copied. An invalid symbol used as COPY operand. The operand of COPY is not followed by a space, comma, or CR.
S003	invalid option sequence	A COPY, PAUSE, MSG, or DO statement immediately follows a DO statement.

S004	invalid option	An argument is not absolute or exceeds 32767. An argument of LCNT is in the range of 10 to 99. An argument of WIDTH is not in the range of 64 to 132. An argument of TARGT does not evaluate to either 16 or 32. An argument of SQUEZ is not in the range of 1 to 99.
S005	PROG	Multiple PROG statements were encountered in a program.
T001	overflow	The intermediate or final result of an arithmetic expression exceeded $2^{31} - 1$.
T002	floating point	An overflow occurred during conversion of floating point constant.
T003	value	The data item exceeds the range for specified type; e.g., X'12345'.
T004	divisor	A division by 0 is attempted.
U001	not used	
U002	undefined symbol	A referenced symbol is not defined in the program.
U003	undefined symbol	An attempt was made to circularly define a symbol; e.g.:

```
A EQU B
B EQU A
```

U004		File specified as an operand of FCOPY, CLIB, or COPY does not exist.
U005		Program name is not found in any of the PCB libraries.

APPENDIX C
PERKIN-ELMER OBJECT CODE FORMAT

Modules in Perkin-Elmer object code format produced by CAL/32 are
divided into records. Each record contains 126 bytes of
information for 32-bit object code, or 108 bytes of information
for 16-bit object code. The first 4 bytes of each record of the
object code format are organized as follows:


```
   ---------------{ {-----------------------------{ {------------
  |     Sequence number        |          Checksum            |
   ---------------{ {-----------------------------{ {------------
Bits:
0                           15 16                            31
```
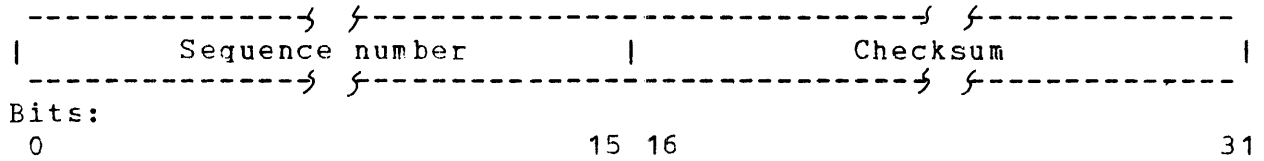

The sequence numbers are sequential negative integers -1, -2, -3,
etc., represented in two's complement form. The first record in
a program must have sequence number -1. Subsequent records must
be in proper order to be loaded.

The checksum is an exclusive OR sum of all halfwords in the
record, except itself, plus a halfword of all 1's.

The remainder of the record is a sequence of items; an item is a
byte of loader information. There are two types of items--loader
items and data items. Each loader item is followed by a certain
number (which can be 0) of data items. The loader items and
their meanings are listed in Tables C-1 and C-2.

TABLE C-1   32-BIT LOADER ITEM DEFINITIONS

LOADER ITEM	MEANING	NUMBER OF DATA ITEMS FOLLOWING
0	End of record	None
1	End of program	None
2	Reset sequence number	None
3	Block data indicator	8-byte name, 3-byte displacement, any absolute data item (20-5B)
4	Absolute program address	3-byte address
5	Pure relocatable program address	3-byte address
6	Impure relocatable program address	3-byte address
7	2 bytes of pure relocatable data	2-byte address
8	2 bytes of impure relocatable data	2-byte address
9	4 bytes of pure relocatable data	4-byte address
A	4 bytes of impure relocatable data	4-byte address
B	Common reference	8-byte address 3-byte displacement
C	EXTRN	8-byte name, followed by item 4, 5, or 6
D	ENTRY	8-byte name followed by item 4, 5, or 6
E	Common definition	8-byte name followed by a 3 byte length
F	Program label	8-character name
10	3 bytes absolute and 3 bytes pure relocatable	6 bytes
11	3 bytes absolute and 3 bytes impure relocatable	6 bytes
12	Load program transfer	Item 4, 5, or 6
13	Define start of chain (reference)	Item 4, 5, or 6
14	Load chain definition address	Item 4, 5, or 6
15	2 bytes absolute and 2 bytes pure relocatable	4 bytes
16	2 bytes absolute and 2 bytes impure relocatable	4 bytes

TABLE C-1 32-BIT LOADER ITEM DEFINITIONS (Continued)

LOADER ITEM	MEANING	NUMBER OF DATA ITEMS FOLLOWING
17	Short form EXTRN	8-byte name and Item 4, 5, or 6
18	Length of impure and pure segments	3-byte impure length and 3-byte pure length
19	Perform fullword chain	None
1A	Perform halfword chain	None
1B	No operation	None
1C	2-byte pure translation table address	2 bytes
1D	2-byte impure translation table address	2 bytes
1E	Not used	N/A
1F	1 byte absolute data	1 byte
20	2 bytes absolute data	2 bytes
21	4 bytes absolute data	4 bytes
22	6 bytes absolute data	6 bytes
23	8 bytes absolute data	8 bytes
.	.	.
.	.	.
.	.	.
5B	120 bytes absolute data	120 bytes
5C-64	Future use	

65	Extended EXTRN reference	8-byte external symbol name
		1-byte flag
		  xxxx xx00 standard EXTRN
		  xxxx xx01 weak EXTRN
		  xxxx xx10 include EXTRN
		4-byte offset
		Item 4, 5, or 6

66	Extended entry	8-byte entry symbol
		1-byte flag
		  xxxx xx00 standard entry
		  xxxx xx01 data entry
		  xxxx xx10 weak entry
		Item 4, 5, or 6

67	Link commands	1-byte length 1-80 characters of command

TABLE C-2   16-BIT LOADER ITEM DEFINITIONS

LOADER ITEM	MEANING	NUMBER OF DATA ITEMS FOLLOWING
0	End of record	None
1	End of program	None
2	Perform chain	None
3	Toggle absolute/relocatable mode	None
4	Transfer address	2-byte address
5	Load program address (ORG)	2-byte address
6	Load reference address	2-byte address
7	Load definition value	2-byte address
8	2 bytes absolute data	2 bytes data
9	2 bytes relocatable data	2 bytes data
A	4 bytes absolute data	4 bytes data
B	2 bytes absolute and 2 bytes relocatable data	4 bytes data
C	EXTRN reference	6-byte name
D	ENTRY definition	6-byte name
E	Decode next item	Next item
E0	Declare common block	6-byte name 2-byte size
E1	Load common block definition value	6-byte name 2-byte offset
E2	2 bytes absolute block data	6-byte name 2-byte offset 2 bytes data
E3	4 bytes absolute block data	6-byte name 2-byte offset 4 bytes data
E4	Reset sequence number to -1	None
E5	1 byte absolute data	1 byte data
E6	1 byte absolute block data	6-byte name 2-byte offset 1 byte data
F	Program label	6-byte name

All items are given in hexadecimal. Note that item E is actually a compound item whose interpretation depends on the item it follows. Item E and the following item are considered a single control item, however, and cannot be split across object records. This effectively allows more than 16 different control items, though most of them require only 1 nibble.

INDEX

PUBLICATION COMMENT FORM

Please use this postage-paid form to make any comments, suggestions, criticisms, etc. concerning this publication.

From _____ Date _____

Title _____ Publication Title _____

Company_____ Publication Number _____

Address _____

_____

_____

FOLD   FOLD

Check the appropriate item.

☐ Error   Page No. _____ Drawing No. _____

☐ Addition   Page No. _____ Drawing No. _____

☐ Other   Page No. _____ Drawing No. _____

Explanation:

FOLD   FOLD

Fold and Staple
No postage necessary if mailed in U.S.A.

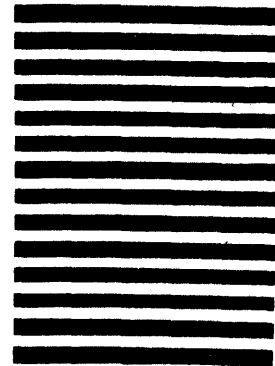**BUSINESS REPLY MAIL**

FIRST CLASS   PERMIT NO. 22   OCEANPORT, N.J.

POSTAGE WILL BE PAID BY ADDRESSEE

PERKIN-ELMER

Computer Systems Division
2 Crescent Place
Oceanport, NJ 07757

TECH PUBLICATIONS DEPT. MS 322A

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES