MAX-PLANCK-INSTITUT FÜR PLASMAPHYSIK

GARCHING BEI MÜNCHEN

A higher level assembler and a linkage editor for the PDP11 computer

Preliminary version

R.A. Pocock, A.W. Brown and R.F. Lathe.

IPP R/3 December 1971

Die nachstehende Arbeit wurde im Rahmen des Vertrages zwischen dem Max-Planck-Institut für Plasmaphysik und der Europäischen Atomgemeinschaft über die Zusammenarbeit auf dem Gebiete der Plasmaphysik durchgeführt.

IPP R/3

R.A. Pocock,
A.W. Brown and
R.F. Lathe

A higher level assembler and a linkage editor for the PDP11 computer (in English)

December, 1971

ABSTRACT:

Section 1 describes a higher level assembler PPL written at the Institut für Plasmaphysik and modelled on the IBM higher level assembler PL360. The assembler is designed to assemble source code on an IBM 360/91 computer, but to generate object code for the Digital Equipment Corporation small machine, the PDP11. The formulation of the language is described in detail.

Section 2 describes a linkage editor, also written at IPP, which is designed to resolve references between program segments produced by PPL and to provide as output a block of processed code suitable for loading by the PDP11 system loader.

C O N T E N T S

SECTION 1

THE PPL COMPILER

		Page
1.	THE PPL COMPILER	1
2.	DESCRIPTION OF THE LANGUAGE DEFINITION	2
2.1	The Metalanguage	2
2.2	Syntactic Elements	2
2.2.1	Terminal Strings	2
2.2.2	Generic Terminal Symbols	2
2.2.3	Scanning the Input	2
2.2.4	Syntactic Unit	3
2.3	Syntactic Operators	3
2.3.1	Juxtaposition	3
2.3.2	Alteration	3
2.3.3	Concatenation	4
2.3.4	Parenthesis	4
2.3.5	Sequence Operator	4
2.3.6	Void Alternative	5
2.3.7	Commentary	5
3.	PPL	6
3.1	Identifiers and Declarations	6
3.1.1	Values and Cells	6
3.1.2	Attributes	8
3.1.3	Cell Declarations	9
3.2	Assignment Statements	13
3.2.1	Arithmetic Assignments	14
3.2.2	Address Assignments	17
3.2.3	Logical Assignments	18
3.3	Control Facilities	19
3.4	Statements	19
3.4.1	Null Statement	19
3.4.2	If Statement	19
3.4.3	Case Statement	22
3.4.4	While Statement	23
3.4.5	For Statement	23
3.4.6	Blocks	25
3.4.7	Procedures	26
3.4.8	Procedure Call Statement	27
3.4.9	Return Statement	27
3.4.1o	Goto Statement	28
3.4.11	Functions	28

Page

3.5 Program Procedures 29
3.6 Identifier Block 3o
3.7 Program Deck 3o
3.8 Commentary 32
3.9 Scope, Allocation and Use of Identifiers 32
3.1o Errors and Error Recovery 34

4. PPL COMPILER OUTPUT 35

4.1 Printed Output 35

4.1.1 Obligatory Printed Output 35
4.1.2 Optional Printed Output 36

4.2 Object Code Output 38
4.3 Error report listing 39

APPENDIX A ERROR CODES 4o

SECTION 2

THE PDP11 LINKAGE EDITOR

Functions Performed 44
Space Allocation 44
Address Constant Relocation 44
External Reference Resolution 44
Linkage Editor Input 45
Linkage Editor Output 45
Linkage Editor Operation 45

APPENDIX A CONTROL CARDS 48
APPENDIX B OBJECT MODULE RECORD FORMATS 49
APPENDIX C RELOCATION DICTIONARY COMMANDS 51
APPENDIX D LOAD MODULE FORMAT 52

SECTION 1

The PPL higher level assembler

1. The PPL Compiler

Many research and development laboratories are acquiring small and medium sized computers for use in experiments requiring on-line data acquisition. These computers often come with their own software, but large research institutions usually have relatively large computer installations, which could be used more effectively in the generation of application programs for the on-line systems. The software programs used on the smaller computers are often slow, due to core size limitations and execution speed. These computers are also hindered by their on-line printers, which may be slow, noisy teletypes. The software of the small and medium machines is limited in its capabilities and usually lacks the helpful output of a compiler, which can be invaluable during debugging.

The power of the large computer can be harnessed for the generation of application software. Aids in the generation of application programs are non-host machine translators, ie. compilers and assemblers, which generate code for one machine (small or medium) but run on the large one. In addition, the other facilities of a large computer installation (file handling systems,off-line program storage etc.) can be of aid in generating application programs. Small computer users are then able to assemble or compile programs, including all the re-tries to correct simple coding mistakes, without touching a paper tape or having to wait for the teletype. Aids in the debugging of application software include simulators that run on the large machine.

The Institut für Plasmaphysik has ordered a PDP11 computer. The Institut also has an IBM 360 Model 91 which is used primarily for scientific applications. This situation allows an excellent application of the above: the power of the 360/91 can be used to facilitate the generation of application software for the PDP11. In particular, the power of the 91 can be used at compile/translate time for faster turn-around and for more useful output in program generation and debugging (such as variable cross-reference listings and formatted object code listings). The programming languages can be designed to force the application programmer to write clearer and more concise programs.

One of the programming languages designed to be compiled on the 360/91 for the PDP11 is PPL (pronounced PeoPle, for PDP11 Programming Language) which is patterned after PL/360. It is designed to have:

1) consistent grammar rules,
2) as few shorthand or obtruse mnemonics as possible,
3) to discourage the programmer from using tricks (which is accomplished by not allowing programs to modify themselves and by hindering mixed mode statements),
4) to have many of the characteristics of higher level languages - free format, block structures, and high level program-flow control statements.

The language, however, is not truly high level, but rather a higher assembler. PPL requires the programmer to have an intimate knowledge of the architecture of the PDP11 and its operational characteristics. In fact PPL, and any language like it which is intended for on-line data acquisition programming of small and medium computers, must provide access to/and use of all the facilities of the machine: special I/O interfaces require special code etc. Thus the programmer must be able to use all the addressing modes of the PDP11 simply; he must be able to express all the machine instructions conveniently, and he must be able to influence, if not control, storage allocation. At the same time, however, it is intended that PPL should obviate much of the drugery and many of the pitfalls of assembly language coding.

2. Description of the Language Definition.

The PPL compiler was generated with the aid of a metacompiler which uses an extended and modified form of Backus Nauer Form for the definition of the compiler. A subset of the metacompiler language (metalanguage) will be used to describe PPL.

2.1 The Metalanguage

The coding of PPL in the metalanguage consists of a set of syntactic equations. Each syntactic equation is made up of a syntactic variable (ie. the name of the syntactic unit), an equals sign, a syntactic expression on the right of the equals sign, and a semicolon as a termination symbol. The names of syntactic units must begin with a letter which may be followed by any number of letters or break characters (_). A part of the input string (ie. input to the PPL compiler) which satisfies the syntactic expression on the right of the equal sign is said to be an instance or example of that syntactic unit. A syntactic equation, then, describes a syntactic class (or the set of all allowable strings), by specifying the grammatical structure of those strings which are eligible for inclusion within the class.

2.2 Syntactic Elements

A syntactic expression (the right hand side of a syntactic equation) is made up of various components, the most primitive of which are the terminal string, and the generic terminal symbols.

2.2.1 Terminal Strings

Both terminal strings and generic terminals imply a direct test of the input string. A terminal string is made up of explicitly displayed characters and is enclosed in apostrophes in the metalanguage. Thus 'A' in the metalanguage means an A in the input string, and 'XYZ' means an XYZ in the input string. To satisfy a terminal string, the input string must match it exactly. Thus, X Y Z in the input string is not an instance of the terminal string 'XYZ' since the input string contains blanks between the X , the Y and the Z .

2.2.2 Generic Terminal Symbols

The generic terminal symbols provide a simple way of specifying any one of a set of possible terminal strings. There are four types of generic terminal symbols:

- .D which stands for any decimal digit (0 to 9),
- .L which stands for any letter of the alphabet (A to Z),
- .A which stands for any alphanumeric (0 to 9 and A to Z),
- .C which stands for any character, including blank and all the special symbols.

2.2.3 Scanning the Input

Whenever the characters in the input string match the requested terminal string or generic terminal symbol, the characters are said to have been scanned. Once part of a source string has been scanned, any further testing will be done on the input string characters which immediately follow the scanned characters. For example, the syntactic equation

```
SAMPLE1 = .L ;
```

means that the input string at this point, if it is a letter, is an instance of a SAMPLE1. Thus if X is the next character in the input string, it is an instance of SAMPLE1, since X is a letter. Further checking of the input string proceeds with the character following the X.

2.2.4 Syntactic Unit

Another basic component of a syntactic expression is a reference to a syntactic unit. A reference to a syntactic unit in an expression means that the input string is to be examined to see whether or not it satisfies the definition of this second syntactic unit. Thus the syntactic equation

```
SAMPLE1A = SAMPLE1 ;
```

means that the input string, to be a SAMPLE1A, must consist of a SAMPLE1, ie. a letter.

2.3 Syntactic Operators

A syntactic expression can also contain metalanguage operators, which specify the relationship between components within the expression. The syntactic operators control the testing order for syntactic primaries. The operators also imply what is to be done if the sought syntactic primaries are not found.

2.3.1 Juxtaposition

The simplest metalinguistic operation is juxtaposition (which has no operator). Juxtaposition means one syntactic unit (or terminal) followed by another, with intervening blanks permitted. In PPL commentary may be used anywhere that blanks are permitted. Commentary is delimited by #: at the beginning of the comment and :# at the end. The compiler ignores commentary, and for the purposes of describing the syntax of PPL, commentary and blanks are considered synonymous. For the characters in the input string to satisfy a juxtaposition, they must satisfy the syntactic units or terminals in the order given. Furthermore, finding the first syntactic unit or terminal of the juxtaposition is a commitment to finding the second. Thus,

```
SAMPLE2 = .L .A  ;
```

means a letter followed by an alphanumeric, with any number of blanks or commentary. Anything satisfying this description is an instance of a SAMPLE2. If a letter is found when looking for SAMPLE2 and the next non-blank character is not an alphanumeric an error will be signalled. Since blanks and commentary in a juxtaposition are equivalent, the following are all examples of source code strings which satisfy the definition of SAMPLE2:

```
A1

B    Z
C #:THIS IS A SAMPLE2:# Y
```

2.3.2 Alternation

The alternation operator, | , is used to indicate that the juxtaposition which follows is a syntactic alternative to the previous one. An example of the use of this operator is shown below in the definition of a REGISTER in the PPL syntax:

```
REGISTER =   'R0'   |   'R1'   |   'R2'
           |   'R3'   |   'R4'   |   'R5'  ;
```

Thus, the syntactic unit REGISTER is satisfied by either R0 or R1 or R2 or R3 or R4 or R5 . When trying to satisfy a syntactic expression, the input string is searched for the first element of the first juxtaposition. If it is found, the additional elements of the first juxtaposition must be found, and an error is signalled if they are not. If the first element is not found, the compiler checks for the first element of the second juxtaposition etc. Thus,

```
SAMPLE3 = .L .A  |  '$' .D  ;
```

would be satisfied by any of the following:

```
A1

B    X

$2
```

while an error would be indicated for the following:

```
A¢

$X
```

since in the first failure case a letter was found (A) but it was not followed by an alphanumeric as required by the first alternative of SAMPLE3. In the second case a $ was found, as required by the first element of the second alternative of SAMPLE3, but it was not followed by the required digit.

2.3.3 Concatenation

The concatenation operator (&) indicates that no blanks or commentary should be skipped before looking for the syntactic element that follows it. For example,

```
SAMPLE4  =  .L  &.A  ;
```

will be satisfied only by a letter which is immediately followed by an alphanumeric with no intervening blanks. Furthermore, if the first letter was found and an alphanumeric does not follow an error will be indicated.

2.3.4 Parenthesis

Parentheses in the metalanguage are grouping operators, as in algebra. The entire sub-expression within the parentheses may be considered to be a single syntactic element; ie. if the input string satisfies the parenthesized sub-expression, it satisfies the syntactic element. Using parentheses and alternation, the equivalent of the syntactic unit of SAMPLE4 might be expressed as

```
SAMPLE5  =  .L &( .L | .D )  ;
```

since an alphanumeric .A is any letter .L or digit .D . Note, the & is distributive - in the above case it implies do not skip blanks before looking for either the second letter or the digit. Failure to find a concatenated letter or digit following the first letter means the syntactic sub-expression was not satisfied. Thus an error will be indicated since the parenthesized sub-expression &(.L | .D) is the second element of a juxtaposition.

2.3.5 Sequence Operator

The sequence operator ($) in the metalanguage means that the syntactic element which follows may be repeated zero or more times. Thus,

```
INTEGER  =  .D  $( &.D )  ;
```

means that an INTEGER is a decimal digit followed by a sequence of zero or more concatenated decimal digits. The parentheses around &.D are required because the scope of the sequence operator is the next syntactic primary, in this case a parenthesized syntactic expression. A slightly more complicated example of the sequence operator is

```
SIMPLE_IDENTIFIER  =  .L  $( &( .A | '_' ) )  ;
```

This means that a SIMPLE_IDENTIFIER in the PPL language can be a letter .L followed by zero or more concatenated alphanumeric characters or break characters (_). The sequence is satisfied even if there are no alphanumeric or break characters following the first letter, and thus a SIMPLE_IDENTIFIER may be a single letter.

2.3.6 Void Alternative

The syntactic element .E or .EMPTY stands for a void alternative, ie. a syntactic element which is always satisfied. It provides a way of indicating an option. Thus,

```
SAMPLE5  =  .L &( .A | .E )  ;
```

means that a single letter alone will satisfy SAMPLE5, since an instance of SAMPLE5 is a letter .L concatenated with an alphanumeric or with nothing.

2.3.7 Commentary

Finally, commentary may be used in the metalanguage. As in PPL, commentary is enclosed in #: and :# delimiters. Commentary in the definition of PPL will often be used to indicate restrictions, eg.

```
INTEGER = .D $( &.D ) #:value<65536:#  ;
```

means an integer may be a digit followed by a sequence of concatenated digits whose value is less than 65536 (the maximum representable value in the PDP11).

3. PPL

There are some general things that can be said about PPL before launching into the full definition of the language. First, there are no reserved words , or words which may not be used as identifiers. All non-identifiers, ie. operators, commands, or statement key-words, begin with a point (.). All PDP11 machine operations are directly expressible in the language. Commentary may appear anywhere that blanks may appear. Finally, all identifiers, except those of the general registers (which are known by default), must be declared before they are used within a program.

3.1 Identifiers and Declarations

3.1.1 Values and Cells

In PPL, the storage elements of the PDP11, the registers and core memory locations (words or bytes),are called cells. Every cell has a set of attributes associated with it, which in general is specified by the user. Cells may have one of three types : INTEGER, ADDRESS,or LOGICAL,which represent the type of value contained within the cell. Each cell also has a length associated with it, ie. it may be a BYTE(8 bits) or WORD(2 bytes) cell. WORD cells must begin on a word boundary. There are certain restrictions that exists between the set of attributes that a cell may have. For example, INTEGER types must be WORD if they are used in arithmetic. These restrictions arise because of the physical characteristics of the PDP11.

A cell is identified by its name, or IDENTIFIER. Most identifiers are programmer defined. However, the names, and attributes of the general registers are predefined. The general registers R0 to R5 have an undefined type and length.

```
REGISTER =    'R0'  |  'R1'  |  'R2'
            | 'R3'  |  'R4'  |  'R5'  ;
```

Programmer defined identifiers must satisfy the definition

```
SIMPLE-IDENTIFIER =  .L $( .A | '_');
```

Thus, an identifier must be one of the following

```
IDENTIFIER =  SIMPLE-IDENTIFIER | REGISTER ;
```

The allocation attribute for memory cells refers to both the location and method of referencing the cell. The allocation attribute may be RELATIVE (allocated together with the program code and accessed using the program counter, or PC register) or STACKED (allocated in the program stack, and accessed in the compiled code by means of the stack pointer, or SP register). A third type of allocation is that of the LITERAL identifiers, ie. identifiers whose constant values are known at compile time. Storage for LITERALs is allocated directly in the program code. PDP11 immediate addressing is used to access them .

A cell may be a member of an ordered group of cells, ie. an array. Array cells are accessed by means of the array name , synonymous with its first element, and a displacement (in bytes) from the first element. The array itself may be either RELATIVE or STACKED. When an array is STACKED, the displacement may only be expressed explicitly as a POSITIVE_VALUE. For RELATIVE arrays, the displacement may be either a POSITIVE_VALUE or an INDEX_EXPRESSION. INDEX_EXPRESSIONs have an optionally auto-decremented or auto-incremented general register, indicated by preceding, or following, respectively, the register identifier with an apostrophe. The amount of increment is equal to 1 for BYTE arrays

and 2 for WORD arrays. If neither auto-increment nor auto-decrement is indicated a fixed offset value, in the form of DEFINED-EXPRESSION, may be given in the case of INDEX_EXPRESSION displacements. In all cases, the displacement is in terms of BYTEs. Thus, if an array is made up of WORDs, the elements 1,2,3,...i must be accessed with the displacements 0,2,4,...2*(i-1). There exists an entirely different method of accessing individual cells in an array which will be discussed later.

The PDP11 also allows for indirect (or deferred) addressing of memory cells. This means that the cell specified does not contain the value to be used, but rather the address of a memory cell which contains the value. Indirect addressing using a memory cell is indicated by preceding the memory cell, which might be an array element, by the character @. The memory cell that contains the address must have the ADDRESS attribute and WORD length. Indirect addressing using a register cell is also permitted. In this case,the register containing the address of the desired memory cell may also be auto-decremented before its use in addressing the memory cell, or auto-incremented after its use. Auto-decrement and auto-increment are indicated, as in an INDEX-EXPRESSION, by preceding or following, respectively, the register indentifier by an apostrophe ('). The amount of the increment or decrement is 2 for WORD length cells, and 1 for BYTE length cells. A register used in indirect addressing without auto-increment or auto-decrement may have a displacement modifier following it in parentheses.

No assumption is made about the attributes of cells referred to indirectly,their attributes being deduced from the context in which the cell is used. Indirect addressing of a cell via a register may be single level, as above, where the address of the cell is in the register, or double level where the address of the cell is in a second cell whose address is in a register. Double level indirect addressing is indicated by @@ preceding the register, which, as in single level indirect addressing, may be auto-decremented or incremented or have a displacement in parentheses. Note that auto-decrementing or -incrementing the displacement applies to the contents of the register, not to the indirect address contained in the second level memory cell.

Indirect addressing provides a second method of referencing array variables. The base address of an array, or the address of any element of the array, may be put into a general register or a memory cell, and then used indirectly to refer to items in the array by the means of an appropriate displacement added to the cell. In addition, if a general register is used for indirect addressing of array elements, it may be auto-incremented or -decremented, thus providing a convenient means of stepping through a STACKED array. Random accessing of STACKED array elements must be done indirectly, and random accessing with auto-increment or -decrement must also be done indirectly.

```
CELL  =   DIRECT_CELL
      |   '@'  #:INDIRECT ADDRESSING:#
               (  MEMORY_CELL  #:ATTRIBUTE=ADDRESS,WORD:#
               |  REGISTER_INDIRECT
               |  '@'  #:DOUBLE LEVEL INDIRECT:#
                       REGISTER_INDIRECT
                       #:FIRST LEVEL INDIRECT (VIA REGISTER) GIVES:#
                       #:    THE ADDRESS OF A MEMORY WHICH HAS THE :#
                       #:    ADDRESS OF THE CELL                   :#
               )   ;


DIRECT_CELL  =  MEMORY_CELL  |  REGISTER  ;
```

```
REGISTER_INDIRECT
     =  '''' REGISTER #:OPTIONAL AUTO-DECREMENT:#
      | REGISTER
        ( ''''          #:OPTIONAL AUTO-INCREMENT:#
        | '(' SIGNED_EXPRESSION ')' #:BYTE OFFSET FROM REGISTER VALUE:#
        | .E )
        #: ONLY ONE OPTION ALLOWED:#  ;


MEMORY_CELL  =  IDENTIFIER  #:ATTRIBUTES=STACKED,ARRAY:#
                '('  POSITIVE_VALUE  ')'  #:BYTE DISPLACEMENT:#
             |  IDENTIFIER  #:ATTRIBUTES=RELATIVE,ARRAY:#
                '('  POSITIVE_VALUE  |  INDEX_EXPRESSION  ')'
                     #:BYTE DISPLACEMENT:#
             |  IDENTIFIER  #:ATTRIBUTE=REGISTER:#     ;


INDEX_EXPRESSION  =  REGISTER
                       ( ( '+' | '-' )  DEFINED_EXPRESSION #:OFFSET:#
                       | .E )  ;
```

3.1.2 Attributes

Every cell, or array of cells, has several attributes associated with it. These attributes are determined either from the definition of the cell identifier, or from the context,when the cell is of a type which is not declared (ie. absolutely or indirectly addressed). A simple user defined identifier may have only one attribute from each of the classes: type, allocation, length, and scope of definition. Within each class of attributes, there is a default in the case of non-specification. In addition, some attributes imply others, for example, ADDRESS implies WORD.

```
ATTRIBUTES
        = #:TYPE.  DEFAULT=INTEGER:#
             '.INTEGER'  |  '.ADDRESS'  |  '.LOGICAL'
          |  '.STRING'  #: EQUIVALENT TO LOGICAL, BYTE ARRAY :#
                '(' POSITIVE_VALUE ')' #: STRING LENGTH :#
        |  #:ALLOCATION, DEFAULT=STACK:#
             '.RELATIVE'  |  '.STACKED'
           |  '.LITERAL' #:INITIAL VALUE REQUIRED:#
                         #:MAY NOT BE EXTERNAL   :#
                         #:MAY NOT BE AN ARRAY   :#
           |  #:LENGTH, DEFAULT=WORD:#
                '.BYTE'  |  '.WORD'
           |  #:SCOPE, DEFAULT=INTERNAL:#
                '.EXTERNAL'  |  '.INTERNAL'
                #:EXTERNAL ONLY WITH .RELATIVE:# ;
```

3.1.2.1 Type

The type attribute refers to the way in which an identifier may be used. There are three type attributes, INTEGER, ADDRESS, and LOGICAL, and one subtype, STRING. INTEGERS are used in arithmetic, and are treated as two's complement binary values. INTEGER is the default type. ADDRESS values are used in addressing and address arithmetic, and are treated as 16 bit binary integers in comparisons. Care should be taken when doing arithmetic on large address values,as all addition and subtraction is done assuming

two's complement binary operands. The type ADDRESS may be applied only to WORD length cells, since all PDP11 addresses must be 16 bits long. LOGICAL cells may be used for the storing of flag bits. Individual bits within a LOGICAL cell may be set or cleared, and other logical operations, such as complementing and logical shifting may be performed with LOGICAL values. Logical shifts are permitted only if the particular PDP11 for which the program is being compiled has the high speed arithmetic unit, KE11-A. The STRING type is actually a subtype of LOGICAL, and is provided to allow initialization of memory with character strings. A STRING is equivalent to a LOGICAL, BYTE array. The .STRING keyword is followed by a parenthesized positive value which is the length of the string (in bytes, one byte per character).

3.1.2.2 Allocation

The allocation attribute refers to the method by which storage for a cell is allocated, and also to the method by which it is accessed in the object code. RELATIVE refers to cells whose storage is permanently allocated with the program at compile time. RELATIVE cells are accessed using PDP11 relative addressing based on the program counter. They may be given an initial value at compile time. STACKED cells have storage allocated for them dynamically in the stack, and are accessed using relative addressing based on the stack pointer. The compiler adjusts the stack pointer when STACKED cells are declared, generates the appropriate relative displacement when the cell is accessed, and removes the STACKED cells from the stack when the block in which they are declared is terminated. STACKED values may not be initialized at compile time. The third type of allocation is LITERAL, which is used to predefine values which are known at compile time. LITERAL values may be used as constants, and space is allocated for them in the object code. They are accessed using PDP11 immediate addressing. An initial value is required for all LITERALs. The default allocation type is STACKED.

3.1.2.3 Length

There are two possible lengths for values in the PDP11, WORD for 16 bit items, and BYTE for 8 bit items. The default length is WORD.

3.1.2.4 Scope

The scope of a variable refers to the extent to which it is known, and therefore accessible. INTERNAL cells may be referenced only from inside the block in which they are declared, while EXTERNAL cells may be referenced from any block. INTERNAL is the default scope. Only RELATIVE non-LITERAL cells may be EXTERNAL, and they may not be initialised. EXTERNAL values are provided to allow communication of values between separately compiled routines. References to EXTERNAL cells are resolved in a symbolic environment prior to loading of the object program. The final memory location of RELATIVE cells is not determined until the link/load time.

3.1.3 Cell Declarations

```
IDENTIFIER_DECLARATION
     = ATTRIBUTES  $( ATTRIBUTES )  #:ATLEAST ONE REQUIRED:#
       IDENTIFIER_DEFINITION $( ',' IDENTIFIER_DEFINITION )  ';' ;
```

```
IDENTIFIER_DEFINITION
     = SIMPLE_IDENTIFIER  ( '(' POSITIVE_VALUE ')'
                             #:FOR ARRAYS: LENGTH IN BYTES:#
                          |  .E #:SIMPLE CELL:#  )
    ( '='
      (  FILL_VALUE
      |  '.SYNONYM'  '(' DEFINED_MEMORY_CELL
                          ( #:FOR .BYTE TO .WORD SYNONYMING ONLY:#
                            '+'  DEFINED_EXPRESSION #:VALUE=0 OR 1:#
                          | .E )
                       ')'
      |  '.LOCATION'  '(' POSITIVE_VALUE ')' #:LOCATION HAS ALLOC ABS:#
    | .E  )  ;


DEFINED_MEMORY_CELL
     =  IDENTIFIER #:PREVIOUSLY DECLARED:#
        (  '(' POSITIVE_VALUE ')'  #:FOR ARRAYS ONLY- AN ARRAY ITEM:#
        |  .E #:SIMPLE CELL OR FIRST ITEM OF ARRAY:#  )   ;


IDENTIFIER  =  SIMPLE_IDENTIFIER  |  REGISTER  ;


REGISTER = 'R0' | 'R1' | 'R2'
         | 'R3' | 'R4' | 'R5'
           #:ATTRIBUTES: TYPE=ANY, LENGTH=EITHER :#   ;
```

An IDENTIFIER-DECLARATION is used to associate attributes with a user selected cell name. Every identifier declaration must begin with an attribute keyword. Further attributes may also be specified, subject to the restrictions previously mentioned. The set of attributes is followed by one or more identifier definitions, seperated by commas. An identifier definition specifies the identifier name, perhaps an additional attribute (eg. array size) and the initial value, if any. A declaration is terminated by a semicolon (;). An identifier definition must consist of at least a simple identifier. If the identifier is followed by a positive value in parentheses, the identifier is an array. The positive value specifies the array length in bytes. LITERALS may not be arrays.

An identifier in an identifier definition may be followed by one of three mutually exclusive conditions, all preceded by an equals sign (=):

1) a fill value, used as an initial value for the cell

2) .SYNONYM followed by an identifier in parentheses with which it is to share memory

3) .LOCATION followed by a positive value in parentheses which specifies the absolute address of this cell

.LOCATION is provided to tell the compiler that this cell has a pre-defined, absolute location. Device registers, trap vectors etc. use this facility. .LOCATION or the specification of a fill value is permitted only with RELATIVE cells.

.SYNONYM allows the definition of multiple names and attributes for a single cell or for an array. Synonymed cells must have the same allocation attribute. In parentheses following .SYNONYM is the name of a previously declared identifier with which this newly declared identifier is to share storage. The previously declared cell may be an element of an array or a simple cell. In either case, the identifier being defined must be containable within the storage allocated to the previously defined identifier. Thus, a simple BYTE memory cell may be set synonymous with a WORD cell, in which case it is considered to be the HIGH ORDER byte of the word. A simple BYTE cell may also be set synonymous with a "WORD memory cell +1", in which case it is a synonym for the LOW ORDER byte of the word. Setting a WORD cell synonymous with a BYTE cell is not permitted except when the BYTE cell is an element of an array which ends on a word boundary (even address). An array may be set synonymous within another array, again with the proviso that the synonymed array is completely containable within the previously declared array. Only the principal identifier of a cell or array not a synonym can be initialized.

Declaration Examples

```
.INTEGER I,J(10);
```

I is an INTERNAL STACKED WORD (by default) INTEGER cell. J is an INTERNAL STACKED WORD array which is 10 bytes long (5 elements).

```
.LOGICAL .BYTE FLAGS(10), IFLAG=.SYNONYM (FLAGS(0));
```

FLAGS is a LOGICAL BYTE array of length 10. IFLAG is a LOGICAL BYTE cell which is another name for the cell FLAGS(0). It is STACKED and INTERNAL by default.

```
.ADDRESS .RELATIVE .EXTERNAL HEADPTR;
```

HEADPTR is WORD in length by default. Since it is EXTERNAL, other separately compiled programs may reference it provided that they contain a similar declaration.

```
.LOGICAL .BYTE .RELATIVE TTYBUF = .LOCATION(.O(777562));
```

TTYBUF is defined to be the RELATIVE LOGICAL BYTE at octal location 777562. TTYBUF has the default attribute INTERNAL.

```
.BYTE .INTEGER K; .LOGICAL LOWBYTE = .SYNONYM(K+1);
```

K is a STACKED WORD INTEGER cell, while LOWBYTE is a STACKED BYTE LOGICAL cell that occupies the same storage as the low order byte of K.

```
FILL_VALUE  =  CHARACTER_STRING  #:FOR .STRING OR LOGICAL BYTE CELLS:#
            |  '.ADDRESS'  '(' DEFINED_MEMORY_CELL ')' #:RELATIVE ONLY:#
            |  DEFINED_EXPRESSION
            |  IDENTIFIER #:ATTRIBUTES=LOGICAL,LITERAL,NOT ARRAY:#
            |  ( POSITIVE_VALUE #:ITERATION COUNT:# | .E )
               '('  FILL_VALUE  $( ',' FILL_VALUE )  ')'
                  #:VALID ONLY FOR ARRAYS:#    ;


CHARACTER_STRING
     = (   '.E"'  #: EBCDIC :#
       |   '"'  #: ASCII :#  )
        $( &(  '""'  #: "" IN SOURC -> " IN STRING :#
           |   .C  #:ANY CHARACTER OTHER THAN " :#  )  )
       '"'  #: CLOSING QUOTATION MARK:# ;
```

A fill value is used to specify the initial value of a memory cell or array. It may be a character string if the identifier has the subtype .STRING. There are two types of character strings, EBCDIC, and ASCII-8, the difference being the internal codes produced for each character. An EBCDIC character string is preceded by the flag .E. In both cases, the character string is enclosed in quotation marks ("). To represent a quotation mark within a character string, two quotation marks ("") must be coded without an intervening character. The pair is required in order to distinguish it from the delimiting quotation mark. The necessity for paired quotation marks within a character string gives rise to the metalanguage terminal definition '""' for a quotation mark in a character string. A character string must be coded entirely on a single card.

A fill value may specify the ADDRESS of another previously defined simple identifier. In this case the identifier which is being initialized must be of the type ADDRESS, and the previously declared identifier must be RELATIVE.

A fill value for LOGICAL values may be a previously declared LOGICAL LITERAL, or it may be expressed as a defined expression. Thus LOGICAL INTEGERs, as well as expressions, may be used to initialize a LOGICAL value. Defined expressions are most useful for initializing arithmetic and ADDRESS type cells.

Syntactically a fill value may in fact be a sequence of fill values, separated by commas, and enclosed in parentheses. A positive value may precede the left parenthesis and is treated as a repetition factor. This form of fill value is restricted to arrays.

A defined expression is a sequence of defined values, either INTEGER or LITERAL, seperated by arithmetic operators. The expression is evaluated at compile time in a strictly left to right fashion with no precedence, to 32 bits of precision. An error is indicated if significance is lost in initializing a 16 bit WORD or an 8 bit BYTE cell. There are two types of INTEGERs: decimal integers, and logical integers (binary, octal, and hexadecimal). All INTEGERs are assumed positive.

Examples:

```
.LITERAL ZERO=0,ONE=1,TWO=2;
```

The values 0, 1, and 2 are associated with the identifiers ZERO, ONE, and TWO respectively. The default attributes are INTEGER WORD INTERNAL.

```
.LITERAL SIX=TWO*3, SEVEN=1+2*TWO+ONE;
```

The identifiers SIX and SEVEN are associated with the values 6 (=TWO*3), and 7 (=1+2*TWO+ONE). Evaluation of the expressions proceeds strictly left to right.

```
.STRING(25) .RELATIVE MESSAGE1="THIS IS AN ASCII MESSAGE.";
```

MESSAGE1 is declared to be a RELATIVE LOGICAL BYTE array of length 25, with the default attribute INTERNAL. The attribute .RELATIVE is required if the string is to be initialized.

```
.STRING(26) .RELATIVE MESSAGE2=.E"THIS IS AN EBCDIC MESSAGE.";
```

MESSAGE2 is declared as a RELATIVE LOGICAL BYTE array of length 26, with the default attribute INTERNAL. This array is initialized with EBCDIC character codes, rather than ASCII character codes, as indicated by the .E immediately preceding the open quotation mark.

```
.STACKED .STRING(80) BUFFER;
```

BUFFER is a STACKED LOGICAL BYTE array of length 80. Allocation of the string BUFFER to the stack might be used to conserve storage by a little used I/O routine.

```
.RELATIVE FANCY(20)=(0, 2(1,2,2(3)),4) ;
```

FANCY is an array of 10 INTERNAL WORD elements by default (=20 bytes). The complicated FILL-VALUE expression results in the following initial values:

```
FANCY (0)  = 0
FANCY (2)  = 1
FANCY (4)  = 2
FANCY (6)  = 3
FANCY (8)  = 3
FANCY (10) = 1
FANCY (12) = 2
FANCY (14) = 3
FANCY (16) = 3
FANCY (18) = 4
```

3.2 Assignment Statements

A set of relationships between values is defined by the monadic and dyadic functions, or operations, which the PDP11 processor is able to evaluate or perform. The relationships are defined by mappings between values or pairs of values known as destination operands, and values known as source operands. These mappings are defined in the PDP11 handbook and will not be defined here. One or more of these operations may be expressed in an assignment statement in PPL.

```
ASSIGNMENT  =  CELL  $( ',' CELL )  #:MULTIPLE ASSIGNMENT:#
                     #:NO LITERALS ON LEFT:#
               '='     (  ARITHMETIC_EXPRESSION
                       |  ADDRESS_EXPRESSION
                       |  LOGICAL_EXPRESSION  )    ';'
            #: CELL ON LEFT MUST AGREE IN TYPE  :#
            #: AND LENGTH WITH EXPRESSION ON    :#
            #: RIGHT OF ASSIGNMENT STATEMENT    :#    ;
```

There are 3 types of ASSIGNMENT statements - arithmetic, address, and logical, corresponding to the three type attributes .INTEGER, .ADDRESS, and .LOGICAL. The existence of different types of assignment statements allows the compiler to do more error checking, and therefore helps the user to write clear, logical code. For example, multiplication of ADDRESS values is nonsensical, as is the use of arithmetic operations on LOGICAL data. These two types of operations are too often mixed together in programs, but they should be classed as tricks which more often tend to confuse rather than help. An assignment statement is made up of a sequence of operands separated by operators which indicate the operations to be performed. The type of an assignment statement is determined by the types of the operands and operators involved in the statement. The length and type attributes of all values used in an assignment statement must agree with each other, and with the type of the operators.

The type and length of an assignment statement are determined by the first cell or operator with a definite type or length, ie. first in a left to right scan of the statement. There are cells and operators which do not have definite type or length attributes. For instance, a cell referred to indirectly may have any type or length, and the addition operator (+), is valid for both ADDRESS values and INTEGER values. It is possible that an assignment statement will have no operands or operators that are specific. In this case the default is INTEGER WORD, and code will be generated accordingly.

Assignment statements like defined expressions, are interpreted in a stricly left to right manner. The receiving cell, or destination (ie. the first cell on the left of the equals sign), is used as the accumulator. All operations are performed on the current contents of the receiving cell. The presence of more than one cell on the left of the equals sign indicates a multiple assignment. Following evaluation of the expression using the first cell on the left, the new value will be copied into the additional destination cells.

3.2.1 Arithmetic Assignments

```
ARITHMETIC_EXPRESSION  =  ( '-'  |  .E )
                             #: UNARY MINUS FOR FIRST ITEM ONLY :#
                          ARITHMETIC_VALUE
                          $(  ARITHMETIC_MONADIC_OPERATOR
                           |  ARITHMETIC_DYADIC_OPERATOR
                              ( ARITHMETIC_VALUE
                              | '.CARRY'  #:FOLLOWING + | - ONLY :# )
                           )  ;

ARITHMETIC_VALUE
        =  CELL  #:ATTRIBUTE=INTEGER:#
        |  INTEGER   ;


POSITIVE_VALUE  =  SIGNED_EXPRESSION   #: VALUE>=0:#   ;


SIGNED_EXPRESSION  =  ( '+'  |  '-'  |  .E  )  #:OPTIONAL SIGN:#
                      DEFINED_EXPRESSION  ;
```

```
DEFINED_EXPRESSION  =  DEFINED_VALUE
                       $(( '+' | '-' | '*' | '/' )
                         DEFINED_VALUE )  ;


DEFINED_VALUE  =  INTEGER
               |  IDENTIFIER #:ATTRIBUTES=LITERAL,INTEGER OR ADDRESS:#

INTEGER  =  DECIMAL_INTEGER  |  LOGICAL_INTEGER  ;


DECIMAL_INTEGER  =  .D  $( &.D )  ;
```

In an arithmetic expression the operands are interpreted and treated as INTEGER values. An arithmetic expression may start with a unary minus, in which case the receiving cell is loaded with the negative of the first arithmetic value. Arithmetic values are cells or LITERAL identifiers with the INTEGER type (decimal, binary, octal, or hexadecimal). Following the first arithmetic value may be a series of operators, some of which require an operand. The monadic arithmetic operators need no operand. They operate in a post-fix fashion on the current contents of the receiving cell.

```
ARITHMETIC-MONADIC-OPERATOR
     =  (  '.ROL'   |   '.ROR'   |   '.INC'   |   '.DEC'   | '.NEG'  )
          &( 'B' #:BYTE:# | .E #:WORD:# )
     |  ( '.ASL' | '.ASR' ) &('B' #:BYTE:# | .E #:WORD:# )
          ( '(' POSITIVE_VALUE ')' #:AMOUNT OF SHIFT<16:#
          | .E #:SINGLE BIT SHIFT:# )
     |  '.AHS'  #:ARITHMETIC HIGH SPEED SHIFT:#
          &( 'B' #:BYTE:# | .E #:WORD:# )
          '(' ( ARITHMETIC_VALUE
              | SIGNED_EXPRESSION  )  ')'  #:SHIFT AMOUNT:#   ;
```

The Arithmetic Monadic Operators correspond to all those PDP11 operations requiring only a destination operand, applicable to INTEGER values. There is no commonly used special symbol that can be selected to indicate these operations, so the appropriate PDP11 mnemonic operation code, preceded by a point (.), is used as a keyword. The optional concatenated suffix B is used to indicate that the operation is BYTE rather than WORD oriented. This also corresponds to the PDP11 mnemonic operation code naming convention. The B suffix is required if the operands are BYTE in length. The arithmetic shifts have been extended artificially to allow multiple bit shifting, since the PDP11 only performs single bit shifting. Multiple bit shifting is actually accomplished by making the appropriate number of single bit shifts. The number of shifts is indicated by a parenthesized positive value following the operation keyword. This number must be less than 16.

There is a monadic operation which is not part of the PDP11 basic instruction set. This is the arithmetic high speed shift (.AHS), which is actually performed by the EAE (Extended Arithmetic Execution unit KE11-A logic option). This operation is defined as monadic since it acts more like a function than an operation, the actual processing being handled by the EAE. In parentheses following the keyword .AHS, which should be concatenated with B in the case of BYTE length operands, is a parenthesized value specifying the amount of the shift - a negative value for a left shift, and a positive value for a right shift.

```
ARITHMETIC_DYADIC_OPERATOR
        =  '+'  |  '-'  #:FOR BYTES: VALUES OF -1,0,1 ONLY:#
        |  '*'  |  '/'  #:VALID ONLY WITH EXTENDED AU:#   ;
```

The dyadic arithmetic operators require a second operand, corresponding to the PDP11 source operand. Addition (+) and subtraction (-) have their usual meanings, and result in the addition of the second operand to the receiving cell. If the second operand happens to be .CARRY, the appropriate action of adding the carry bit will be performed. In addition, the use of addition and subtraction with two cells requires that both be WORD cells since it is impossible to add two BYTE cells on the PDP11, or to add a BYTE cell to a WORD cell. A +1 or -1, or a LITERAL or an INTEGER with a value of +1 or -1 will compile to an .INC or .DEC, whichever is appropriate. This may be specified for BYTE as well as WORD cells. The dyadic operators, multiply (*) and divide (/), may only be used with the EAE unit. Examples:

```
R0=R1+2;
```

Register R0 is loaded with the value contained in register R1, and then 2 as an immediate operand is added to R0.

```
.LITERAL INT=15;

R2=-INT;
```

Register R2 is loaded with the value of INT (ie. 15) and then negated.

```
.INTEGER  I;
.RELATIVE  .INTEGER J(10)=5(0);
I,J(0)=J(0)+10 .ASL .INC .NEG
```

I is loaded with the value of the array element J(0) (initially zero) to which 10 is added. The contents of I are shifted left one (ie. multiplied by 2), incremented, and then negated, resulting in the value -21. Finally, the value I is copied into J(0).

```
.LITERAL  INDEX=4;
.INTEGER  .RELATIVE K(10);
R1,K(R1+2)=INDEX;
```

Register R1 is loaded with the value of INDEX. Then, K(6), the fourth element of the array K, is set to 4. K(6) is the indicated element because R1 now contains 4 plus an offset of 2 ie. K(R1+2)=K(4+2)=K(6).

3.2.2 Address Assignment Statements

```
ADDRESS_EXPRESSION
      =  ADDRESS_VALUE
         $(  '.ASL'  |  '.ASR'
          |  ( '+' | '-' )  ADDRESS_VALUE  )  ;


ADDRESS_VALUE
      =  CELL  #:ATTRIBUTE=ADDRESS:#
      |  INTEGER
      |  '.ADDRESS'  '(' MEMORY_CELL  ')'  ;
```

Address assignment statements, are assignment statements with ADDRESS cells on the left, and an address expression on the right of the assignment operator (=). Address expressions are a subset of the arithmetic expressions, and consist of an ADDRESS value followed by a sequence of monadic operators (the arithmetic shifts) or dyadic operators (+ and -) which must be followed by a second ADDRESS value. Since all ADDRESS values must be WORDs, only shifts on full words are permitted.

An ADDRESS value may be either a cell with the ADDRESS attribute, an identifier which is an ADDRESS LITERAL, an INTEGER, or the address of a memory cell as indicated by the keyword .ADDRESS followed by a cell identifier in parentheses. The compiler will generate the appropriate code to produce the absolute address of the cell at execution time. The cell may be allocated in the stack.

Examples:

```
.ADDRESS  SARRAY(20),INDEX;
INDEX=3;
R0=INDEX-1 .ASL #:GIVES BYTE DISPLACEMENT#:;
R0=R0+.ADDRESS(SARRAY);
@R0=R0;
```

In this example, a random element of the STACKED, ADDRESS array SARRAY is set with its own address, where the random element happens to be the third one. The correct byte displacement of 4 is generated in R0 by INDEX-1 .ASL. Then the starting address of the STACKED array, which is provided by .ADDRESS, is added to the displacement. The fourth assignment statement actually sets the specified address to itself.

```
.INTEGER  .LITERAL  INDEX=5;
.ADDRESS  .RELATIVE  RARRAY(16);
.ADDRESS  .LITERAL  BASE=.ADDRESS(RARRAY);
.ADDRESS  .RELATIVE  DISP=INDEX-1*2;
RARRAY(0)=BASE+DISP;
RARRAY(0)=@RARRAY(0);
```

RARRAY(0), after execution of this code, will contain the address of the fifth element of RARRAY, ie. the one with a displacement of 8, as specified by the value of DISP.

3.2.3 Logical Assignment Statements

```
LOGICAL_EXPRESSION
       =  ( '¬'  |  .E )  #:UNARY OP FOR FIRST ITEM:#
          LOGICAL_VALUE
          $(  LOGICAL_MONADIC_OPERATOR
           |  LOGICAL_DYADIC_OPERATOR   LOGICAL_VALUE  )  ;


LOGICAL_VALUE    =  CELL  #:ATTRIBUTE=LOGICAL:#
                  |  LOGICAL_INTEGER
                  |  CHARACTER_STRING  #:LENGTH=1. AS BYTE VALUE ONLY:#;


LOGICAL_MONADIC_OPERATOR
       =   '.COM'  #:COMPLEMENT:#  &( 'B' #:BYTE:# | .E #:WORD:# )
       |   '.SWAB'  #:SWAP BYTES; ARGUMENT MUST BE WORD:#
       |   '.LHS' #:LOGICAL HIGH SPEED SHIFT, WITH HIGH SPEED AU ONLY:#
             &( 'B' #:BYTE:# | .E #:WORD:# )
             '(' ( ARITHMETIC_VALUE
                 | SIGNED_EXPRESSION )  ')'  #:AMOUNT OF SHIFT:#
        ;


LOGICAL_DYADIC_OPERATOR
       =  ( '.BIS' | '.BIC' )  &( 'B' #:BYTE:# | .E #:WORD:#)
          #:OPERAND THAT FOLLOWS USED AS MASK:#  ;


LOGICAL_INTEGER
       =  '.X('  &( .D  | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' )
             $( &( .D  | 'A' | 'B' | 'C' | 'D' | 'E' | 'F' )
          &')'
        |  '.O('  & .D  #:VALUE<8:#
             $( & .D  #:VALUE<8:#  )
          &')'
        |  '.B('  &( '0' | '1' )  $( &( '0' | '1' ) )  &')'   ;
```

Logical assignment statements consist of LOGICAL cells on the left of the assignment operator and a logical expression on the right. A logical expression consists of an optionally complemented LOGICAL value followed by a sequence of logical monadic operators, or logical dyadic operators and operands which are also LOGICAL values.

LOGICAL values include any cell with the attribute LOGICAL, LOGICAL LITERALs, INTEGERs, and character strings of length 1 which may only be used as BYTE values. The logical monadic operators include complement (.COM) which can be used on BYTE values by concatenating a B to it, swap bytes (.SWAB) which may be used only on WORD values, and the logical high speed shifts which are valid only with the arithmetic unit. These logical shifts may be WORD or BYTE oriented, as is the case for the arithmetic high speed shifts. The amount and direction of the shift is specified at execution time by the arithmetic cell or signed expression that follows the operator keyword in parentheses. In logical shifts, zeros are used to fill vacated bits.

There are only two logical dyadic operators .SET and .CLR, which correspond to bit setting and bit clearing respectively. The logical value that follows is used as a mask. Both the setting and clearing operators can be used with BYTE or WORD values.

Examples of Logical Assignment statements;

```
.LOGICAL  .LITERAL  .BYTE  A=.X(8C),B=.X(40),
                           D=.X(20),C=.X(10);
.LOGICAL  .BYTE  FLAG;
FLAG=FLAG          .X(FO);
FLAG=FLAG  .BICB C;
```

The first logical assignment statement sets the high order four bits of FLAG, leaving the low order four bits undisturbed. The second logical assignment statement clears the fourth bit from the left of FLAG.

```
.LOGICAL ABC;
ABC=.X(C55C).COM  .SWAB;
```

ABC is left with the value .X(C55C)=.B(1010010101011010) as a result of complementing and swapping of .X(C55C).

3.3 Control Facilities

The PPL language provides several methods of controlling the execution sequence of the resultant code. The main objective of PPL program control statements is to decrease the need for programmer generated labels and to increase comprehensibility.

3.4 Statements

3.4.1 Null Statement

```
NULL  =  ';'  #:VOID STATEMENT:#  ;
```

The NULL statement is the simplest statement of all - it does nothing. The NULL statement is indicated simply by a semicolon (;), which is the normal statement terminator. It is preset in the language simply as a space holder. Its use and utility will be demonstrated in conjunction with other control statements.

3.4.2 If Statement

```
IF  =  '.IF'  BOOLEAN_EXPRESSION  '.THEN'  STATEMENT
                                  ( '.ELSE'  STATEMENT  |  .E  )  ;
```

The IF statement provides for the conditional execution of the statement in the .THEN clause, ie. the statement following .THEN. The .THEN clause will be executed if the boolean expression has the value true. The .ELSE clause is optional. When present it is executed if the boolean expression has the value false. In general, following the execution of either the .THEN, or the .ELSE clause (if one is present) control normally passes sequentially to the next statement.

IF statements may be nested, with an IF statement appearing in either the .THEN or .ELSE clause. When IF statements are nested, .ELSE clauses are paired with the immediately preceding, unmatched .THEN clause Thus, in : .IF B1 .THEN .IF B2 .THEN S1; .ELSE S2; statement S1 will be executed if B2 is true while S2 will be executed if B2 is false, both providing that B1 is true. The .THEN clause: .THEN S1 and the .ELSE clause: .ELSE S2 both belong to the same IF statement. As a further example, consider .IF B1 .THEN .IF B2 .THEN S1; .ELSE S2; .ELSE S3; S1 will be executed if B1 and B2 are both true, S2 will be executed if B1 is true and B2 is false, and S3 will be executed if B1 is false since the .ELSE clause: .ELSE S3; pairs up with the first .THEN clause.

The NULL statement may be used to force the proper pairing of .THEN and .ELSE clauses. Consider

```
.IF B1
.THEN .IF B2 .THEN S1;
             .ELSE;
.ELSE S2;
```

in which the NULL statement is used in the first .ELSE clause. This forces the second .ELSE clause: .ELSE S2; to be associated with the first .THEN clause and the value of B1.

```
BOOLEAN_EXPRESSION
      =  BOOLEAN_AND
         #:CONDITIONS SHOULD APPEAR FIRST IN AN EXPRESSION:#
       $( '|' BOOLEAN_AND ) ;



BOOLEAN_AND
     =  BOOLEAN_PRIMARY  $( '&' BOOLEAN_PRIMARY )   ;


BOOLEAN_PRIMARY
     =  CONDITION #:CURRENT CONDITION CODE STATI:#
     |  '¬'  BOOLEAN_PRIMARY
     |  '('  BOOLEAN_EXPRESSION  ')'
     |  RELATION #:RELATION BETWEEN TWO VALUES:#   ;



COMPARISON_VALUE
     =  CELL #:ATTRIBUTE=INTEGER OR ADDRESS:#
     |  DEFINED_EXPRESSION   ;


CONDITION
     =  '.EQ'  |  '.NE'
     |  '.GE'  |  '.LT'  |  '.GT'  |  '.LE'
     |  '.PL'  |  '.MI'
     |  '.HI'  |  '.HIS'  |  '.LO'  |  '.LOS'
     |  '.VC'  |  '.VS'
     |  '.CC'  |  '.CS'   ;
```

A boolean expression consists of a boolean AND followed by a sequence of boolean ANDs seperated by | which is the operator for OR. A boolean expression is true if any one of the boolean ANDs is true, and false if and only if all of the boolean ANDs are false.

A boolean AND consists of a boolean primary followed by a sequence of boolean primaries seperated by &, the operator for AND. A boolean AND is true if and only if all of the boolean primaries are true, and false otherwise.

A boolean primary may be a condition, which refers to the current state of the condition code bits in the PDP11 Central Processor Status Register. The condition codes are used in the simple and signed conditional branch instructions of the PDP11. A condition is true, and thus the boolean primary it represents is true, when the corresponding PDP11 conditional branch instruction would result in a branch. The conditions are provided to allow selection of the .THEN or the .ELSE clause dependent upon the results of the preceding instruction (at execution time) that set the conditions. A condition may only appear as the first boolean primary of the first boolean AND in a boolean expression, since subsequent boolean primaries will invariably change the state of the condition code bits.

The other types of boolean primaries include the complement of a boolean primary, as indicated by a prefix (¬) before a parenthsized boolean expression. Boolean expressions are evaluated with the normal precedence of the boolean operators (¬ taking precedence over & which in turn has precedence over | , and not left to right).

```
RELATION
     =  COMPARISON_VALUE #:EXCLUDES LOGICAL:#
        ( '=' | '¬=' | '>' | '<' | '>=' | '<=' | '¬<' | '¬>' )
        COMPARISON_VALUE #:ANY BUT LOGICAL:#
     |  LOGICAL_VALUE
        (  '=' |  '¬='  )
        |  '.BIT'  &( 'B' #:BYTE:# | .E #:WORD:# )
        LOGICAL_VALUE
              #:VALUES MUST AGREE IN TYPE AND LENGTH:#  ;
```

The relations provide a way of comparing two values of like type and length, where the values may be arithmetic, ADDRESS, LOGICAL, or defined expressions of the appropriate type. A relation is true if the indicated condition is met. For arithmetic and ADDRESS values all the normal arithmetic relations are permitted. However they are interpreted differently depending on the apparent types of their values. A relation between arithmetic values is treated as a relation between binary two's complement values (either BYTEs or WORDs), while a relation between ADDRESS values is treated as a relation between unsigned, 16 bit integers. Two values of type LOGICAL (or a LOGICAL value and a defined expression) may be tested for equality or inequality. The PDP11 bit test comparison may be used, as indicated by the .BIT relational keyword between two LOGICAL values. In a bit test, which may be for BYTE or WORD values, the second logical value is used as a mask, indicating which bits are to be tested. A bit test is true if any of the indicated bits are set (ie. have binary value 1), and is false only when all of the indicated bits are clear (ie. binary value 0). It can happen that no specific type of value is indicated, as in the case of the assignment statement where both values refer to indirectly addressed cells. When no specific type is indicated, the comparison will be made under the assumption that both values are of INTEGER type.

Examples of Boolean Expressions:

```
.INTEGER A,B,C;
.
.
.
.IF A=B | C<=R4
```

The boolean expression is true if the value of cell A is equal to the value of cell B, or if the value of cell C is greater than or equal to the value in register R4.

```
.INTEGER A;
.LOGICAL B;
.ADDRESS ADDR;
    .
    .
    .
.IF A=1 & (B .BIT .X(FF) | @ADDR<R0)
```

The boolean expression is true only if the value of cell A is equal to 1 and either of the two relations contained within the parentheses is true. The first relation in the parenthesized subexpression, a bit test, is true if any of the low order eight bits of the WORD cell B are set. The second relation in the parenthesized subexpression is true if the value accessed indirectly via ADDR is less than R0. The comparison is made assuming that both values are binary two's complement numbers since neither value has a definite type.

Note: The compiler will translate boolean expressions into a compare-and-branch sequence. This sequence is constructed so that comparisons will be suppressed whenever possible. Thus, if it is found that the first boolean primary of a boolean AND is false, the evaluation of the other primaries in the AND may be skipped, since the value of the boolean AND must be false. Similarly, if it is found that the first boolean primary of a boolean OR is true, the evaluation of the other primaries in the OR may be skipped since the value of the boolean OR must be true.

3.4.3 Case Statement

```
CASE  =  '.CASE'  ( ADDRESS_EXPRESSION | ARITHMETIC_EXPRESSION )
                  '.OF'
                  STATEMENT
                  $( STATEMENT )
                  '.END'  ';'  ;
```

CASE statements permit the selection of one of a sequence of statements according to the current value of the WORD, INTEGER cell specified following the .CASE keyword. The statement whose ordinal number is equal to the cell value is selected for execution, and the other statements in the sequence are ignored. Following the completion of the selected statement, control is passed to the statement following the CASE statement.

Example:

```
.CASE  R3  .OF
    R1=R1+R2;
    R1=R1-R2;
    R1=R1 .ROL;
    R1=R1 .ROR;
```

```
   R1=R1 .INC;              #: IDENTICAL TO R1=R1+1 :#
   R1=R1 .DEC;              #: IDENTICAL TO R1=R1-1 :#
   R1=R1 .NEG;              #: IDENTICAL TO R1=-R1 :#
   .END;
 #:ONLY ONE STATEMENT EXECUTED:#
```

3.4.4 While Statement

```
WHILE  =  '.WHILE'  BOOLEAN_EXPRESSION  '.DO'
                $( STATEMENT )
                '.END'  ';'    ;
```

The WHILE statement denotes the repeated execution of the sequence of statements following the keyword .DO, upto the keyword .END, as long as the boolean expression following the keyword .WHILE is true.

Examples:

```
.WHILE R0¬=@R1 .DO R1=R1+2;  .END;
```

In this case the value pointed to by R1 is compared to the value of R0, and when they are not equal R1 is stepped by two. Thus, if R1 points to a table of values, the table may be searched for a match with the contents of R0.

```
.WHILE R0¬=@R1' .DO #:NULL:#; .END;
```

This example is similar to the one above, except that R1 is incremented using the auto-increment feature. Since the comparison is imprecise, ie. neither value has definite attributes, it will be made assuming WORD INTEGER values, and the amount of the auto-increment will be 2. In this example R1 will be 2 greater than the value at which the match was found since the auto-incrementing takes place regardless of the results of the comparison.

3.4.5 For Statements

```
FOR  =  '.FOR'
        FOR_LIST  $( ',' FOR_LIST ) #:SEQUENTIAL EXECUTION:#
        '.DO'
        $( STATEMENT ) #:FOR LOOP BODY:#
        '.END' ';'  ;


FOR_LIST  =  CELL '=' ( ARITHMETIC_EXPRESSION
                      | ADDRESS_EXPRESSION )
             ( '.STEP' ( '-' | .E ) ( ARITHMETIC_VALUE | ADDRESS_VALUE )
             | .E #:STEP BY 1:#  )
             '.UNTIL'
                ( '=' | '¬=' | '<' | '>' | '<=' | '>=' | '¬<' | '¬>'
                | .E #:DEFAULT IS ">" :#
                ( ARITHMETIC_VALUE | ADDRESS_VALUE )
               #:CELL, EXPRESSION, AND .STEP AND .UNTIL VALUES :#
               #:   MUST AGREE IN TYPE AND LENGTH.              :#
             ( '.WHILE' BOOLEAN_EXPRESSION
                      #:MAY NOT CONTAIN CONDITION TESTING:#
             | .E )   ;
```

The FOR statement specifies the repeated execution of the sequence of statements enclosed in the keyword delimiters .DO and .END, which will be called the FOR_loop body. The statements in the FOR_loop body are executed as long as all the statements of the FOR_lists have not been completed. More than one FOR_list can be given in a FOR statement, indicated by separating the FOR_lists with commas. Multiple FOR_lists are used sequentially, ie. the statements in the FOR_ loop body are executed the appropriate number of times for the first FOR_list, and then control passes to the second FOR_list, etc.

The FOR_lists consist of a cell (the control cell), an equals sign followed by an expression similar to an assignment statement, specifying the initial value for the control cell, an optional step value which follows the keyword .STEP, and a limit value which follows the keyword .UNTIL . Preceeding the limit value may be an equality or relational operator specifying the condition at which the FOR_list is complete. When neither an equality or relational operator is supplied, the relation "greater than" (>) is assumed. If no step value is given, the default is 1.

At the beginning of the execution of a FOR statement (FOR_list) the control cell is set with the initial value and is then compared to the limit value using the specified relation. If the comparison yields a true result, the FOR_list is satisfied and the entire process is repeated with the subsequent FOR_lists, if any were given. If the specified condition between the value of the control cell and limit value is not met, ie. the comparison yields a false result, the statements in the FOR_loop body are executed. Then the step value is added to the control cell. The process of comparing the control cell value with the limit value, executing the statements in the FOR_loop body and incrementing the control cell value is repeated until the comparison condition is true, at which point control is passed to the next FOR_list if multiple FOR_lists were specified. When all the FOR_lists of a statement have been satisfied, control passes to the statement following the .END delimiter of the FOR_loop body.

Examples of FOR statements:

```
.INTEGER  .RELATIVE  ARRAY(20),I;
R1=0;
.FOR  I=1  .UNTIL  10;
      .DO
      ARRAY(R1')=I;
      .END;
```

The array elements are set with their ordinal index. The control cell I takes on the values 1 to 10 in steps of 1, while the appropriate element of the array is selected using the initial zero value of R1 with auto-increment (by 2 since the array is made up of WORDs by default).

```
.INTEGER  .RELATIVE  ARRAY(40),I;
R1=0;
.FOR I=1 .UNTIL 10;  .STEP -1 .UNTIL >1
      .DO
      ARRAY (R1')=I;
      .END;
```

Similar to above, except that the array elements will have the values 1 to 10 and then 10 to 1, sequentially.

3.4.6 Blocks

```
BLOCK =   '.BEGIN'
          $(  IDENTIFIER_DECLARATION  )
          $(  PROCEDURE_DECLARATION  )
          $(  STATEMENT  )
          '.END'  ';'   ;
```

Blocks are used to group a sequence of statements into a structural unit which as a whole is classified syntactically as a statement. Identifiers may be declared within a block. These identifiers may be used in any of the subsequent declarations or statements within the block, but are not known outside the block unless they have the EXTERNAL attribute.

Examples of Blocks:

```
.IF R0=1 .THEN .BEGIN
               R1=2;
               R2=5;
               .END;
```

The THEN clause is made up of a block. All of the statements in the block will be executed if the boolean expression holds.

```
.CASE R1 .OF
         X=1;
         .BEGIN X=1; Z=1; .END;
         X=3;
         .END;
```

The second case of this CASE statement is a block, thus allowing more than one action to be performed.

```
STATEMENT = $( LABEL_IDENTIFIER  ':' #:MUST BE ON SAME LINE AS LABEL:# )
           ( IF  |  WHILE  |  FOR  |  CASE  |  GOTO
           |  PROCEDURE_CALL  |  FUNCTION
           |  ASSIGNMENT
           |  BLOCK
           |  RETURN
           |  NULL )  ;


LABEL_IDENTIFIER   =  .L  $( &( .A |  '_' ) )   ;
```

Any statement may be preceeded by an arbitrary number of labels, each followed by a colon (:) which must be on the same card as the associated label. Label identifiers are not explicitly declared, but are declared contextually by their appearance as a label of a statement. The label identifier may be chosen freely, with the restrictions that no two statements in the same block may have the same label, and that the identifier is not the same as any other declared identifier within the block.

There are two types of declarations, the identifier declaration, used to associate an identifier with a particular cell, and the procedure declaration, used to associate an identifier with a procedure.

3.4.7 Procedures

```
PROCEDURE_DECLARATION
        =  '.PROCEDURE'  SIMPLE_IDENTIFIER  ( '(' REGISTER ')'
                                            | .E #:LINK VIA STACK:# )
                              ';'
                    $( IDENTIFIER_DECLARATION )
                    $( PROCEDURE_DECLARATION )
                    $(  STATEMENT  )
                    '.END'  SIMPLE_IDENTIFIER #:PROCEDURE NAME:#  ';'
      |  '.PROCEDURE' '.EXTERNAL'  #:TO DECLARE PROCEDURE EXTERNAL:#
                      SIMPLE_IDENTIFIER  ( '(' REGISTER ')'
                                         | .E #:LINK VIA STACK:# )
          ';'
           #: FOR IDENTIFYING EXTERNAL PROCEDURE NAMES :#
      ;
```

A procedure declaration serves to associate an identifier with a sequence of declarations and statements, called a procedure body. This identifier may then be used in a procedure CALL statement to invoke the execution of the procedure body anywhere within the block in which the procedure declaration resides. The identifier is given the PROCEDURE attribute.

There are two types of procedures, INTERNAL and EXTERNAL. INTERNAL procedures have their procedure bodies specified within the declaration, and may be invoked only within the block in which the declaration resides. EXTERNAL procedures, whose declaration is preceded by the keyword .EXTERNAL, have their procedure bodies specified elsewhere. EXTERNAL procedures may be invoked from any block in which an appropriate declaration resides.

Following the keyword .PROCEDURE is a simple identifier which is the procedure name, and an optional link register enclosed in parentheses. The link register specifies which register is to contain the linkage pointer to the procedure CALL statement which invokes the procedure. During a procedure CALL in which a register is specified, the original contents of the link register are saved in the stack before it receives the linkage pointer. When no register is specified, the linkage pointer is stored in the stack. A semicolon must follow the procedure head (ie. the procedure, identifier and optional register).

For INTERNAL procedures, the procedure body follows the procedure head. The procedure body is terminated by the keyword .END followed by the procedure name and a semicolon. The procedure identifier is required following the .END to allow the compiler to check that all blocks and procedures within this procedure body have been terminated, ie. a matching .END was found for each. When a procedure is invoked, the statements in the procedure body are executed in sequential order, assuming none of the statements alter the flow of control. Control is passed back to the point of invocation when the .END delimiter of the procedure body is reached. The linkage pointer should be in the specified link register when the .END delimiter is reached. If no link register was specified, the linkage pointer is retrieved from the stack.

For EXTERNAL procedures, only the procedure head is required. The procedure body may then be specified in another, seperately compiled, procedure. References to it are resolved in the same external symbolic environment that is used for resolving references to other identifiers with the EXTERNAL attribute.

3.4.8 Procedure Call Statement

```
PROCEDURE_CALL  =  '.CALL'  SIMPLE_IDENTIFIER #:ATTRIBUTE=PROCEDURE:#
                   ';'
```

The procedure CALL statement is used to invoke the procedure whose identifier follows the keyword .CALL . Upon completion of the invoked procedure, control is automatically returned to the statement immediately following the CALL. The identifier in the CALL must have the attribute PROCEDURE, and therefore must have previously appeared in a PROCEDURE declaration. It is then known which register is to be used as the link register.

3.4.9 Return Statement

```
RETURN  =  '.RETURN'  ';'  #: GENERATES APPROPRIATE RETURN JUMP:#  ;
```

The RETURN statement may be used to force the return of control to the point of invocation from anywhere within the procedure, and is thus equivalent to reaching the .END delimiter of the procedure declaration.

3.4.10 Goto Statement

```
GOTO    =  '.GOTO'     LABEL_IDENTIFIER ';' ;
```

The GOTO statement provides for the explicit transfer of control to the statement whose label matches the label identifier in the GOTO statement. This identifier must be the label of a statement within the procedure in which the GOTO statement resides, although not necessarily within the same block. The interpretation of a GOTO statement proceeds as follows:

1. Consider the smallest possible block containing the GOTO statement.

2. If the label identifier designates a statement in this block, program execution resumes at that point. Otherwise, execution of the block is regarded as terminated and the next largest block is considered. Step 2 is then repeated. Care should be used when branching to the inside of a FOR, CASE, or WHILE statement body. No check is made for this possibility and execution will proceed as described for these statements using the current values of the control variables. It is impossible to branch into a nested inner block or procedure.

3.4.11 Functions

```
FUNCTIONS =(  '.HALT'  |  '.WAIT'  |  '.RESET'
           |  '.EMT' '(' POSITIVE_VALUE #:VALUE < .O(400):# ')'
           |  '.TRAP' '(' POSITIVE_VALUE #:VALUE < .O(400):#  ')'
           |  '.CLR'  '('  '.'  $(&( C | V | Z | N ) )  ')'
           |  '.SET'  '('  '.'  $(&( C | V | Z | N ) )  ')'
           |  '.IOT'  ( '(' IDENTIFIER  $( ',' IDENTIFIER ) ')'
                          #:LITERALS ONLY AS IOT PARAMETERS:#
                      | .E )
           )  ';'   ;
```

Functions provide a method of interacting with the PDP11 hardware. All but one of the functions correspond directly to the machine operation with the same mnemonic. The positive values of the trap (.TRAP) and emulator trap (.EMT) functions correspond to the value in the machine instruction available for transmittal to the trap routine. As a result they must be within the range 0 to .O(477). The clear (.CLR) and the set (.SET) functions provide for the explicit setting or clearing of one or more of the condition code bits. The bits to be modified are indicated by the C, V, Z, and N codes which correspond to their similarly named bits in the Processor Status Register. The first code must be preceded by a point (.). If several bits are specified, their code letters should be concatenated.

The IOT function allows interaction with the PDP11 input/output executive system. The IOT function may be followed by any number of LITERAL identifiers, enclosed in parentheses and separated by commas. The values of LITERALs will be generated in-line and are used as parameters by the input/output executive. There are no restrictions on the number, type or length of the LITERALs; WORD LITERALs will be aligned to a full word boundary if necessary, while BYTE LITERALs will be packed in the resultant code.

3.5 Program Procedures

```
PROGRAM = '.PROCEDURE'
            ( '.MAIN'
            | '.INTERRUPT'
            |('.EXTERNAL'| .E ) #:EXTERNAL:#  )
          SIMPLE_IDENTIFIER
          (  '(' REGISTER ')'  #:INVALID FOR .INTERRUPT OR .MAIN:#
          |  .E #:LINK VIA STACK:#  )
          ';'
          $( IDENTIFIER_DECLARATION )
          $( PROCEDURE_DECLARATION )
          $( STATEMENT )
          '.END'  SIMPLE_IDENTIFIER #:PROCEDURE NAME:#
          ';'   ;
```

There are three types of separately compiled program procedures that may be used as PDP11 programs. A MAIN program is the procedure which receives control after loading an entire program into the PDP11. An EXTERNAL program procedure is a sub-program which receives control via a procedure CALL statement. An INTERRUPT program procedure is a sub-program procedure that receives control via an interrupt.

The principal difference between the three types of program procedures is the way in which they receive and return control. A MAIN procedure, as indicated by the keyword .MAIN, receives control from the PDP11 loader program via a jump, and upon completion enters into the WAIT state with a jump to itself. No link register should be given for a MAIN program procedure. An EXTERNAL program procedure receives control via a procedure CALL (a PDP11 JSR instruction), and upon completion returns control via a PDP11 RTS (return from subroutine). EXTERNAL procedures may specify a link register, and if one is given it is the programmer's responsibility to see that the link address is contained in the register upon completion of the procedure (whether indicated by a RETURN statement or by reaching the procedure .END). The EXTERNAL program procedure is the only way to specify the body of an EXTERNAL procedure. An INTERRUPT program procedure, as indicated by the keyword INTERRUPT preceeding the procedure head, receives control via an interrupt vector which provides both the starting point of the interrupt and an initial value of the processor status register (the original values of these two registers are saved in the stack). INTERRUPT procedures return control via a PDP11 RTI (return from interrupt). No link register may be given for INTERRUPT procedures. Two special keywords may be used within an INTERRUPT procedure as SYNONYM identifiers. .OLDPC and .OLDPS may be used as SYNONYM identifiers (for WORD,STACKED identifiers only) for the old program counter value and old processor status value respectively which were pushed into the stack by the interrupt handler. Thus, an interrupt routine may gain access to these values, and, indirectly, to a trap value.

Following the program procedure head is a procedure body that is identical to that of an INTERNAL procedure. Like the INTERNAL procedure declarations, the procedure body is terminated by .END keyword followed by the procedure name and a semicolon.

3.6 Identifier Block

```
IDENTIFIER_BLOCK
     =    '.DATA'   SIMPLE_IDENTIFIER
           (  '.LOCATION'  '(' POSITIVE_VALUE ')'
           |  .E   )
          ';'
          $(  IDENTIFIER_DECLARATION )  #: WITH SPECIAL RESTRICTIONS:#
          '.END'  SIMPLE_IDENTIFIER  #:IDENTIFIER BLOCK NAME:#
          ';'
```

IDENTIFIER blocks provide the only means of specifying the location, names and values of EXTERNAL, non-procedure identifiers. An IDENTIFIER block begins with the keyword .IDENTIFIER followed by the name of the block, and optional location value in parentheses following the keyword .LOCATION (specifying the absolute address at which the block is to be loaded), and a semicolon. The body of an IDENTIFIER block consists solely of identifier declarations. The RELATIVE identifiers in these declarations have the default scope EXTERNAL. The space for the identifiers is allocated within their block, and is relative within the block. The identifiers may be initialized and synonymized within the block. The identifier declarations must satisfy the following restrictions:

1. No allocation attribute other than RELATIVE and LITERAL may be specified.

2. No LOCATION modifier may be given.

The identifiers declared within the IDENTIFIER block are compiled in exactly the order in which they are presented, thus allowing for the initialization of interrrupt vectors. The LOCATION value of the identifier is used to properly locate the vectors.

Examples of Identifier Blocks:

```
.DATA         TRPVCT  .LOCATION(.O(30));
.EXTERNAL .ADDRESS EMIT #:EMULATOR TRAP ROUTINE:#
                  ,TRAPI #: TRAP ROUTINE :#;
.ADDRESS VECTOR(8)=(.ADDRESS(EMIT),.X(0020)#:PROCESSOR STATUS
                    (PRIORITY):#,
          .ADDRESS (TRAPI),.X(0060)#:PROCESSOR STATUS (PRIORITY):#);
.END TRPVCT ;
```

The emulator trap and trap vectors could be specified in this manner.

```
.DATA PRTBLK ;
    .INTEGER PLINE=0,MAXLINE=60,
             PAGE=0;
.END PRTBLK ;
```

An identifier block of this type might be used to hold printer control information.

3.7 Program Deck

```
DECK   =   (  OPTIONS   |   .E #:USE DEFAULTS:#   )
          $(  PROGRAM   |   IDENTIFIER_BLOCK   )
          '.DECKEND'    ;
```

```
OPTIONS =   '.OPTIONS'
            '('
            $( ('NO' | .F )
               &( 'FOBJ'   #:FORMATTED OBJECT LISTING:#
                | 'BOBJ'   #:COMPLETE OBJECT CODE LISTING (BLOCKED):#
                | 'XREF'   #:CROSS REFERENCE DICTIONARY:#
                | 'ATTR'   #:LIST OF ALL ATTRIBUTES FOR IDENTIFIERS:#
                | 'LOAD'  #:OBJECT CODE PRODUCED ON "//SYSOUT" :#
                | 'COPY'  #:OBJECT CODE COPY PRODUCED ON "//SYSCOPY":#
                | 'ERRL'  #:ERROR REPORT GENERATED ON "//ERREP" :#
                | 'SOPT'  #:STORAGE OPTIMIZATION:#
                | 'HSAU'   #:COMPILE CODE FOR HIGH SPEED ARITH. UNIT:#
               )  ','
             )   ')'   ';'

            #:DEFAULTS=    :#
            #:   NOFOBJ    :#
            #:     BOBJ    :#
            #:     XREF    :#
            #:     ATTR    :#
            #:     LOAD    :#
            #:   NOCOPY    :#
            #:   NOERRL    :#
            #:   NOSOPT    :#
            #:   NOHSAU    :#   ;
```

A deck, as input to the compiler, consists of an optional OPTIONS statement, followed by a sequence of procedures or IDENTIFIER blocks. The options specified in an OPTIONS statement hold for all the programs or IDENTIFIER blocks that follow, each of which is treated as a completely seperate entity.

An OPTIONS statement consists of the keyword .OPTIONS followed by a parenthesized list of options seperated by commas, and terminated with a semicolon. Each option may be concatenated with the prefix NO to indicate if is not desired. There are defaults for all the options. The HSAU option tells the compiler that the arithmetic dyadic operators * and /, the arithmetic monadic operator .AHS, and the logical monadic operator .LHS may be used in this compilation. A detailed discussion of the other options is included in Section 4.

Source code may appear in columns 1-72, inclusive. Columns 73-80 may contain sequence numbers which will be reproduced in the listing. The source code may be free format, ie. statements may be split over more than one card. Keywords, identifiers and character strings must appear entirely on one card (ie the interface between column 72 and column 1 is considered to be a blank.) Any number of statements may appear on a line, subject to the physical limitation of 72 characters.

3.8 Commentary

The user may insert commentary anywhere in a deck that a blank may appear (ie. preceding any terminal string or generic terminal that is not concatenated as indicated by the metalinquistic &). Commentary is delimited by #: at the beginning of the comment and :# at the end. Commentary is ignored by the compiler.

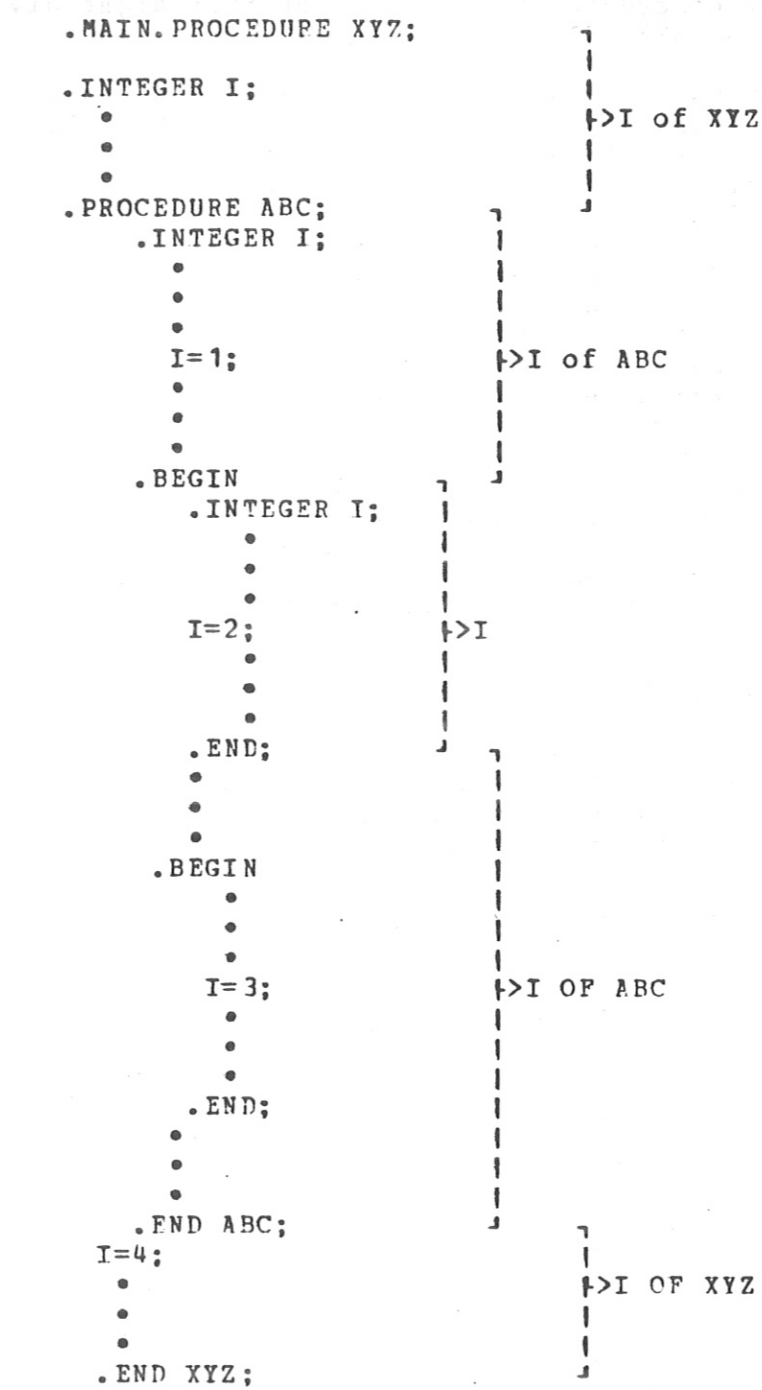
3.9 Scope, Allocation and Use of Identifiers

In general, an identifier is known, and usable only within the procedure or block in which it is declared, and within any inner nested procedures or blocks. When an identifier is used within an inner nested block or procedure, the following method (which is analogous to that used for labels of GOTO statements) is used to determine to which declaration the identifier refers:

1. Consider the smallest block or procedure containing this instance of the identifier.

2. If the identifier appears in a declaration within the block or procedure this is the declaration that is used, ie. the cell to which this identifier refers. Otherwise, the next largest surrounding block or procedure is considered and step 2 is repeated.

Example:



The use of the identifier I within each of the three groups is completely independent of its use in the others, ie. the different I's refer to completely different cells. There are some restrictions that must be placed on the use of STACKED cells within nested procedures. The compiler allocates space for STACKED cells upon entry to the procedure or block within which the declaration appears, by bumping the stack pointer. It frees the space upon exit by restoring the stack pointer to its value upon entry. The compiler knows the relative location with respect to the stack pointer of these cells in the stack and will generate the appropriate code to access them. Errors could occur if the stack pointer is not maintained at the relative location that the compiler assumes is indicated. This situation may arise when a STACKED variable in an outer procedure is refered to from within a nested procedure (nested by its declaration) which is not at the same actual level of nesting at execution time. This is the case with recursive procedures. The compiler will monitor the declared nesting level of procedures or blocks and the potential actual

nesting level (by noting the use of procedure CALL statements), and will flag as an error any procedure CALL statement that might give rise to incorrect access of STACKED cells.

There are three methods of leaving a block: a GOTO statement, a RETURN statement, and reaching the .END delimiter of the block. All must be able to correctly return the stack pointer to its value upon entry into the block. As a result, a certain amount of additional overhead is incurred by the use of the RETURN and GOTO statements. Furthermore, the use of a GOTO statement which references a yet unreached label (ie. a forward branch) may be slightly wasteful of storage. Enough space must be set aside for a branch out of the current block and the attendant restoration of the stack pointer, even though this space may not be required if the label is subsequently found to be within the current block.

It should be noted that the compiler often allocates, temporarily, values in the stack which are used in compiled code. The CASE statement uses the stack transiently to calculate the branch table address for the CASE. The FOR statement uses the stack to keep track of which FOR_list of a multiple FOR_list FOR statement is currently in control. The stack is not used if only a single FOR_list is specified.

3.10 Errors and Error Recovery

When errors are detected during the scan of the source program, they are flagged in the output listing (c.f. Section 4.1.1). The compiler then attempts to recover control by skipping the remainder of the statement in error, ie. by skipping to the terminating semicolon.

4. PPL Compiler Output

The compiler for the PPL language may generate output on four different data sets: a print data set, containing at least a listing of the source statements and error messages, a data set for the compiled object code, a data set for a copy of the compiled object code (which might be used as a back up copy on tape, punched cards, etc.), and a fourth data set for a separate error report.

4.1 Printed Output

Some of the printed output is optional and may be selected by specifying the appropriate option in the OPTION statement. The listing of the source code and attendant error messages is not optional.

4.1.1 Obligatory Printed Output

The obligatory printed output contains a cover page which lists the OPTIONS statement followed by all the options used during the compilation. In addition, a listing of all source statements and attendant error messages is always printed. Each source program or identifier block is listed separately, with the first card used as a title on each page. The title line also contains a page number. Each page, other than the options page, contains an appropriate subtitle. The subtitle used for the source code listing is shown in figure 1.

```
-------------------------------------------------------------------------------
          ||                        |                         |           |1   1
COLUMN    ||          1  1        1|2                         |9         9|2   2
 NUMBER   ||1       8  2  5       9|0                         |2         9|0   4
          ||                        |                         |           |
===============================================================================
          ||                        |                         |           |
HEADING   ||  LOC    BLK N   STMT   |==== CARD IMAGE ====     | SEQ.      |CDNO
__________||________________________|_________________________|___________|_____
          ||                        |                         |   card    |
CONTENTS  ||000000   DD D    DDD    |       card image        |sequence   |DDDD
__________||________________________|_________________________|__field____|_____
```

O= Octal digits
D= Decimal digits

FIGURE 1. Source Program Listing Layout

Each source card line will contain the following information under the appropriate heading:

- LOC - The relative location, in octal, of the first statement on the card
- BLK - the block number of the current block
- N - the nesting level of the block, procedure, or sequence of statements to which the statement belongs
- STMT - the number of the first statement on the line
- CARD IMAGE - the source card image, including the sequence field
- (SP+2) - the stack depth (+2) at the beginning of the line
- CNDO - the card number of the card within the input deck

Errors detected in the source statement will be flagged with a V above the character at which the error was detected. Preceding the error flag will be ERROR AT V in the first 10 columns of the line. Following the error flag will be an error code. A complete list of all error codes, and an explanation of each, is given in Appendix A. Following the listing of the source program will be a copy of the error report information. This lists for each error the card number, card sequence field, error number, and source code surrounding the error point (cf. 4.3).

Each INTERNAL procedure will be flagged by a line specifying the name of the procedure in columns 11 through 18. This line will precede the source card containing the procedure head for the procedure.

4.1.2 Optional Printed Output

4.1.2.1 Formatted Object Code Listing

If the option FOBJ is in effect, a formatted listing of the object code is generated for each source statement. The formatted object code is listed in line with the source statements. On each line appears the relative location in octal of the word or byte (column 50), followed by a colon. For identifier declarations the relative location has the following meaning:

RELATIVE - the location relative to the beginning of the program procedure for identifiers with the INTERNAL attribute,or zero for identifiers with the EXTERNAL attribute

STACKED - the relative displacement of the variable from the stack pointer value upon entry to this procedure or block. This value is enclosed in parentheses.

LITERAL - the location field is left blank

The code necessary to bump the stack pointer follows the identifier declarations of a block or procedure.

The object code word or byte begins in column 60 and is expressed in octal form as follows:

RELATIVE identifiers - initial values if any are given

STACKED identifiers - blank

LITERAL identifiers - the value as it appears when referenced in statements

machine code instruction - the first word contains the various fields, as defined in the PDP11 handbook, separated by blanks. Any additional words needed by an instruction are listed on subsequent lines.

Where a value is not known, for example the relative address of a forward jump, question marks are substituted.

The statement number of the first statement on the line appears in the STMT column of the first line of code generated for a statement. Each identifier declaration word contains the identifier name beginning in column 60, and each machine instruction explicitly specified by the source code contains the corresponding source code beginning in column 60.

4.1.2.2 Blocked Object Code Listing

If the option BOBJ is in effect, a listing of the object code is generated on a separate, appropriately subtitled page following the the program procedure .END . Forward references to labels have been resolved, and thus the corresponding branch or jump instructions have their offset or index words filled. The blocked object code listing consists of PPLLINK commands, and object code in the following format: 16 octal words per line with either 6 octal digits each and 2 spaces between each word, or a pair of octal byte values (3 digits) followed by a space. The subtitle for the blocked object code listing is shown in figure 2.

COLUMN NUMBER	1		
HEADING	OBJECT	CODE WITH LINKER COMMANDS	
CONTENTS	linker object	commands or code with 16 words (in octal) to the line	

FIGURE 2. Blocked Object Code Listing

4.1.2.3 Cross Reference Listing

If the XREF option is in affect, an alphabetized cross reference listing is generated for each identifier declaration and its use. The cross reference listing follows the blocked object listing, beginning on a new appropriately subtitled page. The page and subtitle layout for the cross reference list are shown in figure 3. Each declared instance of an identifier is listed separately, with its location value, length, declaration card number, and the statement number of each reference to the identifier.

The location value field is in octal, and has the following meanings for identifiers:

RELATIVE - displacement from beginning of the procedure or identifier block in which the cell is located

STACKED - position in the stack (relative to the stack pointer upon entry to the current procedure or block)

LITERALS - the value of the literal

LABELS - displacement of the statement to which the label is attached from the beginning of the program procedure

PROCEDURE - displacement of the procedure from the beginning of the program procedure

REGISTER - the register number

The length field is coded in octal, and indicates the number of bytes required for the identifier, ie. the size in bytes for a LITERAL, cell, or array of cells, or the length for a procedure. The length for label identifiers is zero.

The declaration field is coded in decimal and gives the statement number in which the identifer is declared. For label identifiers, the declaration field contains the statement number of the statement which it labels.

The statement number of each statement that references the identifier is listed in the reference section. Those statements in which the value of the cell to which the identifier refers is altered, are flagged with a star (*) following the number. ie. those occurrences of the identifier on the left side of the assignment operator (=) are flagged.

COLUMN NUMBER	1	8	10	15	18	21	25	27	31
HEADING	SYMBOL		LOC		LEN		DCL		REFERENCES
CONTENTS	LAAAAAAA		OOOOOO		OOOOO		DDD		DDD DDD ...

L: Letter A: Alphanumeric
O: Octal digit D: Decimal digit

FIGURE 3. Cross Reference Listing Layout

4.1.2.4 Attribute Listing

If the ATTR option is in affect, a list of all the attributes of each declared instance of an identifier is generated. If the XREF option is also in affect, the attribute listing appears, instead of references, as the first line in the cross reference listing. If the XREF option is not in affect, the location value, length and declaration card number fields of the cross reference listing are given with the attribute listing, but no references are listed. Following the list of attributes is the identifier name. Each procedure name or block number within which this identifier is nested is listed, in nesting order, separated by periods.

4.2 Object Code Output

Object modules can be produced on two different data sets, a SYSOUT data set which is identified by a //SYSOUT DD card, and a SYSCOPY data set which is identified by a //SYSCOPY DD card. The primary object code output appears on the SYSOUT data set if the LOAD option is in affect. The object code produced is compatible with that used by the LINK-11 program linker. If the LOAD option is selected and the COPY option also specified, a copy of the object modules is generated on the SYSCOPY data set.

4.3 Error Report Listing

When the ERRL option is in affect, a separate error report data set is generated on the SYSERRER data set, which is specified on a //SYSERRER DD card. Each error flagged in the source listing is noted in the error report with the following information:

1. the error number
2. the 10 characters on each side of the error position, enclosed in apostrophes and separated by a question mark.
3. the card number followed by a slash (/) and the sequence field (if non-blank) of the card on which the error occurred.

Appendix A. Error Codes.

1. Invalid OPTION keyword.
2. First statement is not a procedure or IDblock.
3. .LOCATION must be followed by a parenthesized positive value, less than .O(177777).
4. Missing right parenthesis.
5. LOGICAL INTEGER values must be enclosed in parentheses.
6. Invalid LOGICAL INTEGER code for this type.
7. LOGICAL INTEGER values may not be void.
8. Only INTEGER and ADDRESS literals may be used in a defined expression.
9. Undefined identifier.
10. Register name may not be used as a SIMPLE_ID.
11. Semicolon is not found where expected.
12. More than one attribute from same class.
13. Literals may not be arrays or have the .RELATIVE attribute.
14. .STRING implies LOGICAL BYTE array.
15. .STRING length must be non-zero, positive and parenthesized.
16. Multiply defined identifier, or invalid attribute keyword.
17. Array size must be a non-zero positive value.A WORD array may only be allocated an even number of bytes.
18. .LOCATION valid only for .RELATIVE identifiers.
19. Literals may not have synonyms.
20. .SYNONYM defined memory cell must be enclosed in parentheses.
21. Defined memory cell must have type .INTEGER, .ADDRESS, or .LOGICAL, and may not be a literal.
22. Array expression may not follow a simple cell identifier.
23. Array displacement must be positive. Displacement for a WORD array may only be an even number of bytes.
24. Synonym BYTE offset possible only for setting BYTEs synonymous to WORDs.
25. Synonym BYTE offset must be positive with a value of zero or one.
26. Identifier declarations may not follow Procedure declarations.
27. .ADDRESS fill value may only be used with ADDRESS identifiers.
28. Invalid defined memory cell identifier.
29. Improper alignment of synonym.
30. Defined array cannot be contained within the SYNONYM array.
31. A simple WORD cell cannot be set synonymous to a simple BYTE cell.
32. .ADDRESS defined memory cell must be enclosed in parentheses.
33. Array length should not be zero.
34. Compound fill value legal only for arrays.
35. Iteration count must be positive.
36. Invalid or missing fill value.
37. Character string must be entirely on one card.
38. Fill value not containable in cell of declared size.
39. Character string of length >1 valid only for .STRING initial value (.LOGICAL .BYTE array).
40. Character string of one character valid only as a LOGICAL BYTE value.
41. Fill value exceeds bounds of cell or array.
42. In a fill value, .ADDRESS defined memory cell must be RELATIVE, or ABSOLUTE for .LOCATION cells.
43. Invalid fill value for the declared identifier.
44. Invalid specification for an identifier definition.
45. Fill value required for literals.
46. Synonyms must have same allocation attribute.
47. STACKED identifiers may not appear in an identifier block.
48. .LOCATION specification may not be used in an IDblock.

49. Only declarations may appear in an identifier block.
50. Logical end of program block, but block name not found.
51. Invalid program or IDblock.
52. Only a register name can be used as a link.
53. Invalid statement or declaration keyword.
54. Invalid statement keyword.
55. Label is already used as an identifier in this block.
56. .GOTO may not transfer control into a block/procedure from outside.
57. Multiply defined label.
58. Label delimiter : must be on same line as label.
59. Invalid or no statement following label.
60. Invalid operator in assignment statement; semicolon not found as expected.
61. Label identifier not found as expected.
62. End of procedure occurred before all blocks closed.
63. Semicolon not found following .END .
64. Semicolon not found following EXTERNAL procedure declaration.
65. Semicolon not found following procedure head.
66. Symbol following .END is not a procedure name.
67. Logical end of procedure, but correct procedure name not found.
68. Procedure identifier not found following .CALL .
69. Invalid call, Outer stacked variables referred to within the .CALLed procedure can not be accessed correctly.
70. Condition code bits for .CLR and .SET must be enclosed in parentheses.
71. Each condition code bit may appear only once in a .CLR or.SET.
72. Invalid condition code bit in a .SET or .CLR, or missing right parenthesis.
73. Invalid or missing condition code bit indicator in a .CLR or .SET.
74. .TRAP value must be less than .C(400), and enclosed in parentheses.
75. Only addresses may be used for indirect addressing.
76. Symbol is not a literal and may not be used in a defined expression.
77. Offset (displacement) in array exceeds array bounds.
78. Register identifier does not follow auto-decrement operator.
79. Signed expression expected as byte offset of register indirect address.
80. Only a positive value may be used as displacement in a STACKED array.
81. Invalid use of identifier as cell.
82. BYTE displacement of item in an array not found as expected.
83. Literal value not containable in cell of desired size.
84. The base address of this indexed array cell is outside of memory.
85. Invalid cell type for .CASE argument.
86. .OF (case body delimiter) not found as expected.
87. Empty .CASE body not permitted.
88. Invalid cell type for an assignment statement.
89. Invalid mix of cell types in an assignment statement.
90. Assignment operator not found as expected.
91. Literals may not appear on left of assignment operator.
92. Cell not found as expected following addition operator.
93. Not a valid beginning of an expression.
94. Cell length does not agree with expression or assignment statement length.
95. This operator may not be used with the NOHSAU option.

96. Only a positive value may be specified as shift value.
97. A parenthesized shift amount must follow the .AHS operator.
98. Invalid shift amount specification in .AHS.
99. Only the values +1 and -1 may be added to or subtracted from BYTES.
100. Memory cell in parentheses must follow .ADDRESS.
101. .ADDRESS may only be used in address expressions.
102. A character string may only be used as a .LOGICAL .BYTE value.
103. Literals may not be used as the argument of a .ADDRESS.
104. Only single characters may be used as LOGICAL values.
105. .SWAB valid only with LOGICAL WORD values.
106. Only LOGICAL INTEGERs may be used as LOGICAL values; decimal INTEGERS may not.
107. BOOLEAN_AND not found as expected following | .
108. BOOLEAN_PRIMARY not found following ¬ sign.
109. BOOLEAN_EXPRESSION not found following left parenthesis.
110. Invalid BOOLEAN_PRIMARY.
111. Relational operator not found as expected.
112. Comparison values do not agree in type.
113. Second comparison value not found as expected.
114. Logical value in relation does not agree in length with operator.
115. Comparison values do not agree in length.
116. BOOLEAN_EXPRESSION not found following .IF.
117. .THEN not found following BOOLEAN_EXPRESSION.
118. Statement not found in .THEN clause.
119. Statement not found in .ELSE clause.
120. BOOLEAN_EXPRESSION not found following .WHILE .
121. .DO not found following .WHILE boolean expression.
122. No specification found following = in an identifier declaration.
123. Arithmetic value not found following arithmetic dyadic operator.
124. Comparison of two literal values in a boolean relation.
125. STACKED identifiers may not be EXTERNAL.
126. .LOCATION, .SYNONYM, and fill values incompatible with .EXTERNAL.
127. Shift amount must be less than 16.
128. BOOLEAN_EXPRESSION for FOR-LIST limit not found as expected.
129. .DO (FOR-loop body delimiter) not following FOR-LIST(s).
130. FOR-LIST control cell has invalid type.
131. Invalid increment value.
132. .UNTIL (FOR-LIST limit delimiter) not found as expected.
133. Limit value not found following .UNTIL .
134. Limit value does not agree in type with control cell.
135. Only literals may be used as .IOT parameter.
136. Relations may only appear as the first primary in a BOOLEAN_EXPRESSION.
137. .ADDRESS value may not be used as a .FOR step value.
138. .ADDRESS value may not be used as comparison value.
139. .STACKED identifiers may not be preset.
140. Positive value expected.
141. Invalid operator sequence or undefined identifier in defined expression.

SECTION 2

The PDP11 Linkage Editor.

PDP-11 LINKAGE EDITOR

The PDP-11 linkage editor forms a portion of the PDP-11 processing system for use in conjunction with the IBM 360. Its purpose is to resolve references between program segments produced by the various language processors and, as a result, form a single block of code. The linkage editor accepts as input the output from the various PDP-11 language processors (object modules) and certain control commands and provides as output a block of processed code (load module) suitable for loading by the PDP-11 system loader.

Functions Performed

The linkage editor performs the following functions:

- allocate space for each control section (CSECT) received as input
- relocate address constants (ADCONs) used as pointers to internal or external symbols
- resolve external references to other user CSECTs or, via automatic library call, to system library routines
- output a load module which is either absolute or load-time block-relocatable.

Space Allocation

Each control section received as input to the linkage editor processor will contain an external symbol dictionary (ESD). The ESD contains a value which specifies the total length of the CSECT, thus enabling the linkage editor to allocate space. Also known to the linkage editor, via an input parameter or control card, is the highest core location available for program storage. Storage for relocatable object modules will be allocated from the top of core down. Therefore the first object module provided to the linkage editor will be allocated to the highest locations available, the second object module to the contiguous lower storage area, etc. This method is necessitated by the operation of the stack pointer (see the PDP-11 Handbook).

Address Constant Relocation

Relocation is performed on both external and internal address constants according to the commands found in the relocation dictionary.

External Reference Resolution

References to external symbols are resolved and the proper address is placed in ADCONs by the linkage editor. All external symbols which the linkage editor is unable to resolve are listed and the resulting load module is flaged as not executable.

Linkage Editor Input

The input to the PDP-11 linkage editor is made up of control statements and object modules. The two types of input data may be freely intermixed and, in general, need not adhere to any specific order. Control statements are used to provide information to the linkage editor concerning entry points and relocatability (APPENDIX A). The object modules are the ouput from the various PDP-11 language processors and contain generated code and formatted instructions for relocating the code and resolving external references (APPENDIX B). An object module is generated for each CSECT processed by the language processor. Several object modules may be generated from one source deck. Each object module contains one ESD and one END record and a sequence of RLD and TXT records. The RLD must always preceed the TXT record(s) to which it refers. The END record is the last record in the object module. The ESD may be placed before or after any RLD or TXT records.

Linkage Editor Output

Two types of load modules may be formed by the linkage editor. Load module type is controlled by an input parameter or control card. An absolute load module contains code with all addresses resolved and without the means available to move the code block to various core load points. It may therefore only be loaded at the load point specified to the linkage editor at link-time. The second type of load module is block-relocatable and may be relocated as a block at load-time. A primary difference is that a block-relocatable load module contains a modified relocation dictionary to relocate absolute address constants. This second load module type may be used for creating overlay structures.

Linkage Editor Operation

The linkage editor operates in a two pass mode in order to minimize the amount of core required for input data storage. A one pass linkage editor was considered, but such a program would require that all of the input text blocks be held in core so that references could be resolved. Furthermore, even though the linkage editor would externally appear as a one pass program, it would differ in no essential logical respects from the one provided.

Five tables are used in the linkage editor to store the data necessary to perform the ADCON relocation and external reference resolution functions. The tables are as follows:

SYMBOL. The SYMBOL table is a "hashed" list of all CSECT names, ENTRY names and external references found in the totality of object modules which will be used to form the load module.

TYPE. The TYPE table contains a flag for each entry in the SYMBOL table to indicate the attribute of the associated symbol. The TYPE flags will have the value provided in the TYPE field of ESD records. Symbols found in RLD records and not defined in

the ESD record(s) of the current CSECT will be flagged as external references. Symbols used in DATABLKs will have a TYPE flag of 1 (ENTRY name). At the end of the first pass, all symbols with the TYPE flag of 2 will be regarded as undefined and, therefore, as errors.

LENGTH. In the case of CSECT symbols, the LENGTH table will contain the total length of the named CSECT. For ENTRY names, the LENGTH table contains the relative displacement of the ENTRY name from the beginning of the CSECT.

ADDRESS. The ADDRESS table contains the absolute address of the indicated CSECT or ENTRY name. These addresses are then used to calculate the relocated ADCONs in the second pass of the linkage editor.

IDFLAG. The IDFLAG table contains, for symbols of TYPE flag=0 (CSECT), the CSECT number, which is the sequential number of the CSECT in the input stream. For ENTRY names, the IDFLAG table contains a pointer to the CSECT in which the ENTRY is located.

During the first pass, the input stream is examined for all ESD records. The information contained in the ESD and RLD records is inserted into the SYMBOL, TYPE, LENGTH and IDFLAG tables as described above. If a symbol of TYPE flag 0 or 1 is found which has previously been entered as a CSECT or ENTRY name, the new TYPE is ignored but a message is printed in the output listing to indicate that the symbol is multiply defined and the load module will be flagged as non-executable. The input stream is also examined for linkage editor control cards and appropriate control flags are set to indicate

1). the name of the load module entry point
2). the highest core location available
3). the type of load module to be generated.

At the end of the first pass, the ADDRESS table is filled in with the absolute address of each symbol in the SYMBOL table. Undefined symbols (TYPE 2) are noted in the output listing and the load module is flagged as non-executable.

In the second pass, the RLD and TXT records are scanned and processed. Each RLD record must proceed the TXT record(s) to which it refers. As an RLD record is scanned, the information about each relocation command is placed in an ordered (by address) string. Each node (RLDNODE) contains:

1). left and right pointers to the neighboring nodes (LPTR and RPTR)
2). a pointer to the symbol to be used in calculating the address (SYMIDX)
3). the relocation command type (COMD)
4). the displacement, relative to the start of the current CSECT of the address to be relocated.

As TXT records are scanned, the RLD is checked for

relocation commands and the necessary functions are performed. The resulting code is queued up for the clean-up phase of the linkage editor. A CCMD, as required by the PDP-11 disc monitor system, is also generated during pass two. The modified relocation dictionary required for load-time relocation is not generated since its characteristics are not presently known. At the completion of pass two, the clean-up phase is initiated, which results in the output of the COMD and the relocated TXT records. Execution of the linkage editor is then terminated.

APPENDIX A

CONTROL CARDS

There are three control cards accepted as input to the linkage editor. The linkage editor control cards may be inserted anywhere in the input stream. The order to the control cards is not important.

ENTRY

The ENTRY control statement is used to indicate to the linkage editor the name of the load module entry point. The ENTRY card contains the word "ENTRY" followed by the entry point name in free format. The length of the entry point name is not restricted. If the ENTRY card is omitted, the entry point will be the first entry point in the first object module in the input stream.

RELOC(ATION)

The RELOC control statement provides the information on the highest core location available for storage of the load module to be generated from the input stream. The RELOC card contains the word "RELOC(ATION)"followed by a decimal number which indicates the highest available core location available. The default load location will be 1024 plus the total length of the load module. Thus the last CSECT in the input stream will be relocated to begin at 1024 and the area available for dynamic storage utilizing the stack pointer will range from 256 to 1023.

BLOCK

The BLOCK control statement indicates to the linkage editor that the generated load module should contain the modified RLD for use by the relocating loader. Load modules generated in conjunction with the BLOCK control statement may be relocated at load-time as a complete block of code to any available area in core. Load modules to be loaded into core by the Overlay Supervisor should be generated using the BLOCK statement. The default load module output does not contain the modified RLD and, therefore, may not be relocated at load-time, although it may be loaded by the relocating loader.

APPENDIX B

OBJECT MODULE RECORD FORMATS

External Symbol Dictionary (ESD)

Bits	Content	
3	".ESD"	Block type identifier
1	blank	
6	RFLAG	if RFLAG is blank the following text blocks up to the next END record may be relocated. If RFLAG>=0 then the text blocks following should be loaded at the octal address given by RFLAG.
1	blank	
N	SYMBOL	CSECT or ENTRY name of arbitrary length. No intervening blanks are allowed. Normally , the first SYMBOL encountered will be the name of the CSECT.
1	blank	
1	TYPE	0=> CSECT name 1=> ENTRY name 2=> external reference
1	blank	
6	VALUE	if TYPE = 0 then VALUE is the length of the CSECT. If TYPE=1 then VALUE is the relative displacement (octal) of the entry point from the CSECT start in bytes. If TYPE=2 VALUE may be omitted.
1	blank	

Relocation Dictionary (RLD)

Bits	Content	
3	".RLD"	Block type identifier
1	blank	
2	COMD	relocation type command as described below.
1	blank	
6	DISP	relative displacement (octal) from the start of the CSECT of the word to be modified.
1	blank	
N	SYMBOL	name of CSECT, ENTRY or external to be

		in computing the relocation address. No SYMBOL is given for internal relocation.
1	blank	

Text Block

Bits	Content	
3	".TXT"	Block type identifier
1	blank	
6	LOC	relative displacement (octal) from the start of the CSECT for the following block of text.
1	blank	
3	DATUM	text data is provided as either bytes
6		or full words. All data is octal.
1	blank	

End of CSECT

Bits	Content	
3	".END"	Block type identifier
1	blank	
N	NAME	the name of the current CSECT.

APPENDIX C

RELOCATION DICTIONARY COMMANDS

Code — Description

0 — INTERNAL RELOCATION

The relocation base of the current CSECT is added to the contents of the location specified by DISP. SYMBOL is omitted. This command is used to relocate a direct pointer to an internal relocatable symbol.

Example: A: MOV #A,%0;

1 — INTERNAL DISPLACED RELOCATION

The displacement from the current location counter + 2 to the absolute address contained in the location pointed to by DISP is calculated and replaced. SYMBOL is omitted. This command is used only when there is a reference to an absolute location from a relocatable CSECT.

Example: CLR 177550

2 — EXTERNAL RELOCATION

The absolute address of the relocated external SYMBOL is added to the contents of the location designated by DISP and relpaced.

Example: MOV #A+10,%0

3 — EXTERNAL DISPLACED RELOCATION

The displacement from the location counter + 2 to the address which is the sum of the relocated base address of the external SYMBOL designated and the contents of the location pointed to by DISP is calculated and replaced.

Example: CLR A+6

APPENDIX D

LOAD MODULE FORMAT

COMD BLOCK

000	xxx	COMD block code
001	nnn	number of words of general information
002	mmm	number of monitor requests
aaa	aaa	lowest address loaded
ppp	ppp	program size in bytes (sum of all relocatable sections)
ttt	ttt	program transfer address. If even, the transfer occurs, if odd, the transfer does not occur.
ddd	ddd	DDT transfer address (0 if DDT is not loaded)
rrr	rrr	0=>absolute load module 1=>load-time relocatable module
sss	sss	load module name (2 words mod 40)
sss	sss	
rrr	rrr	Monitor routine requests are words containing
:	:	the routine number as specified in the monitor
:	:	library.
:	:	
:	:	
rrr	rrr	

DATA BLOCK

001	000	DATA block code
xxx	xxx	number of bytes in data block
yyy	yyy	load address of block
ddd	ddd	binary data
:	:	
:	:	
:	:	
ddd	ddd	
zzz		block checksum