ORGANISATION EUROPÉENNE POUR LA RECHERCHE NUCLÉAIRE
CERN EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

PL-11: A PROGRAMMING LANGUAGE FOR THE DEC PDP-11 COMPUTER

Robert D. Russell
Edited by T.C. Streater
Propriété littéraire et scientifique réservée pour tous les pays du monde. Ce document ne peut être reproduit ou traduit en tout ou en partie sans l'autorisation écrite du Directeur général du CERN, titulaire du droit d'auteur. Dans les cas appropriés, et s'il s'agit d'utiliser le document à des fins non commerciales, cette autorisation sera volontiers accordée.

Le CERN ne revendique pas la propriété des inventions brevetables et dessins ou modèles susceptibles de dépôt qui pourraient être décrits dans le présent document; ceux-ci peuvent être librement utilisés par les instituts de recherche, les industriels et autres intéressés. Cependant, le CERN se réserve le droit de s'opposer à toute revendication qu'un usager pourrait faire de la propriété scientifique ou industrielle de toute invention et tout dessin ou modèle décrits dans le présent document.

© Copyright CERN, Genève, 1974

Literary and scientific copyrights reserved in all countries of the world. This report, or any part of it, may no: be reprinted or translated without written permission of the copyright holder, the Director-General of CERN. However, permission will be freely granted for appropriate non-commercial use. If any patentable invention or registrable design is described in the report, CERN makes no claim to property rights in it but offers it for the free use of research institutions, manufacturers and others. CERN, however, may oppose any attempt by a user to claim any proprietary or patent rights in such inventions or designs as may be described in the present document.
PL-11: A PROGRAMMING LANGUAGE FOR THE DEC PDP-11 COMPUTER

Robert D. Russell *)

Edited by T.C. Streater

G E N E V A
1974

*) Now at Burroughs Corporation, 6300 Hollister Ave., Goleta, Calif. 93017, USA.
ABSTRACT

This manual gives the specification for a new programming language, PL-11, which is a high-level assembler for the DEC PDP-11 computer. The language has been modelled after PL-360. It is block structured, in a similar way to ALGOL-60, and gives access to all the PDP-11 hardware facilities, with the possibility of specifying for which model and hardware options the compiler will generate code. It also has facilities for CAMAC programming.

PL-11 serves as the standard programming language on PDP-11s in CERN's Omega Project, a multiparticle spectrometer whose data acquisition system consists of a number of PDP-11s on-line to a CII 10070.
PREFACE

The PL-11 compiler was written by Bob Russell, during the time he spent at CERN as a Fellow. The implementation of the compiler went through two major steps, each of which had a minor iteration. So, version 2 was the first definitive release of the compiler, and version 4 is that which is currently available. Bob wrote two notes which were published internally to the Omega Group, the first of which described version 2 as it then was, and the second of which described the changes, additions and improvements which went to make up version 4. Unfortunately, Bob did not have time, before he left CERN, to put these notes together to make a single paper. That task has fallen to the present editor, and the two notes mentioned above have been used as the basis of this document.

The PL-11 compiler currently executes as a cross-compiler on the CII 10070 used by the Omega Project. It is written in FORTRAN. Versions are available to run on other machines with FORTRAN capability, but they are not completely guaranteed. Requests for FORTRAN versions, accompanied by a 4' magnetic tape, should be sent to:

CERN Program Library,
DD Division,
CERN,
1211 Geneva 23,
Switzerland.

and should indicate in which density (9-track, 800 or 1600 bpi only) the tape should be written, and which version is required (IBM 360/370, CII 10070, CII IRIS 80, UNIVAC 1100 series, CDC 6000/7000 series, Honeywell 6000 series, XDS SIGMA 5/7). The tape will then be returned with (hopefully) sufficient documentation to allow installation of the compiler. A nominal fee will be charged to cover copying, packaging and postal charges. CERN accepts no responsibility for the correctness or otherwise of these versions, and will not undertake to update or maintain them, but would be grateful for any comments, corrections, or assembler versions of critical routines. These should be addressed to the above. All such changes would become the property of CERN. A note is available from the CERN Program Library describing its functions and giving additional details of the conditions of distribution of programs.

A version of PL-11 exists also for the PDP-11, itself written in PL-11; this runs either under the RSX-11D or DOS systems. Enquiries concerning this version should also be addressed to the above.
## CONTENTS

### SECTION I. INTRODUCTION TO PL-11

1. INTRODUCTORY DESCRIPTION  
   Page 1
2. BACKGROUND  
   Page 1
3. OBJECTIVES  
   Page 2
4. PHILOSOPHY  
   Page 2

### SECTION II. COMPILER CONTROL CARDS

1. INTRODUCTION  
   Page 4

2. INDIVIDUAL CONTROL CARDS
   
   2.1 TITLE  
   2.2 PAGE  
   2.3 STATS  
   2.4 NOGO  
   2.5 PL-11  
   2.6 CODE  
   2.7 NOCODE  
   2.8 INITIS  
   2.9 NOINITS  
   2.10 LIST  
   2.11 NOLIST  
   2.12 NAMES  
   2.13 NONAMES  
   2.14 DOS  
   2.15 EAU  
   2.16 NOEAU  
   2.17 FPP  
   2.18 NOFPP  
   2.19 MODEL20  
   
   Page 4
   Page 4
   Page 4
   Page 5
   Page 5
   Page 5
   Page 5
   Page 5
   Page 5
   Page 6
   Page 6
   Page 6
   Page 6
   Page 6
   Page 6
   Page 7
   Page 7
### SECTION III. BASIC COMPONENTS OF PL-11

#### 1. LITERALS AND CONSTANTS
- 1.1 Decimal literals and constants
- 1.2 Hexadecimal literals and constants
- 1.3 Octal literals and constants
- 1.4 REAL constants
- 1.5 DOUBLE constants
- 1.6 String literals and constants
- 1.7 Address literals and constants
- 1.8 Length literals and constants

#### 2. RESERVED WORDS

#### 3. SYMBOLIC NAMES
- 3.1 Introduction
- 3.2 Identifiers
- 3.3 Pre-declared identifiers
- 3.4 Labels
- 3.5 Equates

#### 4. SOURCE PROGRAM FORMAT
- 4.1 Cards
- 4.2 Spacing
- 4.3 Comments
- 4.4 The block
- 4.5 Modules

### SECTION IV. DECLARATIONS IN PL-11

#### 1. VARIABLE DECLARATIONS
- 1.1 Simple variable declarations
- 1.2 Tabular variable declarations
- 1.3 Limitation on array allocation size
2. SYNONYM DECLARATIONS
   2.1 Variable synonyms
   2.2 Register synonyms
   2.3 BIT synonyms

3. EQUATE DECLARATIONS

4. INITIALIZATION
   4.1 Simple initialization
   4.2 Initialization lists
   4.3 Repetition in initialization lists
   4.4 String constants as initial values
   4.5 Initialization of REAL and DOUBLE variables
   4.6 Initializing stacks
   4.7 Self-defining arrays and stacks

5. PROCEDURE DECLARATIONS

SECTION V. EXPRESSIONS IN PL-11

1. OPERANDS
   1.1 Normal references
   1.2 Subscripted references
   1.3 Stack references
   1.4 Indirect references

2. OPERATORS
   2.1 Monadic operators
   2.2 Dyadic operators
   2.3 Extended dyadic operators for the model 20 EAU option
   2.4 Extended dyadic operators for the model 40 and model 45
   2.5 Bit operators

3. EXPRESSION EVALUATION

4. THE CONCEPT OF "TYPE" IN EXPRESSIONS
   4.1 INTEGER and LOGICAL types
   4.2 BYTE type
   4.3 BYTE registers
   4.4 Mixed types in expressions

5. LITERAL EXPRESSIONS
6. ADDRESS LITERALS IN EXPRESSIONS

7. EXPRESSIONS WITH NO OPERATIONS

8. USE OF CERTAIN PRE-DECLARED NAMES
   8.1 "Absolute" addressing
   8.2 "Pointer register" addressing

SECTION VI. CONDITIONS IN PL-11

1. SIMPLE CONDITIONS
   1.1 Comparisons
   1.2 Bit test conditions

2. COMPOUND CONDITIONS
   2.1 AND-compounds
   2.2 OR-compounds

SECTION VII. CONTROL STATEMENTS IN PL-11

1. 'IF' STATEMENT
   1.1 The incomplete IF statement
   1.2 The complete IF statement
   1.3 Nesting IF statements

2. 'GOTO' STATEMENT
   2.1 Basic form
   2.2 'IF' statements combined with 'GOTO' statements

3. 'CASE' STATEMENTS

SECTION VIII. ITERATION IN PL-11

1. THE 'FOR' STATEMENT
   1.1 Increasing loop-counter
   1.2 Decreasing loop-counter

2. THE 'WHILE' STATEMENT
3. THE 'FOR-WHILE' STATEMENT
   3.1 Increasing loop-counter
   3.2 Decreasing loop-counter

4. "INFINITE-LOOP" 'DO' STATEMENT

5. DEGENERATE LOOP FORMS
   5.1 Loops without a STEP value
   5.2 Loops without an initial value
   5.3 Loops without initial or STEP values
   5.4 Loops without a limit value
   5.5 Loops without a body
   5.6 Miscellaneous combinations

6. GYPSY'S WARNING

SECTION IX. PROCEDURES IN PL-11

1. LOCAL PROCEDURES
   1.1 General
   1.2 Scope
   1.3 Nesting
   1.4 Recursion
   1.5 Returns
   1.6 Parameter passing

2. EXTERNAL AND GLOBAL PROCEDURES
   2.1 Definition
   2.2 Utilization
   2.3 Scope

3. 'TRAP' AND 'EMT' PROCEDURES

4. INTERRUPT PROCEDURES
   4.1 Introduction
   4.2 General interrupt handling
   4.3 CAMAC interrupt handling
   4.4 EXTERNAL interrupt handling
   4.5 GLOBAL and EXTERNAL linkage to interrupt procedures

5. "HARDWARE" PROCEDURES

6. STACK PROCEDURES ON THE MODEL 45 USING THE 'MARK' INSTRUCTION
SECTION X. FLOATING POINT IN PL-11

1. INTRODUCTION

2. PRE-DECLARED REGISTERS

3. FLOATING-POINT OPERATIONS ON THE MODEL 45
   3.1 Tutorial
   3.2 Precision modes
   3.3 REAL and DOUBLE operands
   3.4 Move operations
   3.5 Other operations

4. FLOATING-POINT OPERATIONS ON THE MODEL 40

SECTION XI. THE PL-11 'COMMON' FACILITY

1. INTRODUCTION

2. INITIALIZATION

3. SCOPE AND BLOCK STRUCTURE
   3.1 Description
   3.2 Deck set-up

4. USEFUL DETAILS

5. IMPLEMENTATION

SECTION XII. MEMORY MANAGEMENT INSTRUCTIONS

SECTION XIII. 'CAMAC' PROGRAMMING IN PL-11

1. DESIGN PHILOSOPHY

2. DECLARING 'CAMAC' VARIABLES

3. USING 'CAMAC' VARIABLES

4. 'CAMAC' ARRAYS AND BLOCK TRANSFERS
5. OTHER 'CAMAC' FUNCTIONS

6. CONCLUSION

APPENDIX A: PL-11 ERROR MESSAGES

APPENDIX B: RESERVED WORDS IN PL-11

APPENDIX C: PRE-DECLARED VARIABLES AND REGISTERS
1. REGISTERS
2. MEMORY ADDRESSED BY REGISTERS
3. ABSOLUTE MEMORY
4. EXTENDED ARITHMETIC UNIT 'REGISTERS'
5. CONDITION CODE BIT NAMES
6. FLOATING-POINT REGISTERS FOR THE MODEL 45

APPENDIX D: PRE-DECLARED 'EMT' PROCEDURES IN PL-11
1. I/O Routines
   1.1 READ
   1.2 PRINT
   1.3 TELETIME
2. DATA CONVERSION Routines
   2.1 BINTODEC
   2.2 BINTOOCCT
   2.3 DECTOBIN
   2.4 OCTTOBIN

APPENDIX E: PROGRAMMING EXAMPLES

APPENDIX F: 63 ASCII CHARACTERS RECOGNIZED IN PL-11 STRINGS
APPENDIX G: COMPILER OPTIMIZATIONS

1. 'GOTO' STATEMENTS
2. COMPARISON STATEMENTS
3. 'CASE' STATEMENTS
4. 'FOR' STATEMENTS
5. PROCEDURE 'RETURNS'

APPENDIX H: DIFFERENCES BETWEEN DIFFERENT PDP-11 MODELS
1. INTRODUCTORY DESCRIPTION

PL-11 is a language for programming the DEC PDP-11 computer, designed and implemented by the author as part of the Omega Project at CERN. It belongs to the class of computer languages which has been variously classified as "structured assembly", "high-level assembler", "intermediate language", or "low-level language". It is modelled after PL-560, the first language in this class, and is intended to fill the gap between high-level languages, such as FORTRAN and ALGOL, which are easy to learn and program but are inefficient and cumbersome for small computers and "systems programming", and the basic symbolic assemblers, which are very difficult to learn and program, primarily because of their awkward notation, but are essentially "as efficient as possible", because of their closeness to the basic machine hardware. PL-11 is designed for convenient programming in an ALGOL-like style and notation, but with minimal sacrifice of the efficiency possible in basic assembler code. It is a tool for utilization by both the efficiency-conscious systems programmer and the applications programmer who wishes simply to get a job done. In both cases the language design encourages a clear, concise programming style in which programs are easy to read and modify, and the program listing itself can serve as a convenient form of documentation for the algorithms being programmed.

2. BACKGROUND

In 1968, Wirth published a paper [JACM #15 (1968) p.37] describing PL-560 -- "A Programming Language for the IBM 360". As originally designed, this language was intended primarily as a tool in which to write an ALGOL compiler for the IBM 360 computer, since the standard IBM assembler was virtually unusable, owing to an extremely slow assembly speed, an awkward, obscure notation, and the difficulty of dealing directly with the organization of the IBM 360 addressing mechanism and instruction repertoire. It was soon recognized, however, that this language was exactly the tool that had long been missing in systems work, as well as all other applications previously requiring assembler coding, because of its convenience to the programmer, its compilation efficiency, and the efficiency of the code produced. To illustrate, the compiler (which is itself written in PL-560) will recompile itself on an IBM 360/67 using OS at the rate of 6000 cards per minute (spooled I/O), and on an IBM 360/30 using OS at the rate of 1200 cards per minute (i.e. the speed of the on-line card reader).

Although originally designed for stand-alone use of the IBM 360, PL-560 was quickly modified to operate under the standard IBM operating systems, and since then has been further modified to operate in a time-sharing environment.

Following the precedent set by PL-560, several other languages have been designed with the same objectives. PL-516 was developed by Bell and Wichmann [Software, Vol. 1 (1971), pp. 61-72] for the Honeywell DDP-516 computer at the National Physical Laboratory, England. Although considerably more "primitive" in its basic features than PL-560 (largely because of the primitiveness of the DDP-516 hardware), the authors conclude: "A high-level assembly language for a small computer has been shown to be entirely feasible and to provide an easy method of exploiting the hardware without becoming entangled in machine code" (p. 71). They also state: "We would therefore recommend the use of ALGOL-like assembly languages for computers having 8K 16-bit words...", which is particularly relevant to the 8K 16-bit PDP-11.

In Munich, Germany, a group at the Technische Hochschule has developed a language called PS-440 for their Telefunken TR-4 and TR-440 computers. This language includes features absent in PL-560, and notationally is modelled after ALGOL 68 (PL-560 notation is similar to ALGOL 60). This was designed as a tool with which to write operating systems, and has proved extremely useful in that as well as other areas.
A lesser known development is the languages LP-20 and LP-70, developed in France for the CII 10020 and CII 10070 computers, respectively. These are very similar syntactically to PL-360, but do not seem to be widely known outside of France.

3. OBJECTIVES

PL-11 is the result of a design and implementation effort to fulfil the following objectives (not necessarily in order of importance).

a) The syntax of the language should be easy to learn, to read, and to use. In particular, the notation should be similar to FORTRAN or ALGOL, and should completely avoid the awkward, unnatural notation of assembly language.

b) The language should permit full use of the facilities provided by the PDP-11 hardware. In particular, all registers, storage cells, and machine operations should be accessible to the programmer's control.

c) The code generated by the compiler should be predictable from the language constructs. In particular, nothing should be "hidden" from the programmer, so that it should always be possible to determine from the PL-11 language specifications exactly what code is produced by each language construct.

d) The code generated by the compiler should be efficient. To accomplish this in a manner consistent with points (b) and (c) requires that the language be designed to reflect accurately the capabilities of the hardware, so that each language construct would generate a minimal number of instructions. Optimization, except in a few special, well-defined cases, is then the task of the programmer, not the compiler.

e) The language should allow the programmer full use of the facilities provided by the PDP-11 software environment. In particular, communication with any monitor, and with routines written in assembler language, should be easy and straightforward.

f) The compiler should be written in FORTRAN for the CII 10070 computer. As much as possible, use of the special features of the CII FORTRAN should be avoided so that the compiler could be transported to other machines with FORTRAN capability.

g) The compiler should generate the code in the form of relocatable object modules acceptable to the CII SIRIS7 operating system. This would enable the programmer to make full use of the disk storage, library, and linkage editing facilities of the CII system, and would also permit PL-11 programs to be easily combined with programs produced by A. Jeavons' PAL-11 assembler that runs on the CII 10070, thus satisfying point (e) above.

h) The compiler should be reasonably efficient, and not require inordinate amounts of time or space on the CII 10070.

i) The language should be well documented, so that non-professional programmers can learn and program in PL-11 without a knowledge of the DEC PAL-11 assembly language, and with only a rudimentary knowledge of the PDP-11 hardware.

4. PHILOSOPHY

As in any design effort, the product will reflect to a large extent the author's philosophy and experience. PL-11 shares the same basic philosophy as PL-360. This philosophy can be summarized as follows:

a) The language should do everything possible to encourage a clear, comprehensible programming style. As far as possible the petty details of assembly language notation, the fixed format restrictions of FORTRAN and assembly language, and FORTRAN's artificial restrictions on identifier length and first letter should be abolished.

b) The rules for writing syntactically correct language constructs should be as simple and general as possible. Exceptions and special cases should be minimal, if not nonexistent.
c) The language should include a rich set of structured facilities, so that GOTOs and labels can be virtually eliminated. This improves enormously the readability and understanding of a program, especially for those who must use or modify a program they did not write.

d) The language should not permit automatic defaults, such as are found in FORTRAN and abound in PL/I. Missing items serve to decrease the program's readability and increase the burden on the programmer to remember annoying details. (They are usually remembered incorrectly, or in a confused and muddled state that requires constant recourse to the language manual to determine "what happens when"). If a language is clear and concise, there is no need for someone using it to "leave things out". Invariably the "things" that get left out are poorly designed or ill-conceived in the first place.

e) A program is written to operate on data, not on itself. Therefore programs should not be self-modifying, since this often involves programming "tricks" which take longer to debug (as well as making the debugging more difficult), and which make the resultant program totally incomprehensible to anyone but the author (and sometimes even to the author after a few weeks). In addition, self-modifying programs can be neither re-entrant nor recursive. Ideally the machine hardware would provide a mechanism to protect program code from all forms of access except instruction execution, but this only exists on the bigger PDP-11s. PL-11, however, contains no language components with which a programmer can explicitly modify the code generated.

f) The language should include a convenient method of allowing and encouraging local variables and local procedures, so that their precise area of interest within a program can easily be ascertained. For this reason, PL-11 is a block-structured language similar to ALGOL 60.

g) The compiler should produce meaningful diagnostic error messages. The cryptic, single-letter "flags" generated by most assemblers are totally useless. Although not always possible, the compiler should pin-point the exact cause of the error in such a way that its correction should be obvious to the programmer.

h) The keywords of the language syntax should also be reserved words. This means that each time the word appears in a program, it will always have the same precise meaning. This not only avoids the complicated syntactic analysis needed to find keywords from context (such as is needed in FORTRAN), but eliminates the obscure "tricks" that only serve to confuse the programming style in languages like FORTRAN or PL/I, where keywords are not reserved (for example, using a variable named IF or DO in FORTRAN). The number of keywords, however, should be kept to a minimum (unlike COBOL).

i) The language syntax should be expressible in a precise notation, such as Backus-Naur Form. This tends to reduce errors caused by "misinterpretation" of imprecise English explanations, and also defines exactly what is and is not allowed in a manner that is impossible in a natural language explanation.
Section II. COMPILER CONTROL CARDS

1. INTRODUCTION

Compiler control cards enable the programmer to choose between various compilation options and they are directly analogous to the "assembler directives" found in most assemblers. Strictly speaking, these cards are not part of the PL-ll language itself since they only convey information to the compiler, and therefore they have a fixed format that is considerably different from normal PL-ll statements.

Each control card consists of a single 80-column card with the following fixed format: a control character punched in column 1 and a control command word punched in columns 2-8. Except for the "title" and "include" commands, columns 9-80 of the card are ignored. It is not possible to have more than one command per card. Furthermore, control cards are not printed on the program listing, although they may influence the listing in an obvious way (by starting a new page, by listing the generated code, etc.). Each control card becomes effective when it appears in the program deck and remains in effect until the end of the deck or until cancelled by another control card.

The control character punched in column 1 is a 12-8-2 punch, which on the CERN-ANSI keypunches is a "left bracket" symbol (]). On the unmodified IBM 029 keypunch this hole combination is a "cent sign" (¢). Neither of these symbols is printable on the CII line printer, but the left bracket is a standard PDP-11 teletype character.

At the end of each compilation, the compiler prints out a list of all the options which were in effect at the end of the module. Users should check this list to ensure that they have in fact used the options they desired, since the control cards are not printed anywhere else. In particular, one should check that code has been compiled for the proper model PDP-11 and CAMAC interface.

2. INDIVIDUAL CONTROL CARDS

2.1 [TITLE]

This control command causes a new page on the program listing and, in addition, columns 9-72 of this card will be printed as the heading on that new page and all following pages of the listing until another title card is encountered. Columns 9-72 can contain any legal EBCDIC characters, since they are printed exactly as punched and are otherwise ignored by the compiler. All pages on the listing before the first title card (if any) will have a blank heading. Any number of these commands can appear anywhere in the deck, and each is effective at the place it occurs.

2.2 [PAGE]

This command causes a new page on the program listing with the same heading as on the previous page. Any number of these commands can appear anywhere in the deck, and each is effective at the place it occurs.

2.3 [STATS]

This command causes the compiler to print statistics on the compilation process. This includes the number of times each subroutine in the compiler has been called, the size of every table, the length and frequency of all table searches (name table, reserved words,
grammar rules), and a few other miscellaneous statistics. This option is of interest primarily as a means of "fine tuning" the compiler itself. It can appear anywhere in the deck, but the statistics are always gathered for the entire compilation process. Without this card no statistics are printed.

2.4 [NOGO]

This command prevents the compiler from producing a relocatable object module that is normally used as input to the linkage editor. This command can appear anywhere in the deck. Without this card a relocatable object module is produced if no errors are detected.

2.5 [PL-11]

This command permits the programmer to batch-compile several PL-11 modules in a single job step. It operates identically to the CII job step control card PL-11, but without the overhead of returning to the operating system and then calling the compiler back again. Modules separated by this command are compiled completely independently of one another. Therefore all compiler control options are reset by this command to their original default values before the following module is read and compiled. This implies that each module must have its own set of compiler control cards, whether or not the options are to be the same in successive modules. The appearance of this command is identical with the "end-of-deck" for the module which preceded it.

2.6 [CODE]

Print the instruction words generated by the compiler and their relative addresses. Both are printed as 6-digit octal values on the right side of the program listing page. The code generated by a PL-11 statement is printed after the card containing the PL-11 statement is listed. Any number of these commands can appear anywhere in the deck and each remains effective until the end of deck or the occurrence of a NOCODE command.

2.7 [NOCODE]

Stop printing the code generated by the compiler. This command cancels the effect of a CODE command at the place it appears in the deck. Through the use of these two commands the code generated in a local area of the program can be listed without listing the code for the entire program. This command is the normal default option in effect at the start of a module.

2.8 [INITS]

Print the initial values of variable storage for all variables declared after the appearance of this command until the end of deck or the NOINITS command is encountered. The initial values and their relative addresses are printed as 6-digit octal values on the right side of the page after the table of declared names has been completely printed. Any number of these commands can appear anywhere in the deck.

2.9 [NOINITS]

Stop printing the initial values of variable storage. This command cancels the effect of an INITS command at the place where it appears in the deck. This command is the normal default option in effect at the start of a module.

2.10 [LIST]

Print the PL-11 source cards exactly as they are read from the input deck. This is the normal default option in effect at the start of a module. This command remains in effect until the end of deck or a NOLIST command is encountered.

2.11 [NOLIST]

Stop printing the PL-11 source cards. This command cancels the effect of a LIST command at the place it appears in the deck. Alternate use of these two commands permits the programmer to list parts of his source program without listing all of it. However, source
cards containing errors are always listed, together with the error message, regardless of the NOLIST option. The NOLIST option also stops the printing of the generated code, regardless of the CODE option. Thus the CODE/NOCODE options are ignored during NOLIST, and are effective only when LIST is in effect.

2.12 [NAMES]

Print a table of the name and attributes of each identifier as it is declared. This is the normal default option in effect at the start of a module. This command remains in effect until the end of deck or a NONAMES command is encountered.

2.13 [NONAMES]

Stop printing each identifier as it is declared. This command cancels the effect of a NAMES command at the place it appears in the deck. The table of identifiers printed before the program listing will not contain any identifiers that are declared while NONAMES is in effect.

2.14 [DOS]

The relocatable object module produced by the compiler is to be produced in the PDP-11/DOS (Disk Operating System) format. This output is written as a sequence of 80-column EBCDIC card images on FORTRAN file 200. This file is then used as input to the processor PDP/DOS in order to obtain a DOS format paper tape. (Alternatively, file 200 can be punched directly on cards and read on the PDP-11.) This tape can then be loaded into the PDP-11 with either the stand-alone LINK-11/S or the linkage editor in DOS. If this card is not present, the object module will be produced in the CII 10070 linkage editor format, and will be written onto the GO file on disk rather than on file 200. No object module in any format is produced if there is a compilation error or if the NOOO command is in effect at the end of the compilation. The DOS command can appear anywhere in the deck.

2.15 [EAU]

This control command indicates to the compiler that the PDP-11 on which this compiled program will be run has an Extended Arithmetic Unit. Therefore the compiler will generate code that utilizes this unit for the following PL-11 operations:

```
  *  MULT
  /  DIV
  REM
  SHA  SHIFTA
  SHL  SHIFTL
```

This option is valid only for the models 20 and 40, and is ignored on the model 45. It is the default option for the model 20. This command should appear in the deck before the use of any of the above operators, since it becomes effective at the place it appears.

2.16 [NOEAU]

This command indicates to the compiler that the PDP-11 on which this compiled program will be run does not have an Extended Arithmetic Unit. Therefore the PL-11 operators listed above will cause error messages on the model 20. On the models 40 and 45, the SHL and SHIFTL operators will cause errors, while all the other operators listed above will generate code for the internal hardware instructions as described in Section V. This option is the default for the models 40 and 45. This command should appear in the deck before the use of any of the operators listed above, since it becomes effective at the place it appears.

2.17 [FPP]

This command indicates to the compiler that the PDP-11 on which this compiled program will be run has a Floating-Point Processor. Therefore all floating-point operations are accepted by the compiler. This command should appear in the deck before the use of any floating-point operations or declaration of any floating-point variables, since it becomes effective at the place it appears. This is the default option on the model 45.
2.18 [NOFPP]

This command indicates to the compiler that the PDP-11 on which this compiled program will be run does not have a Floating-Point Processor. Therefore all floating-point operations will cause errors. This is the default option for models 20 and 40.

2.19 [MODEL20]

This command indicates to the compiler that the machine on which this compiled program will be run is a PDP-11 model 20. Therefore the following PL-11 operations are illegal:

- MASKFLIP
- LONGSHIFT
- ROTATE

In addition, all other extensions for the models 40 and 45 will not be accepted in the source program (Section V.2.4). This option implies EAU and NOFPP, and is the default option at the start of a module. This command should appear at the beginning of the deck.

Note that [MODEL05], [MODEL10] and [MODEL15] cards are also accepted, but they are treated identically to [MODEL20] within the compiler.

2.20 [MODEL40]

This command indicates to the compiler that the machine on which this compiled program will be run is a PDP-11 model 40. Therefore the compiler will generate code for the extended instruction set of this machine (see Section V.2.4). This option implies NOEAU and NOFPP. This command should appear at the beginning of the deck.

2.21 [MODEL45]

This command indicates to the compiler that the machine on which this compiled program will be run is a PDP-11 model 45. Therefore the compiler will generate code for the extended instruction set of this machine (see Section V.2.4). This option implies NOEAU and FPP. This command should appear at the beginning of the deck.

2.22 [CALL]

This command indicates to the compiler that the machine on which the compiled program will be run has a CALL CAMAC interface. Therefore the compiler will generate code for all CAMAC declarations and instructions to use this interface. This command should appear in the deck before the use of any CAMAC declarations or instructions, since it becomes effective at the place it appears. This is the default option for the CAMAC interface.

2.23 [CCI1]

This command indicates to the compiler that the machine on which the compiled program will be run has a CCI1 CAMAC interface. Therefore the compiler will generate code for all CAMAC declarations and instructions to use this interface. This command should appear in the deck before the use of any CAMAC declarations or instructions, since it becomes effective at the place it appears.

2.24 [INCLUDE]

This command causes the inclusion, at its point of occurrence, of the card images stored on the logical unit specified on the [INCLUDE] card. The format is

```
[INCLUDE nn]
```

where nn is the logical unit number. No blanks may appear on the card before the logical unit number. This card is used to bring in blocks of PL-11 source stored on logical unit nn. It is useful for predefining tables and buffer layouts, etc. Thus, if several people are working on separate parts of a large program, they may avoid the problems that arise owing to mismatch of variable definitions in their separate program sections. Having included the cards on the unit nn, up to the end-of-file, the compiler then starts to compile what it has just read, exactly as if the INCLUDE statement were replaced by the cards on the file.
### 3. CONTROL CARD SUMMARY

<table>
<thead>
<tr>
<th>Command(s)</th>
<th>Location in deck</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAMAC</td>
<td>beginning</td>
<td>CAMAC interface is CAMAC</td>
</tr>
<tr>
<td>NOCAMAC</td>
<td>beginning</td>
<td>CAMAC interface is NOCAMAC</td>
</tr>
<tr>
<td>CODE (NOCODE)</td>
<td>anywhere</td>
<td>Controls printing of generated instructions</td>
</tr>
<tr>
<td>DOS</td>
<td>anywhere</td>
<td>Produce object module in DOS format</td>
</tr>
<tr>
<td>EAU (NOEAU)</td>
<td>beginning</td>
<td>Run-time PDP-11 has (no) EAU</td>
</tr>
<tr>
<td>FPP (NOFPP)</td>
<td>beginning</td>
<td>Run-time PDP-11 has (no) FPP</td>
</tr>
<tr>
<td>GO (NOGO)</td>
<td>anywhere</td>
<td>Controls production of object module</td>
</tr>
<tr>
<td>INCLUDE</td>
<td>anywhere</td>
<td>Brings in predefined blocks of PL-11 source</td>
</tr>
<tr>
<td>INITS (NOINITS)</td>
<td>anywhere</td>
<td>Controls printing of initialized data storage</td>
</tr>
<tr>
<td>LIST (NOLIST)</td>
<td>anywhere</td>
<td>Controls printing of source cards</td>
</tr>
<tr>
<td>MODEL20</td>
<td>beginning</td>
<td>Run-time PDP-11 is a model 20</td>
</tr>
<tr>
<td>MODEL40</td>
<td>beginning</td>
<td>Run-time PDP-11 is a model 40</td>
</tr>
<tr>
<td>MODEL45</td>
<td>anywhere</td>
<td>Run-time PDP-11 is a model 45</td>
</tr>
<tr>
<td>NAMES (NONAMES)</td>
<td>anywhere</td>
<td>Controls printing of identifier table</td>
</tr>
<tr>
<td>PAGE</td>
<td>anywhere</td>
<td>Starts new page on listing</td>
</tr>
<tr>
<td>PL-11</td>
<td>between modules</td>
<td>For batch compilations</td>
</tr>
<tr>
<td>STATS (NOSTATS)</td>
<td>anywhere</td>
<td>Controls printing of compiler statistics</td>
</tr>
<tr>
<td>TITLE</td>
<td>anywhere</td>
<td>Starts new page with new heading</td>
</tr>
</tbody>
</table>
Section III. BASIC COMPONENTS OF PL-11

1. LITERALS AND CONSTANTS

There are eight kinds of literals and constants in PL-11. These are symbols whose value is defined at compile-time, or, in the case of address constants, at link-edit time, and this value is fixed throughout the entire execution of the program.

It is important that the reader be aware of the difference between literals and constants. A literal is a symbol which may be used either as an immediate value, or in EQUATE declarations (see Section IV.3). It does not therefore occupy variable core-storage. Also, a literal must be representable in 16 bits or less, in order that it may be used as an immediate value. A constant is a symbol used for the initialization of variables. Generally, each of the eight kinds may be used as a constant, but they may not all be used as literals. The differences are noted in each section below.

1.1 Decimal literals and constants

Decimal literals and constants are sequences of one or more decimal digits (0,1,2,3,4, 5,6,7,8,9), preceded by an optional minus sign. Their value is the binary equivalent of the decimal number represented by the digit sequence, or the two's complement of this value if a minus sign precedes the number. Because the PDP-11 word size is 16 bits (15 bits plus sign), the range of legal decimal literals and constants is [-32768, 32767], corresponding to [-2^{15}, 2^{15} -1].

<table>
<thead>
<tr>
<th>Examples</th>
<th>Legal</th>
<th>Illegal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>+1</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0.0</td>
</tr>
<tr>
<td>-32768</td>
<td></td>
<td>32768</td>
</tr>
<tr>
<td>-2</td>
<td></td>
<td>-32769</td>
</tr>
<tr>
<td>-4159</td>
<td></td>
<td>+459.1</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td>500,000</td>
</tr>
<tr>
<td>32767</td>
<td></td>
<td>+32767</td>
</tr>
<tr>
<td>-0</td>
<td></td>
<td>9,000</td>
</tr>
</tbody>
</table>

1.2 Hexadecimal literals and constants

A hexadecimal literal or constant is a sequence of one or more hexadecimal digits (0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F), preceded by a required hash sign (#) (used to indicate "hex" to the compiler). The value of this is the bit pattern represented by the digit sequence (four bits per digit). Because the PDP-11 word size is 16 bits, a maximum of four hexadecimal digits per literal or constant is allowed (including leading zeros). If fewer than four are specified, leading zeros will be supplied (i.e. the digits are right-justified in the PDP word).
1.3 Octal literals and constants

An octal literal or constant is a sequence of one or more octal digits (0,1,2,3,4,5,6,7), preceded by a required dollar sign ($) (used to indicate 'octal' to the compiler). The value of this is the bit pattern represented by the digit sequence (three bits per digit). Because the PDP-11 word size is 16 bits, an octal literal or constant may have up to a maximum of six digits (including leading zeros), and if there are six digits, the first must be 0 or 1 (this is due to the fact that six full octal digits represent 18 bits; since only 16 bits are available, the leading octal digit must be restricted). If fewer than six are specified, leading zeros are assumed (i.e. the digits are right-justified in the PDP word).

Examples:

<table>
<thead>
<tr>
<th>Legal</th>
<th>Illegal</th>
</tr>
</thead>
<tbody>
<tr>
<td>#$A1</td>
<td>-#A1</td>
</tr>
<tr>
<td>#$ABCD</td>
<td>+#ABCD</td>
</tr>
<tr>
<td>#$F103</td>
<td>#$F.1</td>
</tr>
<tr>
<td>#$A000</td>
<td>#$A0000</td>
</tr>
<tr>
<td>#$000B</td>
<td>#$000B</td>
</tr>
<tr>
<td>#$FE0C</td>
<td>#$GEIJ</td>
</tr>
</tbody>
</table>

1.4 REAL constants

A REAL constant is a sequence of one or more decimal digits (0,1,2,3,4,5,6,7,8,9), preceded by an optional minus sign, and either including a decimal point, or being followed by an exponent indicator, or both. The exponent indicator consists of the required letter E followed by a sequence of decimal digits (0,1,2,3,4,5,6,7,8,9). The sequence may be preceded by an optional plus or minus sign. The exponent value specified in the second decimal digit sequence indicates the power of ten by which the first sequence is to be multiplied. Thus the following are all legal ways of representing π to five decimal places:

- 3.14159
- 3.14159.E-0005
- .000314159E+0004

REAL constants must lie approximately in the range ±1.0E±39. The following are acceptable REAL constants in PL-11:

- 2.0
- 20E-1
- 0.5
- -473.15
- 9.1E4
- 257E (≈257.0)
- .0
- 119.E4
- -.00001

Note that each constant contains either a decimal point (.) or an exponent indicator (E) or both, and that the constant has no embedded blanks. The following are not acceptable REAL constants:
20  This is an integer constant
9+4  This is an integer constant expression, value = 13
1.45*7  Exponents must begin with E
2.4E1.5  Exponents must be integers
  E4  This is the variable named E4
  .E  There must be at least one digit in the mantissa.

REAL constants cannot be used in constant expressions (i.e. expressions that are evaluated at compile-time), and cannot be used as literals.

1.5  DOUBLE constants

A DOUBLE constant is a sequence of one or more decimal digits (0,1,2,3,4,5,6,7,8,9), preceded by an optional minus sign, including an optional decimal point, and followed by a required exponent indicator. The exponent indicator consists of the required letter D followed by a sequence of decimal digits (0,1,2,3,4,5,6,7,8,9). The sequence may be preceded by an optional plus or minus sign. The exponent value specified by the second decimal digit sequence indicates the power of ten by which the first sequence is to be multiplied.

DOUBLE constants must lie approximately in the range ±1.0D±39.

The following are acceptable DOUBLE constants in PL-11:

2.0D  20D-1
0.516D
3.1415926536D  10D5
-9D  257D+3

Note that all DOUBLE constants must contain the exponent indicator (D). PL-11 does no automatic mode change from REAL if there are a large number of significant digits, as some FORTRAN compilers do.

DOUBLE constants cannot be used in constant expressions (i.e. expressions that are evaluated at compile-time) and cannot be used as literals.

1.6  String literals and constants

A string is a sequence of one or more characters preceded and terminated by a required quote mark (apostrophe) ('). Strings of more than two characters may only be used as constants; that is, to initialize arrays and stacks. Strings of one or two characters may be used as literals or constants. The apostrophe character (') is represented within a string by two successive apostrophes.

There are three types of string, as detailed below:

i)  EBCDIC strings

If a string is preceded by a hash sign (#), each character in the string will be translated into internal 8-bit EBCDIC codes.

ii)  Rad50 strings

If a string is preceded by a dollar sign ($), it will be translated into a "radix 50" string, as defined by DEC. This type of format is heavily used in the PDP-11 systems, since it allows three characters to be stored in one 16-bit word. Only the characters A-Z, 0-9, $, and . have unique rad50 codes. PL-11 will also translate a question mark (?) into the rad50 code for dollar sign ($), and an at-sign (@) into the rad50 code for period (.). All other characters are translated into blanks. Rad50 strings always require an even number of bytes, and strings are padded with trailing blanks if necessary to ensure this. See the DEC systems manual for more details on this format.
iii) ASCII strings

Strings that are not preceded by any character are translated into 7-bit internal ASCII codes, as are used by DEC in the PDP-11.

The maximum length string is 256 characters. When used in word operations, a single character will be right-justified with leading zeros. The following are legal examples:

<table>
<thead>
<tr>
<th>String</th>
<th>String value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>'AB'</td>
<td>AB</td>
<td>ASCII</td>
</tr>
<tr>
<td>'#  '</td>
<td>'</td>
<td>EBCDIC</td>
</tr>
<tr>
<td>&quot;$A'B'</td>
<td>A'B</td>
<td>RAB50</td>
</tr>
<tr>
<td>'......'</td>
<td>'</td>
<td>ASCII</td>
</tr>
<tr>
<td>'01'</td>
<td>01</td>
<td>ASCII</td>
</tr>
<tr>
<td>'#ABC01'</td>
<td>ABC01</td>
<td>EBCDIC</td>
</tr>
</tbody>
</table>

The following are illegal examples:

- AB'
- ''
- 'AC'
- ''
- '-01,'
- *'AB'

1.7 Address literals and constants

An address literal or constant is the name of a variable or procedure enclosed in parentheses and preceded by the reserved word REF. The value of this is the absolute binary address where the named variable is located during the program execution. Since this value is fixed at load time, names used in address literals or constants cannot be registers or contain register references.

Examples:

<table>
<thead>
<tr>
<th>Legal</th>
<th>Illegal</th>
</tr>
</thead>
<tbody>
<tr>
<td>REF(X)</td>
<td>REF(+X)</td>
</tr>
<tr>
<td>REF(ABC)</td>
<td>REF(-ABC)</td>
</tr>
<tr>
<td>REF(Q(5))</td>
<td>REF(Q(R5));</td>
</tr>
<tr>
<td>REF(Q(5+10))</td>
<td>REF(Q(R0+5))</td>
</tr>
<tr>
<td>REF(A(1))</td>
<td>REF(1)</td>
</tr>
<tr>
<td>REF(B(-92))</td>
<td>REF(-B(92));</td>
</tr>
<tr>
<td>REF(X(#F))</td>
<td>REF X(#F)</td>
</tr>
<tr>
<td>REF(P)</td>
<td>REF(PUSH(R5))</td>
</tr>
</tbody>
</table>

If ABC is a procedure, then REF(ABC) will be the absolute memory address of the first instruction of that procedure. If the procedure has been declared as an EMT or TRAP procedure, its address is the EMT or TRAP number (range 0-255).

1.8 Length literals and constants

A length literal or constant is the name of a variable enclosed in parentheses and preceded by the reserved word LENGTH. The value of this is the number of bytes of storage which have been reserved for that variable. For examples of its use, see Section IV.4.7.
2. RESERVED WORDS

A reserved word is a sequence of letters and/or digits whose meaning is defined as part of the PL-11 syntax. They can therefore be used only as defined by the rules of the language, and in no other way. In particular, a reserved word cannot be used as a symbolic name. A list of reserved words is given in Appendix B. Their use and meaning is explained in the rules of the grammar. Note that no embedded blanks or other symbols may be written as part of a reserved word.

3. SYMBOLIC NAMES

3.1 Introduction

A symbolic name is defined in PL-11 as a sequence of one or more letters (A-Z) and/or digits (0-9), starting with a letter. Names can be of any length -- there is no artificial limitation imposed on the number of characters allowed in a name, provided the name contains no embedded blanks, and no characters other than letters and digits. Note, however, that only the first 12 characters of any symbolic name serve to make that name unique. Remaining characters are accepted but are not remembered by the compiler. Thus

```
ABCDEFHJKL123
```

and

```
ABCDEFHJKL456
```

will be treated by the compiler as the same symbolic name. The choice of a symbolic name is up to the programmer, with the exception that reserved words (see Appendix B) cannot be used as names.

This feature of PL-11 is designed to encourage programmers to choose meaningful names for their variables, procedures, and labels. While this is good programming practice in any environment, it is extremely important in a project such as Omega, where there are at least three different kinds of computers, and many, many programmers at all levels of familiarity with the machines and the languages. In such a situation, it will be normal for people to be using and modifying programs they did not write, and perhaps that were intended for use on one machine and are being adapted for another. Therefore it is essential that the programs be easily readable and understandable to everyone. Whereas the symbol XRFTB may have meaning to the author of the program, it is totally useless as a meaningful symbol to anyone else reading the program (and perhaps even to the author a few weeks later). The symbol CROSSREFTABLE is a much more meaningful choice of identifier than XRFTB. Even in a simpler case, DIGITMASK is preferable to DMSK. People do not talk or write in pidgin English -- why should they program in it? Why make it more difficult than it already is by forcing the programmer to think in a cramped, artificial manner (XGLNK, ZBIN, ILGBLK) when a much more readable, pronounceable, and understandable style is possible?

Examples:

<table>
<thead>
<tr>
<th>Illegal</th>
<th>Legal</th>
</tr>
</thead>
<tbody>
<tr>
<td>IQRI</td>
<td>IQRI</td>
</tr>
<tr>
<td>VERYLONGIDENTIFIER</td>
<td>$ABC</td>
</tr>
<tr>
<td>BEGINNING</td>
<td>BEGIN</td>
</tr>
<tr>
<td>A123B</td>
<td>A:B</td>
</tr>
<tr>
<td>WHYNOT</td>
<td>WHY NOT</td>
</tr>
<tr>
<td>TRYTT</td>
<td>TRY-IT</td>
</tr>
<tr>
<td>DOIT</td>
<td>DO</td>
</tr>
</tbody>
</table>

3.2 Identifiers

All memory cells (variables), registers, and procedures are referred to in PL-11 by symbolic names that are chosen by the programmer. All identifiers (except those described in 3.3 below) must be declared before they are used. For details, see Section IV on declarations.
3.3 Pre-declared identifiers

The compiler is prepared to recognize a small set of names as having predefined meaning. Thus R0, R1, R2, R3, R4, R5, have the meaning of Register1, Register2, ..., Register5, and can be used by the programmer without declaring them. Pre-declared identifiers can, however, be re-declared by the programmer, in which case they will have whatever new meaning the programmer assigns to them. In this sense, they are not reserved. The pre-declared identifiers and their meaning are listed in Appendix C.

3.4 Labels

A label is used to name a statement that will be later referred to by a GOTO statement. Unlike identifiers, labels do not have to be declared, since they are automatically defined when they are used to name a statement. Any number of labels separated by colons (:) may prefix the same statement, but of course labels are not required on any statement. Owing to the powerful structuring facilities in PL-11, which include blocks, the IF, CASE, and iteration statements, and the ability to define both local and global procedures, the use of labels in a PL-11 program should be minimal, perhaps an order of magnitude less than the number required in an ordinary assembly language.

A label definition consists of the symbolic name, followed by a colon (:) preceding the statement being labelled. Examples are:

<table>
<thead>
<tr>
<th>Legal</th>
<th>Illegal</th>
</tr>
</thead>
<tbody>
<tr>
<td>HERE : (statement)</td>
<td>HERE (statement)</td>
</tr>
<tr>
<td>A : B : C : (statement)</td>
<td>A,B,C : (statement)</td>
</tr>
<tr>
<td>L19FCH : (statement)</td>
<td>19 : (statement)</td>
</tr>
</tbody>
</table>

3.5 Equates

An equate name is one that is declared to represent a compile-time value only. It does not represent any entity in the run-time machine (such as a register, memory cell, statement, etc.), but is a way of giving a programmer-defined symbolic name to a literal value. Its purpose is simply one of convenience for the programmer. See Section IV.3 for details.

4. SOURCE PROGRAM FORMAT

4.1 Cards

PL-11 is a free-field, card independent language designed to make programs easy to write and to read. Columns 1-72 inclusive on every card are interpreted by the compiler such that column 1 on card 2 immediately follows column 72 on card 1, column 1 on card 3 immediately follows column 72 on card 2, etc. Columns 73-80 on all cards are completely ignored, and may be used for sequencing if desired. Any number of complete or partial statements may be punched anywhere on a card, since all statements are required to end with a semi-colon (;), and all card boundaries are completely ignored.

4.2 Spacing

Spacing should be used by the programmer to improve readability and to exhibit the structure of the program. Except for the following restrictions, blanks are entirely optional and have no significance to the compiler.

a) No blanks may be embedded within a name or literal.

b) Two or more names and/or reserved words and/or literals appearing in succession must be separated by at least one blank, in order that the compiler be able to recognize them as distinct entities.

c) Blanks appearing within a string are significant and represent the ASCII, EBCDIC or RADSO code for a space, depending on the type of string.
4.3 Comments

Comments may appear anywhere in a PL-11 program that blanks may appear (except in a string literal). Usually they will be written between statements, although this is not necessary. The form of a comment is: the reserved word COMMENT followed by any sequence of characters (not including a semi-colon) terminated by a semi-colon (;). A single comment may extend over several cards, and statements and comments can be intermixed on the same card. Examples:

```
COMMENT THIS IS A COMMENT;
COMMENT G1; COMMENT WHY?; COMMENT BY!!;
COMMENT ! a # - + & ? = ( ) ! . , : ;
```

4.4 The block

PL-11 is a block-structured language, like ALGOL. This means that anywhere that a statement may appear, a block may appear instead. A block consists of the reserved word BEGIN, with a required matching reserved word END. In between the BEGIN and the END, in other words inside the block, may appear any number of declarations and statements. This implies that blocks may be nested. Thus the following are equivalent:

```
BEGIN
BYTE X;
0 => X;
END

BEGIN
BYTE X;
BEGIN 0 => X; END;
END
```

The most useful application of blocks is in control statements (Section VII) and iteration statements (Section VIII), since these only allow one statement -- and hence block.

4.5 Modules

The syntactic definition of a PL-11 module is "any statement followed by a period(.)", or "any procedure declaration followed by a period (.)". Usually a module requires more than one statement, so the statement will be replaced by a block. This will look like:

```
BEGIN
  (declarations for the whole module)
  (executable statements)
END.
```

or

```
GLOBAL PROCEDURE ABC;
BEGIN
...
END.
```

Of course, the positioning of these items on a card, and the spacing, is irrelevant. A recommended programming practice is to always follow the initial BEGIN with a comment describing the program. A short example of a "complete" PL-11 source program is:

```
BEGIN COMMENT PROGRAM TO ADD UP 10 NUMBERS;
ARRAY 10 INTEGER A = (1,2,3,4,5,10,20,30,40,50);
INTEGER SUM = 0;
FOR R1 FROM 0 STEP 2 UPTO 18 Do SUM + A(R1);
END.
```
A PL-11 program often consists of a single module, since the main program and all the procedures which it calls are compiled together in a single compilation step. It is, however, possible for a total running program in the PDP-11 to be built by the DOS or CLI linkage editors from separate modules, which are compiled separately. In this case there would be one module containing the main program and perhaps some of the procedures called from the main program, and the other modules containing the other procedures. This referencing is accomplished in PL-11 by use of the GLOBAL and EXTERNAL declarations of procedures. The compiler distinguishes between modules containing only procedures, and one which contains a main program (as well as procedures). For main program modules, the compiler supplies to the linkage editor the address of the first executable instruction, and the linkage editor in turn puts this address into the executable load module so that it may be used by the PDP-11 loader to start execution at the proper address. In this way the load-start address is separated from the execution-start address.

This produces several benefits to the user: (1) The main program module does not have to be input first to the linkage editor -- modules may be read in any order, since the module containing the main program will identify itself to the linkage editor; (2) stand-alone programs are self-starting -- in particular, if a PL-11 program is a monitor, it will begin execution as soon as it is loaded without operator intervention.
Section IV. DECLARATIONS IN PL-11

All identifiers must be declared by the programmer before they can be used in a PL-11 program. The only exception is the small set of pre-declared identifiers (see Section III.3.2 and Appendix C).

Since PL-11 is a block-structured language, declarations can appear after any BEGIN, and will be in effect only until the matching END is encountered, at which point the effect of the declaration is voided. (See Section III.4.4 on Block Structure.) Declarations can appear in any order, the only requirements being that 1) all declarations in a block must appear before any of the executable statements in that block; 2) any references used as part of a definition (as in a procedure or a synonym) must have a well-defined value (that is, no forward referencing is possible, except for labels, which are not declared).

The following are the kinds of declarations allowed in PL-11.

1. VARIABLE DECLARATIONS

   A variable declaration serves: 1) to reserve core storage; 2) to associate a symbolic name with the storage; 3) to associate a type with the storage; and 4) to give an initial value to the storage (see Section IV.4). Variable declarations can be further classified into simple and tabular declarations.

1.1 Simple variable declarations

   There are six kinds of simple variable declarations, depending on which of the six types allowed in PL-11 is associated with the storage. The six types are:

   1.1.1 BIT

   A BIT variable occupies a single bit in storage. The declaration causes the next available bit in store to be allocated.

   1.1.2 BYTE

   A BYTE variable occupies a single 8-bit byte in storage. The declaration causes the next available byte to be allocated.

   1.1.3 INTEGER

   An INTEGER variable occupies a single 16-bit word in storage. The declaration causes the next available word, aligned automatically on an even-byte boundary, to be allocated.

   1.1.4 LOGICAL

   The same as an INTEGER, above, with one important exception, which is its use in comparisons (see Section VI.1.1). Otherwise they are interchangeable.

   1.1.5 REAL

   A REAL variable occupies two consecutive 16-bit words in storage. The declaration causes the next two available words, aligned automatically on an even-byte boundary, to be allocated.
1.1.6 DOUBLE

A DOUBLE variable occupies four consecutive 16-bit words in storage. The declaration causes the next four available words, aligned automatically on an even-byte boundary, to be allocated.

The form of a simple declaration is:

\[(\text{type}) \ (\text{list of symbolic names});\]

where (type) is one of the reserved words:

BIT, BYTE, INTEGER, LOGICAL, REAL, DOUBLE.

Note that all variable types may be used on any machine, but none of the operations and functions of REAL and DOUBLE are available on the model 20. They are all assumed for the model 45. On the model 40, the [FPP card (see Section II.2)] must be used to tell the compiler that the floating-point option is available.

Example:

\[
\begin{align*}
\text{BYTE} & \ A,B; \\
\text{BIT} & \ X; \\
\text{INTEGER} & \ AA,BB; \\
\text{BIT} & \ C,D,E,F,G,H,I,J,K; \\
\text{BYTE} & \ TIN; \\
\text{REAL} & \ PQ,R; \\
\text{DOUBLE} & \ CC; \\
\text{BYTE} & \ Z;
\end{align*}
\]

Storage is allocated to each name as it appears, in bit, 1-byte, 1-, 2- or 4-word units as necessary. Therefore the above declarations would cause the following relative address assignments:

<table>
<thead>
<tr>
<th>Relative byte addr.</th>
<th>Bit No. within byte</th>
<th>Name</th>
<th>Size in bytes</th>
<th>Element type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>A</td>
<td>1</td>
<td>BYTE</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>B</td>
<td>1</td>
<td>BYTE</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>X</td>
<td>-</td>
<td>BIT</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>AA</td>
<td>2</td>
<td>INTEGER</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>BB</td>
<td>2</td>
<td>INTEGER</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>C</td>
<td>-</td>
<td>BIT</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>D</td>
<td>-</td>
<td>BIT</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>E</td>
<td>-</td>
<td>BIT</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>F</td>
<td>-</td>
<td>BIT</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>G</td>
<td>-</td>
<td>BIT</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>H</td>
<td>-</td>
<td>BIT</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>I</td>
<td>-</td>
<td>BIT</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>J</td>
<td>-</td>
<td>BIT</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>K</td>
<td>-</td>
<td>BIT</td>
</tr>
<tr>
<td>10</td>
<td>-</td>
<td>TIN</td>
<td>1</td>
<td>BYTE</td>
</tr>
<tr>
<td>12</td>
<td>-</td>
<td>PQ</td>
<td>4</td>
<td>REAL</td>
</tr>
<tr>
<td>16</td>
<td>-</td>
<td>R</td>
<td>4</td>
<td>REAL</td>
</tr>
<tr>
<td>20</td>
<td>-</td>
<td>CC</td>
<td>8</td>
<td>DOUBLE</td>
</tr>
<tr>
<td>28</td>
<td>-</td>
<td>Z</td>
<td>1</td>
<td>BYTE</td>
</tr>
</tbody>
</table>
Note that the bytes at relative addresses 3 and 11 are unused, since the next variables to be assigned, AA and PQ, are INTEGER and REAL, respectively, and must therefore be assigned to even-address bytes (i.e. on a word boundary).

1.2 Tabular variable declarations

There are two kinds of tabular variable declarations in PL-11, called ARRAY and STACK. The only difference is the address given to the symbolic name: arrays are assigned the address of the first (lowest address) cell in the table; and stacks are assigned the address following the last (highest address) cell in the table.

Otherwise, both kinds of tabular declarations are used to allocate a block of one or more cells, each having identical type. The table as a whole has a single name, and the individual elements in the table are referred to by subscripting that name (see Section V on Expressions). The form of these declarations is:

\[(\text{table-kind})[(\text{literal-expression})]\text{type}\] \[(\text{list of symbolic names})\]

where (table-kind) is one of the reserved words ARRAY or STACK, and (literal-expression) is the number of elements in the table. The (literal-expression) is optional, and its omission gives rise to self-defining arrays and stacks, for details of which see Section IV.4.7.

Examples:

\[
\begin{align*}
\text{ARRAY} & \quad 10 \quad \text{BYTE A,B,C;} \\
\text{STACK} & \quad 7 \quad \text{INTEGER Y,Z;} \\
\text{ARRAY} & \quad 60 \quad \text{LOGICAL P,Q;} \\
\text{STACK} & \quad 2*3+1 \quad \text{BYTE ROAR;} \\
\text{ARRAY} & \quad 4 \quad \text{DOUBLE TIME;}
\end{align*}
\]

These declarations would cause the following relative address assignments:

<table>
<thead>
<tr>
<th>Relative byte addr.</th>
<th>Table size</th>
<th>Name</th>
<th>Size in bytes</th>
<th>Element type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10-bytes</td>
<td>A</td>
<td>1</td>
<td>BYTE</td>
</tr>
<tr>
<td>10</td>
<td>10-bytes</td>
<td>B</td>
<td>1</td>
<td>BYTE</td>
</tr>
<tr>
<td>20</td>
<td>10-bytes</td>
<td>C</td>
<td>1</td>
<td>BYTE</td>
</tr>
<tr>
<td>44</td>
<td>7-words</td>
<td>Y</td>
<td>2</td>
<td>INTEGER</td>
</tr>
<tr>
<td>58</td>
<td>7-words</td>
<td>Z</td>
<td>2</td>
<td>INTEGER</td>
</tr>
<tr>
<td>58</td>
<td>60-words</td>
<td>P</td>
<td>2</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>178</td>
<td>60-words</td>
<td>Q</td>
<td>2</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>305</td>
<td>7-bytes</td>
<td>ROAR</td>
<td>1</td>
<td>BYTE</td>
</tr>
<tr>
<td>306</td>
<td>32-bytes</td>
<td>TIME</td>
<td>8</td>
<td>DOUBLE</td>
</tr>
</tbody>
</table>

Note that the "number" in the declaration is always the number of elements in the array. If the type of these elements is INTEGER or LOGICAL, twice this "number" of bytes will actually be allocated. If it is REAL or DOUBLE, then four or eight times this "number" will be allocated, respectively.

**Important note:**

It is not possible to declare BIT arrays or stacks in PL-11, because the PDP-11 hardware does not have the necessary mechanism to select a bit in an array according to the value in an index register. Bit selection must be done statically at compile-time with the symbolic declarations described above. Dynamic run-time bit selection (i.e. the Nth bit of an array, where N is variable at run-time) must be explicitly programmed by the programmer using the shifting and masking operations of PL-11.

1.3 Limitation on array allocation size

Each array and stack variable which causes storage to be allocated when it is declared is restricted to a size of at most 512 integers or 1024 bytes. However, with this restriction
the compiler is able to keep the initial values for all declared variables on disk, and therefore the total amount of variable storage which can be allocated by a PL-11 program is now potentially infinite (limited only by the size of the CII disks!).

Array and stack synonyms can be of any size, since they do not cause storage to be allocated or initialized. Therefore, if the programmer wishes to use an array of say 2000 integers, he should allocate and initialize the space as 4 arrays of 500 integers each, then define a synonym name for the whole array. For example:

\[
\begin{align*}
\text{ARRAY 500 INTEGER T1, T2, T3, T4;} \\
\text{ARRAY 2000 INTEGER BIGONE SYN T1;} \\
\end{align*}
\]

2. **SYNONYM DECLARATIONS**

It is often useful to associate more than one name with a single storage address, which is the function of the FORTRAN Equivalence statement; and this is the purpose of Synonym declarations in PL-11 -- to give synonymous names to the same storage. It is also possible for the synonymous names to have different types. In PL-11 the "synonym" concept has been slightly generalized to allow a name to be synonymous with any "storage reference" (see Section V on Expressions). The only requirement is that any names referred to must have been previously defined. There are three kinds of synonym declarations -- variable, register, and bit synonyms.

2.1 **Variable synonyms**

The form is the same as for the two types of declarations (namely simple and tabular) declared above, with the addition that the (list of symbolic names) may include synonyms. A synonym has the form

\[(\text{name}) \text{ SYN (reference)}\]

The (reference) must be a name already declared, and the (name) must not be a type that would occupy more core space than the (reference). They may both be of type BIT, but if only (name) is of type BIT, then the alternative BIT synonym (see Section IV.2.3, below) must be used.

2.1.1 **Illegal examples**

\[
\begin{align*}
\text{BYTE ABC ; INTEGER X SYN ABC;} \\
\text{BYTE ABC ; BIT X SYN ABC;} \\
\end{align*}
\]

2.1.2 **Legal examples**

\[
\begin{align*}
\text{BYTE ZAP SYN A, ZORCH SYN B, ZING SYN C;} \\
\text{INTEGER TOP SYN X, BOTTOM SYN Y, MIDDLE SYN IND(2);} \\
\text{REAL A, B, C SYN A;} \\
\text{INTEGER AA, BB, CC SYN MEMORY(100), F SYN BB;} \\
\text{BYTE AAA SYN AA, DD SYN C(1);} \\
\text{LOGICAL LTOP SYN X, MARKER SYN Q(4);} \\
\text{ARRAY 3 BYTE P SYN X(1), S SYN Y;} \\
\text{INTEGER PLACE SYN PUSH (R1), ALSO SYN IND(POP(R2));} \\
\text{STACK 4*3+1 LOGICAL MARK SYN PLACE;} \\
\text{REAL X; INTEGER XHI SYN X, XLOW X(2);} \\
\end{align*}
\]

The type associated with each name is the type included in the declaration; the storage is that defined by the reference. Names cannot be made synonymous with literals, since literals do not define any storage. Literals can, however, be given symbolic names by use of the EQUIATE declaration, which is described in Section IV.3. Names cannot be made synonymous with register references except as described next in Section IV.2.2.
Notice that synonyms have as their LENGTH the number of bytes specified in their declaration. For example:

```
ARRAY 20 BYTE X;
ARRAY 15 REAL Y;
ARRAY 17 LOGICAL Z;
ARRAY 2 LOGICAL XX SYN X,
       YY SYN Y,
       ZZ SYN Z;
```

Then XX, YY and ZZ all have LENGTH 4.

2.2 Register synonyms

Ordinarily the registers are referred to in PL-11 by the pre-declared names R0, R1, R2, ..., R5. However, it is often convenient to be able to use other symbolic names for the registers. The form of register synonym declarations is:

```
(type) (list of synonyms);
```

In this case only register references can be used as synonyms.

Examples:

```
INTEGER MARKER SYN R1, POINTER SYN R0;
BYTE P SYN R5, T SYN R4, Q SYN T, Q2 SYN MARKER;
```

These synonyms can obviously be used anywhere the original register names can be used; as for example, in subscripts and in stack references, i.e. A(MARKER), PUSH(P), POP(T), etc.

2.3 BIT synonyms

It is possible to give symbolic names to bits in any variable, register, or memory cell through the use of bit synonyms. The form is:

```
BIT (list of synonyms);
```

Each synonym is of the form

```
(name) SYN (literal expression) OF (reference)
```

Example:

```
INTEGER AA, BB, CC;
BIT AA15 SYN 15 OF AA, BB0 SYN 0 OF BB, FLAG SYN 2 OF AA, IOFLAG SYN 4 OF CC;
BIT Q SYN 15 OF CCR, P SYN 2 OF MEMORY(44);
BIT XX SYN 10 OF R4;
BIT ZZ SYN XX;
```

The symbolic name AA15 is given to bit 15 of the integer variable AA, BB0 to bit 0 of BB, FLAG to bit 2 of AA, etc. ZZ and XX are both symbolic names for bit 10 of register 4. Bit synonyms can be used in exactly the same manner as other bits.

3. EQUATE DECLARATIONS

It is often convenient to be able to define symbolic names that represent literal (i.e. compile-time) values, and this is the purpose of the Equate declaration in PL-11. The form is:
EQUATE (list of equates);

where the form of the elements of (list of equates) is

(name) SYN (literal expression)

Examples:

EQUATE A SYN 25, B SYN #F1A, C SYN B+2*A;
EQUATE X SYN REF(T(4)), Y SYN X-10, Z SYN 2*C/A;
EQUATE Q SYN REF(T), Q2 SYN REF(T)-REF(S);

In effect, the equate statement makes the symbolic name synonymous with the compile-time value of the literal expression. The symbolic name can then be used anywhere in PL-11 where a literal can be used. See Section V.5 for details of literal expressions.

4. INITIALIZATION

At the time storage is allocated to a variable, it is possible to specify an initial value that the storage will have at the time the program is loaded. This initialization is part of the storage declaration statement, rather than a separate statement (such as the DATA statement in FORTRAN). Synonyms cannot be initialized.

4.1 Simple initialization

The initial value for a variable must be a literal (or literal expression), and is separated from the variable in the declaration list by an equals sign (=). For example:

INTEGER A = 0, B = 10, C = -10, D = #E09*7;
BYTE  X = 'H', Y = #FF, Z = 0, W = -15+#A7, T = 'P'/5;
LOGICAL E, F = 0, G, H = -9, I = REF(A);

Note that E and G in the last declaration are not initialized. Therefore their value when the program begins execution will be automatically initialized to zero. This feature of PL-11 is different from most other languages, where uninitialized variables have some arbitrary value at the beginning of program execution, thereby causing many obscure bugs.

4.2 Initialization lists

In order to be able to initialize elements of an array, a list of literal expressions enclosed in parentheses may follow the equals sign. For example:

ARRAY 5 INTEGER A = (0, 0, 1, 1, 2), B = (10, 9, 8, 7, 6),
                   C = (#FF, -15, 2*E, REF(A(4))) ;

The initial values are assigned, one per element, starting with the first element in the array. If there are fewer elements in the initial value list than in the array, the last elements in the array are initialized to zero. In the above example, the fifth element of C (i.e. C(5)) is automatically initialized to zero. If the list is longer than the storage allocated to the array, an error message is issued by the compiler.

4.3 Repetition in initialization lists

For convenient initialization of large arrays, a repetition value followed by a * may be specified in front of a parenthesized list of initial values. This repetition value may be any literal expression (except an address constant). In addition, elements within a list can themselves be lists, with optional repetition values preceding them, and this nesting can occur to any depth. For example:

ARRAY 100 INTEGER A = 100*(-1), B = 50*(0,1),
                  C = 10*(0, 1, 5*(0), 1, 2*(0)), D = (50*(0), 50*(-1)),
                  E = (25*(0,1), 2 * 5*(1, 2, 3*(#F)));
In this example, all 100 elements of A are initialized to minus one; the 100 elements of B alternate zeros and ones; those of C occur in 10 repetitions of the pattern 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0; the first 50 elements of D are set to zero, the last 50 to minus one; the first 50 elements of E alternate zeros and ones and the last 50 consist of 10 repetitions of the pattern 1,2 #F, #F, #F.

**Warning:**

The values to be repeated *must* be enclosed in parentheses, even if there is only one. ARRAY 5 INTEGER X = 5*2; initializes the first element to ten (5 times 2), and the rest to zero. ARRAY 5 INTEGER X = 5*(2); will initialize all 5 elements to two.

### 4.4 String constants as initial values

When used as an initialization value, a string is treated as a consecutive sequence of bytes (1 character per byte) that are loaded into ascending byte addresses in storage. Thus the "boundaries" between elements in an array are ignored. For example:

```
ARRAY 18 BYTE MESSAGE = 'THIS IS A MESSAGE.';
```

will initialize all 18 bytes of the array MESSAGE -- the first to 'T', the second to 'H', the third to 'I', etc., the last to '.'. In the following example

```
ARRAY 9 INTEGER MESSY = 'THIS IS A MESSAGE.';
```

is equivalent to

```
ARRAY 9 INTEGER MESSY = ('TH', 'IS', ' I', ' S ', ' A ', ' ME', ' SS', ' AG', ' E.');
```

Obviously the first form is preferable to the second. Care must be used when strings are used with repetition values, since

```
ARRAY 5 INTEGER TABLE = 5(' ');
```

initializes each element to a single '*' in the low-order byte, whereas

```
ARRAY 5 INTEGER TABLE = '******';
```

initializes the first two elements to '*', the third element to '*', and the last two elements to zero.

### 4.5 Initialization of REAL and DOUBLE variables

As with all other types of variables, REAL and DOUBLE variables are initialized in the same statement in which they are declared. For example:

```
REAL PI=3.14159, TEN=1E+1, FIFTY=50.;
DOUBLE DPI=3.1415926536D, Z=99D,ZQ, ZZ=.123D;
```

Variables such as ZQ in the above example, which are not explicitly initialized by the programmer, are automatically initialized to zero by the compiler. As with other data types, repetition lists can be used to initialize arrays and stacks. For example:

```
ARRAY 64 REAL BIGZ = (1.0,2.3,10*(4.1E+5,.111),42*(.9));
```

If desired, constants of other types can be used to initialize REAL and DOUBLE variables, but considerable care must be exercised by the programmer when doing this, since the compiler performs no conversion (from integer to real, for example), and does not change the number of bits needed to represent the constant in order to fit the size of the variable. Integer, octal, and hexadecimal constants are represented in 16 bits, real constants in 32 bits, double constants in 64 bits, and strings in 8n bits, where n is the number of characters in the string. For example:

```
REAL P=22, R=#FFE1, S=$1234;
```

initializes the first word of P to the integer value 22, the second word to zero; the first word of R to the bit pattern FFE1 hex, the second word to zero; the first word of S to the bit pattern 1234 octal, the second word to zero.
4.6 Initializing stacks

Stacks are initialized in the same manner as arrays, starting with the lowest relative address for the first initial value, and assigning successive values to successively ascending storage locations. Since stacks are usually used in the reverse order (high address to low address), care must be used to initialize stack storage as desired.

4.7 Self-defining arrays and stacks

A very common use of byte arrays is to store character strings that are used to print headings, messages, etc. If the programmer, however, is obliged to count the number of characters between the quotation marks, there is a possible source of error introduced. Also, this places an unnecessary burden on the programmer, as the compiler should be able to do this work. If the message is changed, the count would also have to be changed.

To remove the annoyance and errors this would cause, the compiler contains "self-defining arrays and stacks", whose size is automatically determined by the number of initial values given in the declaration. This is done simply by omitting the number of elements in the array declaration. The example in Section 4.4 above then becomes

```
ARRAY BYTE MESS1 = 'THIS IS A MESSAGE.';
```

Since the initial value of MESS1 is a string of 18 characters, 18 bytes will automatically be allocated as the size of MESS1.

In order to utilize this number in the program, one may use the LENGTH literal. Thus LENGTH(MESS1) is the number of bytes allocated to MESS1, 18 in this case. Hence if a PRINT procedure requires the message address in R0, and its length in R1, one could print the message with the following sequence:

```
REF(MESS1) => R0;
LENGTH(MESS1) => R1;
PRINT;
```

Now if the programmer wishes to change the text of his message, this is all he has to change, since when the program is recompiled, the new text string will determine the number of bytes allocated and the LENGTH of MESS1.

As a final note, it should be pointed out that the concept of self-defining arrays applies to all types -- INTEGER, LOGICAL, REAL, and DOUBLE, as well as BYTE. The number of bytes allocated is determined by the number of elements specified in the initialization list, and this list can include any mixture of constants and repetition factors, as is shown in the following examples:

```
ARRAY INTEGER X = ($157,22,'THIS IS', 2*(0,1),5),
Y = 10*(-19*LENGTH(X),REF(X),REF(Y)),
Z = 2*(4*LENGTH(X),4*(LENGTH(Y),REF(Y))),
P = LENGTH(X)/2*(0),
S = LENGTH(Y)*5,
W = ('HOW',LENGTH(S),REF(S),'TOO');
STACK BYTE L = (4*(1,2*(0,5,7),-3),'HIKE.'),
M = 9*(Z);
```

then

```
LENGTH(X) is 22
LENGTH(Y) is 60
LENGTH(Z) is 56
LENGTH(P) is 22
LENGTH(S) is 2
LENGTH(W) is 12
LENGTH(L) is 37
LENGTH(M) is 9 .
```
5. **PROCEDURE DECLARATIONS**

The concept of procedures (or subroutines) is an important one in many programming techniques, and the procedure facility in PL-11 is designed to provide a flexible yet powerful method by which programmers can conveniently define and use procedures. Procedures can be declared in *any* block in a PL-11 program, with the scope of their definition being simply that block (exactly the same as any identifier or label). This means that local procedures can be defined anywhere that is convenient, and, in particular, that procedures can be declared within procedures to any depth (see Section III.4.4 on Block Structure). A procedure declaration is *not* compiled separately (as in FORTRAN), but is part of the block in which it is declared (as in ALGOL).

The form of a procedure declaration is:

```
    (heading); (body);
```

where the (heading) has the form

```
    PROCEDURE (symbolic name)[((register))]
```

and the (body) is *any* statement or block.

Examples:

```
    PROCEDURE P (R1); IF A>B THEN 1 => R5 ELSE 0 => R5;
    PROCEDURE COMPUTE (R4); A(R1) + B(R2) - C(R3) * 4 + D;
    PROCEDURE LONG (R0);
    BEGIN .................... END;
```

The (symbolic name) is the name by which the procedure can be called from the block. The (register) optionally specified in parentheses after the name is the one used in the PDP-11 jump-to-subroutine instruction whenever the procedure is called. This means that when the subroutine is being executed, that register will contain the return address. Hence this is often called the "return register". If no (register) is specified, then the program counter (PC) is used in the jump-to-subroutine instruction, and the return address is on the stack.

The (body) of a procedure is *any* statement or block, and consequently can reference any identifiers or labels that occur in outer blocks. See Section IX on Procedures for a more detailed description, and for the methods of calling, returning, and parameter passing in PL-11 procedures.
Section V. EXPRESSIONS IN PL-11

The mechanism for manipulating data in PL-11 is called the *data expression*, or simply, the *expression*. All computations and all movements of data are written as expressions by the programmer, and these are translated by the compiler into the necessary PDP-11 instruction sequences.

An expression has two basic components -- *operands* and *operators*.

1. OPERANDS

Operands can be further classified into two primary kinds -- those that define a value, and those that define a storage cell or register in the PDP-11 hardware. In PL-11, all operands in the first category are called *literals*. The run-time value of a literal is a constant value known at compile-time. The literal does not occupy any data storage at run-time, but, depending on the context, it is either made part of the instructions generated by the compiler (i.e. immediate instructions), or is used to provide information to the compiler, such as how much storage to allocate. Expressions containing only literal operands are evaluated at compile-time, and are called "literal expressions" (see Section V.5).

The run-time value of register or storage cell operands is unknown at compile-time, and will in all probability change as the program is executed. Therefore operands in this category do not define the value to be used at run-time, but rather they specify a *register* or *variable reference* which is made part of the instructions generated by the compiler. At run-time, this reference will be used by the addressing hardware of the PDP-11 to obtain a value for use in the computation. There are several kinds of reference, called *reference modes*.

1.1 Normal references

These are written as the name of the register or variable (i.e. storage cell). The run-time value of this reference is the contents of the register or variable at that moment during the execution of the program.

Examples:

\[
R1 \quad ABCD \quad X \quad RS.
\]

1.2 Subscripted references

A subscripted reference in PL-11 is written as the name of a *variable* (not a register) followed by a subscript expression enclosed in parentheses. A subscript expression can contain only literal operands and/or one register reference that is added to the other operands.

<table>
<thead>
<tr>
<th>Legal</th>
<th>Illegal</th>
</tr>
</thead>
<tbody>
<tr>
<td>A(R5)</td>
<td>A(-R5)</td>
</tr>
<tr>
<td>B(4)</td>
<td>R5(4)</td>
</tr>
<tr>
<td>C(R1=10)</td>
<td>C(10+R1)</td>
</tr>
<tr>
<td>D(2*24+R3-1)</td>
<td>D(2*R3-24)</td>
</tr>
<tr>
<td>E(#1A+3/19)</td>
<td>E(R1+R2)</td>
</tr>
</tbody>
</table>
If k is the run-time value of the subscript expression, then the run-time value of this reference will be the contents of a variable whose location in memory is k bytes from the location of the variable being subscripted.

**Warning:**

INTEGER and LOGICAL values must be stored on even byte addresses in the PDP-11. When allocating storage the compiler always ensures proper address alignment for the declared variables. However, it is the programmer's responsibility to ensure that a subscript expression does not cause a misalignment. This simply means that in referencing integer or logical variables, subscript expressions should always have an even (or 0) value.

For example, if A is a five-element integer array, the references to each of the five elements would be A(0), A(2), A(4), A(6), A(8), respectively.

### 1.3 Stack references

There are two kinds of stack references: pushing a stack to add a new top item; and popping a stack to delete the top item. In PL-11, stack references are written as one of the reserved words PUSH, POP, BPUSH, BPOP, LPUSH, or LPOP (depending on the type of the item being pushed or popped), followed by a register reference enclosed in parentheses.

**Examples:**

- PUSH(R2)
- POP(R1)
- BPUSH(R5)
- LPOP(R5)

The value of this reference will be the storage cell whose address is in the specified register (also called the stack pointer). In addition, stack references have an important side-effect -- the value of the stack pointer is incremented (on POPs) or decremented (on PUSHes) automatically by the autoincrement and autodecrement features of the PDP-11 hardware. References that produce integer or logical values (PUSH, POP, LPUSH, LPOP) will change the stack pointer by 2 each time. References to byte stacks (BPUSH, BPOP) will change the stack pointer by 1 each time.

### 1.4 Indirect references

An indirect reference in PL-11 is written as one of the reserved words IND, BIND, or LIND (depending on the type of the item obtained indirectly) followed by an integer or logical reference (of the kind described previously in 1.1, 1.2, or 1.3 above), enclosed in parentheses.

**Examples:**

- IND(R2)
- IND(A)
- BIND(A(R1))
- BIND(B(R4+10))
- LIND(PUSH(R5))
- LIND(POP(R5))

The value of this reference will be the storage cell whose address is produced as the value of the reference enclosed in parentheses. Only one level of indirect referencing (called "deferred address mode" by DEC) is allowed in the PDP-11 hardware. Hence the following is illegal:

IND(IND(A))

### 2. OPERATORS

Operators in PL-11 are special symbols or reserved words that cause one or more PDP-11 instructions to be generated. The operator symbols have been chosen to be as consistent as possible with existing mathematical and programming language conventions in representing to the programmer the effect caused by the generated instructions. Operator rather than function notation was chosen because of its familiarity, and because it is particularly applicable to a linear left-right expression evaluation (see Section V.3). Operators can be classified into two kinds: **monadic** (those that operate on a single operand), and **dyadic** (those that operate on two operands).
2.1 Monadic operators

The following table gives the symbols for the PL-11 monadic operators and their meanings. Except for ABS, each operator is translated into a single PDP-11 instruction.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>twos complement (arithmetic negation)</td>
</tr>
<tr>
<td>ABS</td>
<td>absolute value</td>
</tr>
<tr>
<td>COMP</td>
<td>ones complement (logical negation)</td>
</tr>
<tr>
<td>SWAP</td>
<td>interchange 2 bytes in a word</td>
</tr>
<tr>
<td>SLA</td>
<td>shift left 1-bit arithmetic</td>
</tr>
<tr>
<td>SRA</td>
<td>shift right 1-bit arithmetic</td>
</tr>
<tr>
<td>SLC</td>
<td>shift left 1-bit circular</td>
</tr>
<tr>
<td>SRC</td>
<td>shift right 1-bit circular</td>
</tr>
</tbody>
</table>

2.2 Dyadic operators

The following are the "standard" PL-11 dyadics -- each operator can be translated into a single PDP-11 machine instruction.

| Symbol | Meaning | |
|--------|---------| |
| +      | add     | |
| -      | subtract| |
| MASKON | bit set | |
| MASKOFF| bit clear| |
| =>     | move    | |

2.3 Extended dyadic operators for the model 20 EAU option

On the model 20, it is possible to have an optional extended arithmetic unit (EAU) for multiplying, dividing, and shifting. In PL-11 there are two types of dyadic operators for using the extended arithmetic unit: 1) those that simply initiate the unit's operation (all operand set-up and result retrieval must be done explicitly by the programmer) -- these are translated into a single PDP-11 instruction; 2) those that perform all the necessary set-up and retrieval operations in order to use the unit correctly -- these will generate several PDP-11 instructions (similar to a "macro" expansion), but require correspondingly less effort by the programmer who in this case does not really "see" the extended arithmetic unit. (It is expected that systems programmers will use the first type, application-type users the second.)

2.3.1 Operators for initiation only (called "programmed" dyadics)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULT</td>
<td>move right operand into multiplier</td>
</tr>
<tr>
<td>DIV</td>
<td>move right operand into divisor</td>
</tr>
<tr>
<td>SHIFTA</td>
<td>move right operand into arithmetic shifter</td>
</tr>
<tr>
<td>SHFTL</td>
<td>move right operand into logical shifter</td>
</tr>
</tbody>
</table>

2.3.2 Macro-type operators (called "macro" dyadics)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>multiply</td>
</tr>
<tr>
<td>/</td>
<td>divide (result is quot:ent)</td>
</tr>
<tr>
<td>REM</td>
<td>remainder</td>
</tr>
<tr>
<td>SHA</td>
<td>shift arithmetic</td>
</tr>
<tr>
<td>SFL</td>
<td>shift logical</td>
</tr>
</tbody>
</table>
Note:
A type 1 "programmed" operation by definition always leaves the result in the MQ.

2.3.3 Equivalence of the two forms

\[
\begin{align*}
A \times B & \text{ is equivalent to } A \Rightarrow MQ \times B \Rightarrow A; \\
A / B & \text{ is equivalent to } A \Rightarrow MQ \div B \Rightarrow A; \\
A \text{ REM } B & \text{ is equivalent to } A \Rightarrow MQ \div B; AC \Rightarrow A; \\
A \text{ SHA } B & \text{ is equivalent to } A \Rightarrow MQ \text{ SHIFTA } B \Rightarrow A; \\
A \text{ SHL } B & \text{ is equivalent to } A \Rightarrow MQ; 0 \Rightarrow AC \text{ SHIFTL } B \Rightarrow A
\end{align*}
\]

2.4 Extended dyadic operators for the model 40 and model 45

Note:
In what follows for this section, only MASKFLIP is standard on both machines. All the remaining operations are standard for the model 45, but require the EIS option for the model 40. The [E]AU control card must be used for these operations, on the model 40.

2.4.1 Operations on any non-byte register

A) MASKFLIP -- EXCLUSIVE-OR

The PL-11 dyadic operator MASKFLIP is used to indicate an EXCLUSIVE-OR operation in the models 40 and 45. The right operand of this operation must be a non-byte register, but the left operand can be any PL-11 integer or logical operand. This operation works on each bit of the two 16-bit operands independently. The Nth bit in the resultant 16-bit word is a one if the Nth bits of the two operands are not the same (i.e. one is a 'one' and the other is a 'zero'), and is a zero if the Nth bits of the two operands are the same (i.e. both are one or both are zero). For example, to take the ones complement of the value of A we can write either

\[
A \text{ COMP;}
\]

or

\[
\#FFFF \Rightarrow R1; \ A \text{ MASKFLIP } R1;
\]

Of course, the first method is faster, shorter, and compatible with model 20.

The twos complement is either

\[
A-;
\]

or

\[
\#FFFF \Rightarrow R1; \ A \text{ MASKFLIP } R1 + 1;
\]

Again, the first method is preferable.

A common use of the MASKFLIP operation is to change the state of a flag word from all ones to all zeros, or all zeros to all ones. This can be done at the same time as the flag is tested, as follows:

\[
\#FFFF \Rightarrow R1;
\]

\text{IF FLAG MASKFLIP } R1 = 0
\text{THEN BEGIN COMMENT FLAG WAS ALL ONES, IS NOW ALL ZEROS; END}
\text{ELSE BEGIN COMMENT FLAG WAS ALL ZEROS, IS NOW ALL ONES; END;

Another example is for flip-flopping an index between two values:
1 \Rightarrow R1; \text{ FLIFFLOP10 MASKFLIP R1; COMMENT CHANGES ZERO INTO ONE, OR ONE INTO ZERO;}
3 \Rightarrow R2; \text{ FLIFFLOP21 MASKFLIP R2; COMMENT CHANGES ONE INTO TWO, OR TWO INTO ONE;}
6 \Rightarrow R3; \text{ FLIFFLOP24 MASKFLIP R3; COMMENT CHANGES TWO INTO FOUR, OR FOUR INTO TWO;}

Note that MASKFLIP is a natural adjunct to the already existing mask instructions MASKON and MASKOFF. MASKON sets a bit, MASKOFF clears a bit and MASKFLIP inverts (i.e. "flips") a bit.

B) \textbf{SHIFTA.-- arithmetic shift}

The PL-11 dyadic operator SHIFTA is used to indicate an arithmetic shift of any number of bits either right or left. The left operand of this operation must be a non-byte register, but the right operand can be any PL-11 INTEGER or LOGICAL operand. The low-order 6 bits of the right operand are taken as a signed value N in the range (-32, +31). If N is positive, the value of the specified register is shifted to the left N bits, all bits entering at the right are set to zero. If N is negative, the value of the specified register is shifted to the right |N| bits. All bits entering at the left are set to the value of the original sign bit. Thus a shift of N bits effectively multiplies the register value by 2^N (which if N is negative results in a division by 2^{|N|}).

For example, to multiply all N elements of an integer array X by 16, we can write:

\[
\text{REF(X) } \Rightarrow \text{ R0; FOR R1 FROM N STEP -1 DOWNTO 1 DO M0 } \Rightarrow \text{ R2 SHIFTA 4 } \Rightarrow \text{ POP(R0);}
\]

To divide all N elements by 32 we can write:

\[
\text{REF(X) } \Rightarrow \text{ R0; FOR R1 FROM N STEP -1 DOWNTO 1 DO M0 } \Rightarrow \text{ R2 SHIFTA -5 } \Rightarrow \text{ POP(R0);}
\]

For compatibility with the model 20, the PL-11 operator SHA is treated identically with the SHIFTA operator on models 40 and 45.

2.4.2 \textbf{Operations on 32-bit values}

The hardware long arithmetic shift, multiply, and divide operations require the programmer to pay somewhat more attention to detail than is the case for other operations. This is due to two factors:

- The left operand of a long arithmetic shift, multiply, and divide must be an even register (i.e. R0, R2, or R4). When working with 32-bit values, this register always holds the high-order 16 bits.

- The odd register associated with the even register specified is also involved in the operation (R1 is associated with R0, R3 with R2, and R5 with R4). When working with 32-bit values, this register always holds the low-order 16 bits.

A) \textbf{LONGSHIFT.-- long arithmetic shift}

In PL-11 the long arithmetic shift operation is indicated by the use of the dyadic operator LONGSHIFT. The left operand of this operator must be an even register (R0, R2, or R4), but the right operand can be any PL-11 INTEGER or LOGICAL operand (i.e. a register, a cell in storage, etc.). The even register specified as the left operand and its associated odd register are treated as a 32-bit twos complement number with the high-order 16-bits in the even register. The low-order 6 bits of the right operand are treated as a signed shift count N in the range (+31, -32). If N is positive, the even-odd register pair is shifted left by N bits. Bits filled in on the right are always zero. A negative shift count causes a right shift by |N| bits. Bits filled in on the left are copies of the original sign bit.

For example, to convert a 32-bit twos complement binary number stored in the two words pointed to by R1 into octal digits in ASCII at the location pointed to by R0, we can use the following routine:
GLOBAL PROCEDURE DWBINOCT;
BEGIN
EXTERNAL PROCEDURE SAVE(R5), RESTORE(R5);
SAVE;    COMMENT SAVE ALL REGISTERS ON THE STACK;
POP(R1) => R2; M1 => R3;    COMMENT HIGH-ORDER IN R2, LOW IN R3;
R0+11;    COMMENT ADJUST R0 FOR PUSHING;
FOR R5 FROM 10 STEP -1 DOWNTO 1 DO
    BEGIN    COMMENT ONCE ROUND FOR THE RIGHT 10 DIGITS;
        R3 => R4 MASKOFF #FFF8 + '0' => BPUSH(R0);
        R2 LONGSHIFT -3;
    END;
COMMENT LEFTMOST DIGIT CAN ONLY BE 0-3;
R3 MASKOFF #FFFC + '0' => BPUSH(R0);
RESTORE;
END;

A routine to convert a sequence of up to 11 ASCII octal digits terminated by a non-digit at the location pointed to by R0 into a 32-bit two's complement binary number stored at two words pointed to by R1, would be:

GLOBAL PROCEDURE DWOCTBIN;
BEGIN
EXTERNAL PROCEDURE SAVE(R5), RESTORE(R5);
SAVE;
0 => R2 => R3;    COMMENT CLEAR THE WORKING REGISTERS;
FOR R5 FROM 11 STEP -1 DOWNTO 1
   WHILE BPOP(R0)==R4-'0' >= 0 & R4 <= 7 DO
      BEGIN
        R2 LONGSHIFT 3; R3 MASKON R4;
      END;
      R2 => POP(R1); R3 => M1;    COMMENT STORE THE RESULT;
    RESTORE;
END;

B) Multiplication

In PL-11 the multiplication operation is indicated by the use of the dyadic operator * (asterisk). The left operand of this operator must be an even register (R0, R2, or R4), but the right operand can be any PL-11 INTEGER or LOGICAL operand (i.e. a register, a cell in storage, a cell on the stack, etc.). The result of the multiplication is a 32-bit two's complement product. The low-order 16 bits of this product are in the odd register associated with the even register specified as the left operand. The high-order 16 bits of the product are in the even register itself and are considered the RESULTS of the operation. (It is unfortunate that DEC choose to leave the low-order bits in the "wrong" register, since the results in the even register are virtually useless as a 16-bit quantity.)

For example, to multiply A by B we can write:

    A => R0 * B;

The high-order 16 bits of the product are left in R0, the low-order 16 bits in R1. To store the low-order 16 bits in C, we must write:

    A => R0 * B; R1 => C;

The carry bit is set on by the * operation if the result is outside the range (-32768, +32767), thereby indicating that the high-order word is significant.
C) Division

In PL-11 the division operation is indicated by the use of the dyadic operator / (slash). (For compatibility with the model 20, the operator DIV is treated identically to the slash / on the models 40 and 45.) The left operand of this operator must be an even register (R0, R2, or R4), but the right operand can be any PL-11 integer or logical operand. The even register specified as the left operand and its associated odd register are treated as a 32-bit two's complement dividend with the high-order 16-bits in the even register. After division by the divisor specified as the right operand, the even register will have the 16-bit quotient and the associated odd register will have the 16-bit remainder. The quotient is considered the result of this operation.

For example, after multiplying A by B, to divide by C, we can write:

\[ A \Rightarrow R0 \ast B / C; \]

Since the multiplication produces a 32-bit product in R0 and R1, that register pair already contains the 32-bit dividend for the divide operation. The result of this divide is a 16-bit quotient in register R0 and a 16-bit remainder in R1. To store the quotient in D, we can write:

\[ A \Rightarrow R0 \ast B / C \Rightarrow D; \]

Second example: Suppose we wish simply to divide A by B. We have the problem of first making the 16-bit A into a 32-bit dividend. There are several ways of doing this:

i) If A is known to be positive, we can simply load it into the odd register and clear the even register to get a 32-bit positive number for division. This can be written as:

\[ A \Rightarrow R3; 0 \Rightarrow R2 / B; \]

The result is a 16-bit quotient in R2 and a 16-bit remainder in R3.

ii) In most cases A can be either positive or negative, and the way to make a 16-bit two's complement number into a 32-bit two's complement number, is simply to propagate the sign bit through the high-order 16 bits. This can be done in two ways:

a) Using the SXT instruction (which is provided on the models 40 and 45 especially for this type of operation)

\[ A \Rightarrow R3; \text{MINUS} \Rightarrow R2 / B; \]

b) Using the long arithmetic shift instruction

\[ A \Rightarrow R2 \text{LONGSHIFT} -16 / B; \]

Both methods produce the same results: a 16-bit quotient in R2 and a 16-bit remainder in R3. Case (a) is significantly faster as it utilizes a special feature of the PDP-11 models 40 and 45 hardware. Case (b) is more similar to the methods necessary on other machines.

Case (a) operates as follows: A \( \Rightarrow R5 \) loads the odd (low-order) register R3 with the 16-bit number we wish to divide. In addition, the condition code bits are set as a result of this move: ZERO is set to 1 if \( A = 0 \), otherwise ZERO is set to 0; MINUS is set to 1 if \( A < 0 \), otherwise MINUS is set to 0. Therefore the instruction: MINUS \( \Rightarrow R2 \); sets R2 to all ones or all zeros, depending on the state of the MINUS bit, which in fact corresponds to the sign of A. Therefore the sign of A has been duplicated into all 16 bits of R2, and the R2, R3 register pair now contains a 32-bit two's complement representation of A ready for division by B.

Case (b) operates as follows: A \( \Rightarrow R2 \) loads the even (high-order) register with the 16-bit number we wish to divide. The LONGSHIFT operator treats an even/odd register pair as a 32-bit 'register', so that R2 LONGSHIFT -16 shifts the 32 bits in register pair R2 (high-order), R3 (low-order) right 16 bits. Bits shifted off the right end are lost, and bits entering on the left are set equal to the sign bit of the odd register. A shift of 16 effectively moves A from R2 into R3 while propagating its sign through R2. Therefore we are left with a 32-bit two's complement representation of A, ready for division by B.
D) More examples of * and / on the models 40 and 45

i) Computing an average

It is very often necessary to compute the average value of a set of numbers, e.g.

$$\bar{x} = \left( \frac{\sum_{i=1}^{N} x_i}{N} \right).$$

An important consideration in this computation is the fact that even though each of the values "X(I)" may only be a 16-bit number, the sum of all N of them may require more than 16 bits. This can easily happen if all "X(I)" are positive and N is large. Therefore we must perform the summation so as to keep a 32-bit result, which is very convenient for the final division by N which requires a 32-bit dividend.

Assume that the integer array X contains the N values to be used. We will keep a 32-bit running total in register R2 (high-order) and R3 (low-order). The average value will be stored in XBAR:

```assembly
0 => R2 => R3;  COMMENT START THE SUMMATION AT 0;
REF(X) => R4;   COMMENT R4 WILL POINT TO THE NEXT ELEMENT
               IN ARRAY X;
FOR R5 FROM N STEP -1 DOWNTO 1 DO
  BEGIN
    R3 + POP(R4);  COMMENT ADD "X(I)" TO THE LOW-ORDER SUM;
    R2 + CARRY;    COMMENT R2 KEEPS THE HIGH-ORDER SUM;
  END;
R2 / N => XBAR;  COMMENT COMPUTE THE MEAN AND STORE IN XBAR;
```

ii) Computing a weighted average

Another frequent computation is a weighted average:

$$\bar{x} = \left( \frac{\sum_{i=1}^{N} f_i x_i}{\sum_{i=1}^{N} f_i} \right).$$

We assume that integer array X contains the N values (as before), and integer array F contains the N weights. In addition, we assume that

$$\sum_{i=1}^{N} f_i$$

can be represented as a 16-bit integer, although

$$\sum_{i=1}^{N} f_i x_i$$

will require 32 bits. We will keep the 32-bit running total in register R2 (high-order) and R3 (low-order) and the 16-bit running total in register R4. The result is stored in XBAR:

```assembly
0 => R2 => R3 => R4;  COMMENT CLEAR ALL THE TOTALS;
FOR R5 FROM N => R5+N-2 STEP -2 DOWNTO 0 DO
  BEGIN
    F(R5) => R0;    COMMENT GET THE NEXT WEIGHT INTO R0;
    R4 + R0;        COMMENT SUM THE WEIGHTS IN R4;
    R0 * X(R5);     COMMENT COMPUTE F(I)*X(I) IN R1, R0;
    R3 + R1;        COMMENT LOW-ORDER TOTAL IN R3;
    R2 + CARRY + R0; COMMENT HIGH-ORDER TOTAL IN R2;
  END;
R2 / R4 => XBAR;  COMMENT COMPUTE THE MEAN, STORE IN XBAR;
```
E) Remainders (modulo)

The divide operation produces both a quotient and a remainder. In order to be able to operate directly on the remainder of the operation, the PL-11 dyadic operator REM can be used. The left operand must be an odd non-byte register, but the right operand can be any PL-11 INTEGER or LOGICAL operand. The specified register and its associated even register are treated as a 32-bit two complement dividend with the low-order 16 bits in the specified odd register. After the division the 16-bit quotient is in the associated even register, and the odd register will contain the 16-bit remainder, which is considered the result of this operation. For example, to find the value of A modulo B and store it in C, we can write:

\[ A \Rightarrow R1; \text{MINUS} \Rightarrow R0; R1 \text{ REM } B \Rightarrow C; \]

2.4.3 Operations on odd non-byte registers

A) MULT---single register multiplication

It is possible to perform a single-precision multiplication as a "special case" of the hardware of the models 40 and 45 with the PL-11 dyadic operator MULT. The left operand of this operator must be an odd non-byte register (R1, R3, or R5), but the right operand can be any PL-11 INTEGER or LOGICAL operand. The result is a 16-bit two complement product in the specified odd register. The associated even register is unchanged by this operation.

For example, to multiply A by B in a single word, we can write:

\[ A \Rightarrow R1 \text{ MULT } B; \]

If the result of this operation requires more than 16 bits (which can easily happen if A and B are large), only the low-order 16 bits are retained. The rest is lost although the carry bit is set if the result is outside the range (-32768, +32767).

B) ROTATE---circular right shift of a single register

It is possible to perform a circular right shift (or "rotation") N bits as a "special case" of the hardware of the models 40 and 45 with the PL-11 dyadic operator ROTATE. The left operand of this operator must be an odd register (R1, R3, or R5), and the right operand can be any PL-11 INTEGER or LOGICAL operand provided that the low-order 6 bits of this have a value N that is in the range (-32, 0). The 16 bits of the specified register will be shifted in a circular fashion to the right by \(|N|\) bits. It is not possible to shift to the left in a circular fashion, because if N has a positive value, this shift performs identically with the SHIFIA operator. It is most unfortunate that DEC have resorted to such special case bit-pushing in this extension, when they had so carefully avoided it in the previous PDP-11 architecture. Evidently they still allow "bit-minded" hardware types to destroy a basically sound architecture. An educated guess is that this extension was not made by the original designers, and may even have not been known to them.

2.5 Bit operators

Any combination of bits may be set or cleared by use of the reserved word PL-11 SET and CLEAR statements. Thus, for example,

\[ \text{SET (CARRY, OVERFLOW)}; \]

sets the hardware condition code bits CARRY and OVERFLOW. The code generated to do this uses instructions which operate on these bits. Also, if we have

\[ \text{BIT } A,B,C,D,E; \]

then

\[ \text{SET (A,D)}; \]

sets to 1 the bits A and D declared above. The other bits in the same byte are unchanged. The code generated to do this is the same as for the MASKON operator in PL-11.
CLEAR (ZERO);
clears the hardware ZERO bit in the condition code.

CLEAR (C);
clears to 0 the bit C declared above. The other bits are unchanged. The code generated to
do this is the same as for the P L-11 MASKOFF operator.

Since each SET and CLEAR statement in P L-11 generates a single machine instruction,
the bits to be SET or CLEARed must have the same byte address. Therefore the following
statement is illegal:

SET (A,CARRY);
since A is a bit in memory, and CARRY is a hardware condition code bit. Similarly, if we
have the declarations

BIT X,Y;
BYTE R;
BIT Z;

three bytes will be allocated -- the first byte containing bits X and Y, the second byte
having the name R, and the third byte containing bit Z. Therefore bit Z is at a different
memory address than bits X and Y, and hence cannot be combined with X or Y in a single
operation.

CLEAR (X,Z);
will produce a compilation error -- it must be written as two separate statements:

CLEAR (X); CLEAR (Z);
because the hardware requires two separate instructions if the bits are at different ad-
dresses.

Suppose however we had declarations in the following order:

BIT S,T;
BIT V;
BYTE P;

then bits S, T, and V would be consecutive bits in the same byte, and the following would
be acceptable in P L-11:

CLEAR (S,V);

The difference between this case and the first case should be obvious -- when a bit is
to be allocated, the compiler always allocates the next unused bit after the previous de-
claration. In the first case, the declaration preceding BIT Z caused a full byte to be
allocated to R. Therefore, Z is the first bit of the next byte. In the second case, the
declaration preceding BIT V allocated a bit to S and a bit to T. Therefore V is just the
next unused bit, which happens to be in the same byte.

It should be pointed out that the bit selection mechanism can be done symbolically by
simply declaring a few compile-time constants. For example, if we have

EQUATE SIGN SYN 15, ODD SYN 0, BIT2 SYN 2;

we can then have declarations such as:

BIT Q SYN SIGN OF CCR,
P SYN BIT2 OF MEMORY(44),
FLAG SYN BIT2 OF AA;
It is not necessary to give synonym names to all bits that are to be set, cleared, or tested, since the bit selection expression can be written anywhere a bit variable can be written. For example:

\[
\text{SET (Q); SET (SIGN OF CCR); SET (15 OF CCR)};
\]

are all identical and all equally acceptable. Similarly, bit variables and bit expressions can be mixed, provided they refer to the same memory address:

\[
\text{CLEAR (Q, BIT2 OF CCR, 13 OF CCR)};
\]

The expression before the 'OF' must always be a constant that can be evaluated at compile-time, for the same reasons as mentioned before concerning bit arrays (i.e. lack of hardware bit selection with an index register).

3. **Expression Evaluation**

The manner in which expressions are written and evaluated in PL-11 is perhaps the most distinguished aspect of the language, and is significantly different from that of any other programming language. Undoubtedly it will be a source for error by first-time users accustomed to programming in FORTRAN or ALGOL, but will be appreciated by those accustomed to assembly language as a natural method of expressing the instruction sequences typical of assembly language programming.

The most important concept is that expressions are evaluated in strictly left to right order. There is no hierarchy of operators: they all have equal precedence, and parentheses cannot be used to group subexpressions as in FORTRAN or ALGOL. (Both operator hierarchy and parenthesization require temporary storage. Since the whole philosophy of the language is to require the programmer to allocate all storage, any intermediate results must be explicitly programmed for.) The only use of parentheses is for subscripts, stack references, and indirect references as explained in Section V.1 above. Dyadic operators are written in the normal "infix" notation, but monadic operators are written in "postfix" (i.e. after the operand) instead of the normal "prefix" (i.e. before the operand). Finally, an expression must be evaluated in either a hardware register or a cell of storage that will hold the result. All PDP-11 hardware instructions require that the result replaces one of the operands. Therefore an expression in PL-11 will be evaluated in the cell or register defined by the leftmost operand in the expression, provided no move operations occur in the expression. Moves are discussed after the following examples.

Examples:

- \(A+1;\) add 1 to value of \(A\), result in \(A\).
- \(A+B+C;\) add \(B\) to value of \(A\), result in \(A\);
  then add \(C\) to that value of \(A\),
  result in \(A\).
- \(A-B(R4)-10\) subtract \(B(R4)\) from value of \(A\), result in \(A\); then subtract 10 from that value,
  result in \(A\).
- \(A\); twos complement value of \(A\), result in \(A\).
- \(A\, \text{ABS};\) absolute value of \(A\), result in \(A\).
- \(A-\,\,\, \text{-B ABS SLA } +C;\) negate \(A\), result in \(A\);
  subtract \(B\) from that value, result in \(A\); take absolute value of that value,
  result in \(A\); shift that value left by 1,
  result in \(A\); add \(C\) to that value,
  result in \(A\).
Note that any expression involving more than one operator can be written as a sequence of expressions containing only one operator. For example, the following is equivalent to the last example: \( A; A + B; A \text{ ABS}; A \text{ SLA}; A + C; \)

The move operator \( \Rightarrow \) is similar to any other dyadic operator, such as + or -, except that in addition to moving a value into the right operand, it also specifies that the register or storage cell on the right of the move operator will be used to evaluate the remainder of the expressions (up to another move).

Examples:

\[
A \Rightarrow B; \quad \text{move the value of } A \text{ into } B, \\
\text{result in } B.
\]

\[
A + B \Rightarrow C; \quad \text{add } B \text{ to value of } A, \text{ result in } A; \\
\text{move } A \text{ to } C, \quad \text{result in } C.
\]

Notice that in \( A + B \Rightarrow C; \) \( A \) and \( C \) will always have the same final value, since this expression is identical to the expression sequence \( A + B; A \Rightarrow C; \) However in \( A \Rightarrow C + B; \) which is equivalent to \( A \Rightarrow C; C + B; \) the final values of \( A \) and \( C \) will be different (unless \( B \) is zero). In this case the value of \( A \) is unchanged.

\[
A + R4 - \text{SRC} \Rightarrow R1 + 10 \text{ ABS}; \quad \text{Add value in } R4 \text{ to } A, \text{ result in } A; \text{ then} \\
\text{negate that value, result in } A; \text{ then} \\
\text{shift that value right circular by } 1, \text{ result in } A; \text{ then move that value } R1, \text{ result in } R1; \text{ then add } 10 \text{ to that value, result in } R1; \text{ then take the absolute value of that value, result in } R1.
\]

The following longer examples are given with an equivalent sequence of one operator expressions (each of which is a single PDP-11 machine instruction (except for \( \text{ABS} \))):

i) \[ 1005 \Rightarrow R2 + R1 \text{ MASKON B MASKOFF} \#F1A \text{ SRA SRA} + \text{POP(R4)}; \]
\[ 1005 \Rightarrow R2; R2\text{+R1; R2 MASKON B; R2 MASKOFF} \#F1A; R2 \text{ SRA; R2 SRA; R2+POP(R4)}; \]

ii) \[ 0 \Rightarrow R4 + \text{IND(R1)} \Rightarrow R5 \text{ SRA SLC} \Rightarrow B \Rightarrow \text{PUSH(R1)}; \]
\[ 0 \Rightarrow R4; R4\text{+IND(R1); R4} \Rightarrow R5; R5 \text{ SRA; R5 SLC; R5} \Rightarrow B; B \Rightarrow \text{PUSH(R1)}; \]

iii) \[ -15 \Rightarrow R4 + A - B \text{ SWAP} + 23 \text{ REM C(R1) SHL 12}; \]
\[ -15 \Rightarrow R4; R4+A; R4-; R4-B; R4 \text{ SWAP; R4*23; R4 REM C(R1); R4 SHL 12}; \]

Notes

1) The minus sign: In PL-11 the minus sign (-) serves three functions:

As a prefix to decimal literals, indicating a minus value, as -23 -4105 -9994.

In infix notation, as the dyadic subtract operator, e.g. \( A - B; \) \( \text{PUSH(R4)} - \text{POP(R1)}; \)

R5 - IND(R3);

In postfix notation as the monadic negation operator, e.g. \( A -; R2-; C(R3*2)-; \)

In most cases there is no confusion as to which is intended. Clearly a) \( A * -22; \) means multiply \( A \) by \(-22; \) b) \( B^- -99; \) means negate \( B \), then add \(-99, \) and c) \( -1 \Rightarrow R4 \text{ MASKOFF} -1024; \) means move \(-1 \) to \( R4 \), then \( \text{MASKOFF} -1024. \) However, the following is an ambiguous case:

\[ B - 45; \]
Does this mean subtract -45 from B? Or does it mean negate B, then subtract 45? In PL-11, the second interpretation will be used, since subtracting -45 from B is far more likely to be written as B+45; which is also easier to understand! Hence we have that

\[ B - 45; \] is equivalent to \[ B; B - 45; \]

2) A literal can be the leftmost operand in an expression only if a) the first operator is a move or b) the next operand is also a literal, in which case the operation is performed at compile-time.

For example, the following examples are legal:

\[ -10 + #F19 -22 \Rightarrow R4; \quad -19 \Rightarrow R1; \quad 0 \Rightarrow A; \]

the following are illegal:

\[ #F1 - R(R4); \quad 10 * B \Rightarrow R1; \quad -10 * A; \]

This restriction is due to the convention that expressions are evaluated in the storage cell or register defined by the leftmost operand. Literals do not occupy storage, and thus there would be no place to put the result of the operation.

4. THE CONCEPT OF "TYPE" IN EXPRESSIONS

In PL-11, all variables and registers have one of three possible types: BYTE, INTEGER, and LOGICAL. The PDP-11 hardware imposes certain restrictions on the use of these types in expressions. These are discussed next.

4.1 INTEGER and LOGICAL types

In the evaluation of an expression, there is no difference between INTEGER and LOGICAL types (the difference appears in comparisons -- see Section VI). They are both considered 16-bit words, and all operations mentioned in Section V.2 above can be used on them. The type of the result of an operation will obviously be the type associated with the variable or register that contains the result. In all arithmetic operations, INTEGER and LOGICAL values are treated as signed quantities, in which the leftmost bit in the word is the sign, and the remaining 15 bits comprise the magnitude, in two complement notation. In the logical operations (COMP, SWAP, MASKON, MASKOFF, SHIFTL, and SIL), these two types are treated as unsigned 16-bit quantities.

4.2 BYTE type

The type BYTE corresponds to an 8-bit byte in the PDP-11 hardware. Not all the operations listed in Section V.2 above can be used with byte type, since the hardware is not capable of performing them. The only monadic that cannot be used is SWAP, since by definition SWAP exchanges the two bytes in a word, and obviously must therefore apply only to word operands. In the dyadic however, only the operators MASKON, MASKOFF, and \( \Rightarrow \) (move) can be used with bytes. The only arithmetic that can be performed on bytes is to add or subtract 1 (i.e., if A and B are bytes, A+1 and B+1 are legal, but A+B and A-B are not). None of the extended arithmetic unit operations can be performed on bytes -- the hardware requires word operands. This means that in general bytes will be useful only for bit setting and testing, and for character manipulation. Literals used in operations with BYTE registers and variables must be representable in 3 bits on the PDP-11 (i.e., one-character strings, at most two hex digits, or a decimal number in the range \([-128, +127]\]).

4.3 BYTE registers

The pre-declared names R0, R1, ..., R5, corresponding to the hardware 16-bit registers 0, 1, ..., 5, have type INTEGER in PL-11. It is possible, however, to declare and use BYTE registers which are synonymous with one of the integer registers. For example:

\( \text{BYTE \quad Y1 \ \text{SYN} \ R1, \ Y2 \ \text{SYN} \ R2; } \)
will cause identifiers Y1 and Y2 to refer to registers 1 and 2, respectively, but to have type BYTE. The only operations which can be performed on Y1 and Y2 are those which can apply to BYTE types in general, namely all monadics except SWAP, and the dyadics MASKON, MASKOFF, =>, Add 1, Subtract 1. All other operations are illegal for any BYTE, whether variable or register.

When used with BYTE registers, all the allowed operations, except move (=>), use as their operand the low-order 8-bit byte of the 16-bit register. The high-order byte is unchanged by the operation. There is, however, an anomaly in the use of the move operator with BYTE registers, and that is that when a byte value is moved into a BYTE register it will be put in the low-order byte, and the first bit of this byte will be propagated through all 8 bits of the high-order byte of the register. This means that the operation to move values into a BYTE register considers a byte to be a signed 8-bit number (leading sign bit, 7-bit magnitude) in twos complement notation, and the 8 high-order bits of the register are set to the same value as the sign bit, thereby implicitly converting an 8-bit signed number into a 16-bit signed number. There is no way of moving a value into a register without affecting the high-order byte of the register -- the sign propagation is part of the PDP-11 hardware. In PL/I this is true for literals as well as variables that are moved into a BYTE register (i.e. 0 => Y1 clears the whole register).

4.4 Mixed types in expressions

As stated previously, INTEGER and LOGICAL types are indistinguishable in expressions, and hence can be used in any combination. In general, BYTE types cannot be mixed with INTEGER or LOGICAL types in an expression. There are only two occasions when mixing is allowed; these are the following:

i) BYTE types can be moved into an INTEGER or LOGICAL register, in which case the sign propagation explained in Section V.4.3 above will occur;

ii) INTEGER or LOGICAL registers can be moved into BYTE types. If they are moved into a BYTE register, the sign propagation explained above will occur. If they are moved into a BYTE variable, the low-order byte of the register will be stored. Unfortunately, no overflow indication is given if the value of the register is not in the range [-128, +127]. This is one of the few weaknesses of the PDP-11 hardware design, since a hardware overflow indication of this type would appear to be a "natural" complement of the hardware sign propagation when a byte value is loaded into a register.

Examples:

Assume the declarations: BYTE Y1 SYN R1;
                   BYTE X; INTEGER A;

If R1 contains #ABCD, X contains #86, and A contains #1234, then:

    after X => R1; R1 will contain #FF86, X is unchanged.
    after X => Y1; R1 will contain #FF86, X is unchanged.
    after R1 => X; X will contain #ABCD, R1 is unchanged.
    after Y1 => X; X will contain #ABCD, R1 is unchanged.
    after A => R1; R1 will contain #1234, A is unchanged.
    after R1 => A; A will contain #ABCD, R1 is unchanged.
      A => Y1; and Y1 => A; are both illegal.
    after Y1 => R1; R1 will contain #FFCD.
    after R1 => Y1; R1 will contain #FFCD.
    after 1 => Y1; R1 will contain #0001.
    after 0 => Y1; R1 will contain #0000.
    after -1 => Y1; R1 will contain #FFFF.

In general, BIT types cannot be mixed with other types, since the hardware instructions necessary to do the operations do not exist. Thus:
BIT A; INTEGER B;
A + B;

is illegal. Also,

B + OVERFLOW;

is illegal. There are two cases, however, where a type BIT may be mixed with an INTEGER. The first of these is:

MINUS =⇒ A;

which we have already seen. This is for the models 40 and 45 only. The other example concerns the CARRY bit.

In order to perform multiple-precision operations it is possible to add and subtract the CARRY bit with an ordinary operand such that if the CARRY bit is set, a 1 is added to or subtracted from that operand, and if the CARRY bit is clear, a 0 is added or subtracted. In PL-11 this is written as:

(operand) + CARRY
(operand) - CARRY

These operations each generate a single PDP-11 instruction and can only be done with the CARRY bit (for example, (operand) + OVERFLOW is illegal).

For example, if AHIGH and ALOW are the two 16-bit halves of a 32-bit number, and BHIGH and BLOW are the halves of another 32-bit number, then to add B to A we write:

ALOW + BLOW;  AHIGH + CARRY + BHIGH;

and to subtract B from A we write:

ALOW - BLOW;  AHIGH - CARRY - BHIGH;

If we have N-element integer arrays X and Y to represent multiple-precision numbers of 2N bytes (16*N bits) then adding X to Y can be done as:

FOR R1 FROM N2 STEP -2 DOWNTO 4 DO
BEGIN
  Y(R1-2) + X(R1-2);
  Y(R1-4) + CARRY;
END;

Y + X;

and for subtracting X from Y:

FOR R1 FROM N2 STEP -2 DOWNTO 4 DO
BEGIN
  Y(R1-2) - X(R1-2);
  Y(R1-4) - CARRY;
  Y - X;

where we have assumed the declarations:

ARRAY N INTEGER X,Y;
EQUATE N2 SYN 2*N;
5. **LITERAL EXPRESSIONS**

An expression that consists of only literal operands is called a literal expression, and is evaluated at compile-time to produce a single value which is then utilized by the compiler. All monadic operators except SLC and SRC can be used in literal expressions. The dyadics which may be used are: `+`, `-*`, `/`, `REM`, `MASKON`, `MASKOFF`, `SHL`, and `SHA` [that is, all "standard" dyadics except move (=>), and all "macro"-type extended arithmetic unit dyadics]. The operands can be any decimal, hexadecimal, or octal literal, or a string of one or two characters (longer strings can only be used in initialization). Address literals require special consideration (see below). Literal expressions, like all expressions in PL-11, are evaluated in strict left-to-right order, with no operator precedence and with no parenthesization allowed. All intermediate values, as well as the final result and the operand values, must be in the range `[-32768, +32767]`, or a compilation error will result.

**Examples:**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Compilation equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>10 + 2 * 4</code></td>
<td>48</td>
</tr>
<tr>
<td><code>#1F SHL 3 MASKOFF #30</code></td>
<td>C8</td>
</tr>
<tr>
<td><code>-128 SRA - / 10 + 4 * 3</code></td>
<td>30</td>
</tr>
<tr>
<td><code>&quot;9' MASKOFF #P0 REM 4 * -2 ABS</code></td>
<td>2</td>
</tr>
</tbody>
</table>

If the leftmost part of an expression is a literal subexpression, this will also be evaluated at compile-time. For example:

```
2 * 4 + 1 + R1 - 3 + 10  is equivalent to  9 + R1 - 3 + 10
```

6. **ADDRESS LITERALS IN EXPRESSIONS**

Address literals are given only a relative value at compile-time, since the actual address is not known until load time, and may vary from one load to another. Therefore a literal expression anywhere in PL-11 can contain at most one address literal additively, or any number of differences of address literals, of the form `REF(P) - REF(Q)`, since the difference of two relative values is always constant.

**Examples:**

<table>
<thead>
<tr>
<th>Legal</th>
<th>Illegal</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>REF(P) + 10</code></td>
<td><code>REF(P) * 10</code></td>
</tr>
<tr>
<td><code>REF(P) - REF(Q)</code></td>
<td><code>REF(P) + REF(Q)</code></td>
</tr>
<tr>
<td><code>#FF + REF(P)</code></td>
<td><code>#FF - REF(P)</code></td>
</tr>
<tr>
<td><code>REF(P(10))</code></td>
<td><code>REF(P) - REF(Q) - REF(R)</code></td>
</tr>
<tr>
<td><code>REF(P) - REF(Q) + REF(R) - REF(S)</code></td>
<td><code>REF(P) + REF(R) - REF(Q) - REF(S)</code></td>
</tr>
<tr>
<td><code>REF(P) + #A1C - REF(Q)</code></td>
<td><code>#A1C - REF(Q) + REF(P)</code></td>
</tr>
<tr>
<td><code>REF(P)</code></td>
<td><code>REF(P)</code></td>
</tr>
</tbody>
</table>

7. **EXPRESSIONS WITH NO OPERATIONS**

A single operand, without any associated operators, is a "degenerate" case of an expression. If this appears as a statement, the compiler will generate a "test-against-zero" instruction, in order to cause a reference to that operand and to set the condition code according to the value produced by the reference. For example:

```
A;  simply tests the value of A
B(R1+2); tests the value of B(R1+2)
R3;  tests the contents of R3
```

This is sometimes useful when the operand reference causes side effects, as in stack operations:

```
POP(R1);  tests the top item of the stack and
          then removes it from the stack by
          adding 2 to the stack pointer R1.
```
PUSH(R2); moves the stack pointer R2 down by subtracting 2, then tests the item pointed to by R2.

However, a literal expression which is used as a complete statement will generate an error, since it is evaluated at compile-time and therefore generates no run-time code (i.e. 10 * 2 - 4; is an illegal statement).

8. USE OF CERTAIN PRE-DECLARED NAMES

Appendix C contains a list of pre-declared names and their meaning in PL-11. A brief explanation may be useful to clarify the utility of some of these names.

8.1 "Absolute" addressing

Variables that are assigned storage in PL-11 declarations are given relative addresses. This is to say that their location in storage is not determined until the program is linkage edited, and the absolute address assigned by the linkage editor may vary from one loading to the next. Only the relative storage positions of variables with respect to other variables in the same program will remain constant. However, PL-11 contains a general mechanism to access any word (or byte) at an arbitrary absolute location in memory. This is the function of the pre-declared name "MEMORY". For example, a reference to MEMORY(34) is a reference to the integer word at absolute address 34. In general, whatever the value of the subscript of array MEMORY, that subscript will be the absolute address that is used in the storage reference. Although MEMORY is pre-declared with type INTEGER, any byte can be accessed by simply declaring, for example:

BYTE BMEM SYN MEMORY;

A reference to BMEM(31) will then refer to the BYTE at absolute address 31 of the PDP-11 memory.

8.2 "Pointer register" addressing

A very common programming technique is to load the address of a variable or an array into a register, and then use this register as a pointer with which to access memory. For example, if we declare

ARRAY 10 INTEGER A;

and then execute

REF(A) => R1;

then at run-time, R1 will contain the absolute address of the first element of array A. To refer to this element, we can write either

IND(R1) (i.e. refer indirectly through R1)

or

MEMORY(R1) (i.e. refer to absolute address contained in R1).

However, to refer to the second element, we must use

MEMORY(R1+2)

and to the third,

MEMORY(R1+4), etc.

In order to save writing, and also to make the use of this "pointer" technique clearer, the names M0, M1, ..., MS have been pre-declared in PL-11 to have the meaning MEMORY(RO), MEMORY(R1), ..., MEMORY(RS), respectively. In the above example this means that we can write simply M1, M2, ..., M10 to refer to the first element of A, M1(2) to refer to the second, M1(4) to the third, etc. In a programming example, to clear all 10 elements of array A to 0, we might write:
FOR R1 FROM REF(A) STEP 2 UPTO REF(A(18)) DO 0 \Rightarrow M1;

In this case, R1 always points to the next item in the array to be cleared.

The type of M0, M1, ..., M5 is pre-declared to be INTEGER, meaning that the 16-bit word at that absolute address is being pointed to. Since this concept of a register "pointer" is very useful, PL-11 has also the pre-declared name B0, B1, ..., B5 to refer to the BYTE instead of the INTEGER at that absolute address. For example, if we declared

```
ARRAY 10 BYTE X;
```

then to clear all 10 elements to 0 we could write:

```
FOR R1 FROM REF(X) STEP 1 UPTO REF(X(9)) DO 0 \Rightarrow B1;
```

*Warning:*

Since the names M0, M1, ..., M5 and B0, B1, ..., B5 already imply the use of a register, any subscript used with these names cannot include a register. Therefore M1(R2) and B0(R3+4) for example are illegal, since they would require double indexing in an instruction, a feature not present in the PDP-11 hardware.
Section VI. CONDITIONS IN PL-11

Conditions are the PL-11 mechanism for testing and comparing items in the PDP-11. Conditions can only be used as part of statements that alter the flow of control based on the results of the test or comparison. In PL-11, this means a condition can only follow either the word "IF" in an IF statement, or the word "WHILE" in an iteration statement. The result of a condition is always either "true" or "false".

Conditions can be classified into two categories: simple and compound.

1. SIMPLE CONDITIONS

There are three kinds of simple conditions: comparison, bit tests, and status tests.

1.1 Comparisons

1.1.1 Form

Comparisons are by far the most commonly used type of condition in PL-11. Their form is:

\[
\text{<any expression> <compare-op> <operand>}
\]

where (any expression) is any non-literal data expression (see Section V on Expressions), and (compare-op) is one of the following comparison operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
<tr>
<td>=</td>
<td>equal to</td>
</tr>
<tr>
<td>/=</td>
<td>not equal to</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
</tbody>
</table>

The operators >, /=, <= are composed of two adjacent card characters, but are treated as a single operator by the syntax. They must, however, be punched in the correct sequence (i.e. <= and /= are not legal, whereas => is the arithmetic "move" operator).

Conditions are evaluated in the following manner:

i) The expression on the left is evaluated, if necessary (see Section V on Expressions).

ii) The expression is compared with the operand on the right.

iii) The result is "true" if the PDP-11 condition code set by the comparison satisfies the meaning of the operator; otherwise the result is "false".

iv) This result is always "interpreted" by a conditional branch instruction to alter the flow of control as defined by the IF or WHILE clause in which the condition appears.
Examples:

\[
\begin{align*}
A & \geq B & C & > 0 & D & = E \\
A+B(6) & \neq -23 & \text{POP(R4)} & \leq 0 & \text{PUSH(R2)} & < \text{IND(POP(R2))}
\end{align*}
\]

The following are illegal comparisons:

- \(A \geq B+52\) expressions not allowed on the right
- \(A =/ B\) illegal compare operator
- \(0 > B\) as with expressions, literals cannot appear on the left of a comparison.

1.1.2 Determining "type" in comparisons

As was generally true of expressions, items of type BYTE cannot be compared with items of type INTEGER or LOGICAL, owing to the fact that BYTES are 8-bit quantities, and both INTEGER and LOGICAL are 16-bit quantities. However, INTEGER and LOGICAL types can be mixed in a comparison and this is the only place in PL-11 where there is difference between an INTEGER item and a LOGICAL item. INTEGERs are considered to be 16-bit signed arithmetic numbers (15 bits plus leading sign bit), whereas LOGICALs are considered to be 16-bit unsigned logical bit patterns. This means that the ordering of INTEGERS is different from the ordering on LOGICALs (see pp. 23–24 of the PDP-11 Handbook for more details).

In PL-11, a comparison between two LOGICALs or between a LOGICAL and a literal will be considered unsigned. All other comparisons are considered signed. Thus a comparison between an INTEGER and a LOGICAL is the same as a comparison between two INTEGERS (namely, signed).

Note:

See also Appendix G on compiler optimization.

1.2 Bit test conditions

1.2.1 Using BIT variables

In a bit test using a BIT variable, the condition is simply the BIT itself; for example:

\[
\text{BIT X;}
\]

Then a condition involving X is "true" if the bit is set, and "false" if the bit is not set.

1.2.2 Using non-BIT variables

The PDP-11 hardware provides the programmer with the ability to test any bit or combination of bits in a variable or register, and these tests are reflected in PL-11 in a straightforward manner. The form of a bit test is:

\[
\text{(any expression) (bit-test-op) (mask)}
\]

where (any expression) is any non-literal data expression, (bit-test-op) is one of the reserved words ANY or NONE, and (mask) is any operand that will be used as a mask (this is often a literal, but is not required to be).

Bit tests are evaluated as follows:

i) the expression on the left is evaluated, if necessary;

ii) the expression is ANDed with the mask on the right (using the BIT or BITB instruction in the PDP-11 hardware) -- neither the expression nor the mask is modified;

iii) the condition is "true" if a) the bit-test operator is NONE and the AND produces an all-zero bit pattern; or b) the bit test operator is ANY and the AND does not produce an all-zero bit pattern.

In all other cases the condition is "false".
Examples:

A ANY #10A7
POP(R4) NONE IND(R1)
B MASKOFF #F7 ANY #FE08

1.2.3 Condition code bit tests

The PDP-11 condition code consists of 4 bits which are set by the arithmetic and logical operations performed by hardware instructions. These bits can be tested in a PL-11 program by simply using the reserved word which names the bit, or the comparison operator (without operands) that represents the bit or bit combinations. The following table illustrates the correspondence between PL-11 pre-declared names (or operators) and the PDP-11 hardware bit settings:

- **ZERO**
  - true if Z bit = 1
- **CARRY**
  - true if C bit = 1
- **OVERFLOW**
  - true if V bit = 1
- **MINUS**
  - true if N bit = 1
- **=**
  - true if Z bit = 1
- **/=**
  - true if Z bit = 0
- **<**
  - true if N bit = 1 or V bit = 1, but not both
- **>=**
  - true if N bit and V bit are the same
- **<=**
  - true if Z bit = 1 or (N bit = 1 or V bit = 1, but not both)
- **>**
  - true if Z bit = 0 and (N bit and V bit are the same).

In addition, any of these status tests can be preceded by the reserved word NOT to reverse the truth of the test. For example:

- **NOT ZERO**
  - true if Z bit = 0
- **NOT CARRY**
  - true if C bit = 0
- **NOT OVERFLOW**
  - true if V bit = 0

etc.

2. Compound conditions

Compound conditions are simple conditions combined with the symbols & or !, and are called "AND-compounds" or "OR-compounds", respectively.

*Notation:*

Let C1, C2, ..., Cn be any simple conditions.

2.1 AND-compounds

The form of an AND-compound is any number of simple conditions connected together with the symbol & (ampersand):

\[ C1 \& C2 \& C3 \& \ldots \& Cn \]

The AND-compound is true if and only if all the separate simple conditions C1, C2, ..., Cn are true. The simple conditions are evaluated left to right, starting with C1, and evaluation stops as soon as a false simple condition is found. Hence C3 is evaluated only if C1 and C2 were both true, etc., and Cn is evaluated only if all previous conditions C1, C2, ..., Cn-1 were true.
Examples:

If we have the declarations

\[
\begin{align*}
\text{INTEGER A,B,C,D,E,F;} \\
\text{ARRAY 10 INTEGER Q;} \\
\text{BIT X,Y,Z;} 
\end{align*}
\]

then the following are legal "AND-compounds":

\[
\begin{align*}
A & \land B \land C & \land D & \land X & \land Y \\
Q(R1) + R4 & \geq 0 & \land \text{POP(R1)=POP(R4)} & \land \text{NOT CARRY} \\
\text{CARRY} & \land \text{NOT OVERFLOW} & \land Z & \land Y & \land X & \land E=F . 
\end{align*}
\]

2.2 OR-compounds

The form of an "OR-compound" is any number of simple conditions connected with the IBM 029 "|" (or-bar), which in the CERN-ANSI character set is the symbol "!'" (exclamation mark):

\[
C_1 \lor C_2 \lor C_3 \lor \ldots \lor C_n .
\]

The OR-compound is true if and only if any of the separate simple conditions \(C_1, C_2, \ldots, C_n\) is true. The simple conditions are evaluated left to right, starting with \(C_1\), and evaluation stops as soon as a true simple condition is found. Hence \(C_2\) is evaluated only if \(C_1\) is false, \(C_3\) is evaluated only if \(C_1\) and \(C_2\) were both false, etc., and \(C_n\) is evaluated only if all previous conditions \(C_1, C_2, \ldots, C_{n-1}\) were false.

Examples: With the same declarations as above, the following are legal "OR-compounds":

\[
\begin{align*}
A & \lor B \lor C \lor D \lor E \lor F \lor X \lor Y \\
Q(R1) + R4 & \geq 0 \lor \text{POP(R1)=POP(R4)} \lor \text{NOT CARRY} \\
\text{CARRY} \lor \text{NOT OVERFLOW} \lor Z \lor Y \lor X \lor E=F . 
\end{align*}
\]

Note:

& and ! cannot be used in the same compound. Thus the following is illegal:

\[
A \lor B \land C \land D \land E=F \land X \land Y .
\]
Section VII. CONTROL STATEMENTS IN PL-11

There are three kinds of control statements in PL-11:

IFs, GOTOs, and CASEs.

1. 'IF' STATEMENT

As in many higher-level languages, the IF statement in PL-11 is the principal statement for decision-making and altering the flow of control. There are two forms of the IF, called the complete and incomplete forms, depending on whether or not the ELSE-clause is present.

Notation:
Let S1, S2, S3 be any statements or blocks.
Let C be any conditional expression (see Section VI).

1.1 The incomplete IF statement

The basic form is:

IF C THEN S1; (Followed by the next statement S3).

This is compiled into code to do the following:

i) Evaluate the conditional expression C.

ii) If the value of C is true, control passes to statement (or block) S1. If C evaluates to false, statement (or block) S1 is skipped and control passes to S3 (the next statement).

iii) If control passed to S1 in step (ii), and there is no branch out of S1, the next statement in sequence at the completion of S1 will be S3 (the next statement after the IF).

Note:
S1 cannot be labelled, and attempts to branch to S1 (or into S1 if S1 is a block) will be detected as errors by the compiler. Branches out are of course legal.

Examples:

IF A & B & C THEN D => E;
IF SIGN OF R4 : 10 OF B THEN P => Q;
IF CARRY & NOT OVERFLOW THEN 3 => R;
IF CARRY THEN 0 => X;
IF A > B THEN 25 => C;
IF R0+4 <= 19 THEN A => B + C - 2;
IF A(4) >= B(-20) THEN
BEGIN .... END;
IF C>D & B=R3 & A(R1+2)<#5F THEN
BEGIN .... END;
IF A > 0 THEN FOR R2 FROM 1 STEP 2 UPTO 100 DO 0 => X(R2);
1.2 The complete IF statement

The basic form is:

\[
\text{IF } C \text{ THEN } S1 \text{ ELSE } S2; \quad (\text{followed by the next statement } S3).
\]

This is compiled into code to do the following:

i) Evaluate the conditional expression \( C \).

ii) If the value of \( C \) is true, control passes to statement (or block) \( S1 \). If there is no branch out of \( S1 \), the next statement in sequence at the completion of \( S1 \) will be \( S3 \) (the next statement after the IF). \( S2 \) is completely by-passed.

iii) If the value of \( C \) is false, control passes to statement (or block) \( S2 \). \( S1 \) is completely by-passed. If there is no branch out of \( S2 \), the next statement in sequence at the completion of \( S2 \) will be \( S3 \) (the next statement after the IF).

Notes:

i) Neither \( S1 \) nor \( S2 \) can be labelled, and attempts to branch to either \( S1 \) or \( S2 \) (or into either of them if they are blocks) will be detected as errors by the compiler. Branches out are of course legal.

ii) There is no \textit{statement} after \( S1 \) (before the ELSE), since the complete IF does not end at this point, but must include the ELSE and \( S2 \).

Examples:

\[
\begin{align*}
\text{IF } A > B & \quad \text{THEN } 25 \Rightarrow C \quad \text{ELSE} \quad -19 \Rightarrow C; \\
\text{IF } A \geq 0 & \quad \text{THEN } A \Rightarrow R0 \quad \text{ELSE} \quad A \Rightarrow R0 \Rightarrow; \\
\text{IF } A>0 & \quad B=0 : C=0 \quad \text{THEN} \\
& \quad \text{BEGIN} \quad \ldots \quad \text{END} \quad \text{ELSE} \quad P(R4-2) \Rightarrow R0; \\
\text{IF } A/B & \quad C=R0+4 > 0 \quad \text{THEN} \quad 0 \Rightarrow R1 \\
& \quad \text{ELSE} \quad \text{BEGIN} \quad \ldots \quad \text{END}; \\
\text{IF } R0=R4+9-A(4) & \quad > 0 \quad \text{THEN} \\
& \quad \text{BEGIN} \quad \ldots \quad \text{END} \\
& \quad \text{ELSE} \quad \text{BEGIN} \quad \ldots \quad \text{END};
\end{align*}
\]

1.3 Nesting IF statements

Since \( S1 \) and \( S2 \) can be any type of statement, they may in fact be IF statements. Usually this poses no problems, as in the following examples.

i) \[
\text{IF } A < B \quad \text{THEN} \quad \text{IF } C > D \quad \text{THEN} \quad R0 \Rightarrow R4 \Rightarrow B;
\]

which is the same as

\[
\text{IF } A<B \quad & \quad \text{AND} \quad C>D \quad \text{THEN} \quad R0 \Rightarrow R4 \Rightarrow B;
\]

ii) \[
\begin{align*}
\text{IF } A > 0 & \quad \text{THEN} \quad R0 \Rightarrow B(R3) \quad \text{ELSE} \\
\text{IF } A = 0 & \quad \text{THEN} \quad R0 \Rightarrow B(R1) \\
& \quad \text{ELSE} \quad R0 \Rightarrow B(R1) \Rightarrow B(R3);
\end{align*}
\]

However, the following statement is ambiguous without a further qualification:

\[
\text{IF } A < B \quad \text{THEN} \quad \text{IF } C > D \quad \text{THEN} \quad 1 \Rightarrow R0 \quad \text{ELSE} \quad 2 \Rightarrow R0;
\]

The problem is: with which of the two THENs should the single ELSE be paired? Although it could be paired with either of them, in PL-11 it will be paired with the second THEN. Hence the above statement would be performed as follows:
if A < B is false, the next statement after ; is done
if A < B is true and C > D is true, 1 => R0 is done
if A < B is true and C > D is false, 2 => R0 is done.

The general rule is as follows: each ELSE is paired with the rightmost unpaired THEN which precedes it. The grouping in the above statement is then:

IF A < B THEN  IF C > D THEN  1 => R0 ELSE 2 => R0;

To obtain the other grouping, a BEGIN - END must be used:

IF A < B THEN  BEGIN IF C > D THEN  1 => R0; END, ELSE 2 => R0;

Note that this is analogous to the general problem of parenthesizing expressions. In FORTRAN A + B * C is interpreted as A + (B*C), but if parentheses are used, we can also get the expression (A+B) * C. Just as parentheses can always be added to FORTRAN (but not PL-11) expressions to make the interpretation clear, so can BEGIN - END pairs be used in PL-11 IFs to make the interpretation clear. Thus the following two are identical:

IF A < B THEN BEGIN IF C > D THEN 1 => R0 ELSE 2 => R0; END;
IF A < B THEN IF C > D THEN 1 => R0 ELSE 2 => R0;

2. 'GOTO' STATEMENT

2.1 Basic form

The GOTO statement is the only means provided in PL-11 for an unstructured change in the normal flow of control from statement to statement.

If L is a label defined elsewhere (by being preaced to a statement), the form of a GOTO statement is:

GOTO L;

where GOTO is written as one word with no embedded blanks.

When executed, the next statement in sequence will not be the one following the GOTO, but will be the statement labelled L. This GOTO statement is compiled into a single unconditional branch.

Notes:

i) Owing to the block structure of PL-11, it is possible for statements in different blocks (either nested or parallel) to have the same label L. A statement GOTO L will then transfer control to the most local statement with label L (that is, to the L in the same block as the GOTO, if there is one; otherwise to the L in the next outer block, etc.). Since a label, like any other identifier, can be defined only once in a block, there is no difficulty in determining which statement will be the next in sequence.

ii) See also Appendix G on compiler optimization.

2.2 'IF' statements combined with 'GOTO' statements

A very common use of the GOTO is as part of an IF statement. Examples are:

IF A > B THEN GOTO ALPHA;
IF A <= 0 THEN GOTO B4 ELSE GOTO X295Y;
IF A=0 : B<0 : C<-19 THEN 1 => R0
   ELSE GOTO ERROR;
Notes:

i) Although useful, the need for such constructs in PL-11 programs should be much less
    than in a normal assembly language or even FORTRAN program. This is due to the fact
    that the powerful structuring facilities of the IF, CASE, and iteration statements, as
    well as the convenient local proceduring, eliminates the need for labels and explicit
    GOTOs in most programming techniques.

ii) See also Appendix G on compiler optimization.

3. 'CASE' STATEMENTS

    The IF statement, in its complete form, allows control to be transferred to one of two
    possible alternatives, depending on whether a condition is true or false. The CASE state-
    ment allows control to be transferred to one of any number of possible alternatives depending
    on the value in a register. As in the IF, when execution of the selected alternative is
    complete, control automatically passes to the next statement following the entire CASE state-
    ment.

    Notation:

    Let S1, S2, ..., Sn, Sx be any statements or blocks.
    Let E be any expression whose final evaluation occurs in an INTEGER or LOGICAL register.

    The basic form is:

    CASE E OF
    BEGIN S1; S2; ... Sn; END; (followed by the next statement Sx).

    This is compiled into code to do the following:

    i) The register specified by E is shifted left by 1, thereby doubling its original value.

    ii) If the original value was 1, control is passed to statement (or block) S1, if the ori-
        ginal value was 2, control is passed to S2, etc., up to n.

    iii) If there is no branch out of the statement (or block) selected in step 2, Sx will be
        the next statement to be executed on the completion of the selected statement.

    Warnings:

    i) The register value when control passes to the selected statement Sk is double that
given by it the programmer in E. This is due to the fact that each address in the
PDP-11 requires 2 bytes of memory. Hence the index into the branch table constructed
by the compiler must be twice the number of the desired entry.

    ii) If the value specified by E is less than 1 or greater than n, the results are undefined,
and control will be passed to some random place in memory. There is no automatic
checking -- this is the programmer's responsibility.

Examples:

    CASE R0 OF BEGIN 4 => R5; A - 2 + B; GOTO ERROR; END;
    CASE A => R4 + B - 10 OF
      BEGIN
        IF A < B THEN C => D;
        BEGIN 0 => R4; A => R5; -9 => B; END;
        C => D => R1;
        K(R1-4) => R0;
        FOR R1 FROM 0 STEP 2 UPTO 100 DO 0 => A(R1);
        GOTO L;
        0 => R1;
      END;
Notes:

i) The sequence following the word "OF" in the case statement (called the CASE-body), resembles syntactically a block. That is, it starts with the word BEGIN, ends with the word END, and contains any number of statements or sub-blocks separated by semicolons. In PL-11, all the other syntactic properties of a block are also allowed to a CASE-body. This means that

a) any of the statements S1, S2, ..., Sn and the END can be labelled. Control can therefore be transferred from one statement in the case to another, or from a statement in the case to one outside, but not from outside the case to S1, S2, ..., or Sn;

b) any number or type of declaration, including procedure declarations, can be written after the BEGIN and before the first executable statement S1. The scope of these declarations (as well as of any labels on S1 through Sn) is the CASE-body "block". These declarations are ignored, when applying the case selection rule, since the first executable statement will be selected when the value of E is 1.

ii) The "BEGIN" following the "OF" cannot be labelled.

iii) See Appendix G on compiler optimizations.
Section VIII. ITERATION IN PL-11

Iteration statements

There are four kinds of iteration statements in PL-11:

1) FOR statements;
2) WHILE statements;
3) FOR-WHILE statements;
4) "infinite-loop" DO statements.

Notation:

- \( V \) some variable or register
- \( E_1, E_2, \text{and } E_3 \) any data expressions
- \( C \) any conditional expression
- \( S_1, S_2 \) any statements or blocks.

1. THE 'FOR' STATEMENT

This is the standard loop-counting type of statement and corresponds to the DO-loop of FORTRAN. It has two different subforms, depending on whether the loop counter is increased or decreased each time around the loop.

1.1 Increasing loop-counter

The form is:

\[
\text{FOR } V \text{ FROM } E_1 \text{ STEP } E_2 \text{ UPTO } E_3 \text{ DO } S_1; \quad \text{(followed by next statement } S_2)\).
\]

This is translated by the compiler into code to do the following:

a) Evaluate \( E_1, E_2, \text{and } E_3 \), in that order. Note that only expressions containing operators need evaluation at this time.

b) Move the value of \( E_1 \) to \( V \) as the initial value.

c) If the value of \( V \) is less than or equal to the value of \( E_3 \), execute \( S_1 \), otherwise terminate the loop and start executing \( S_2 \) (the next statement or block).

d) After executing \( S_1 \), add the value of \( E_2 \) to \( V \) and return to step (c).

1.2 Decreasing loop-counter

The form is:

\[
\text{FOR } V \text{ FROM } E_1 \text{ STEP } E_2 \text{ DOWNTO } E_3 \text{ DO } S_1; \quad \text{(followed by next statement } S_2)\).
\]

This is compiled into exactly the same sequence as in (1.1) above except that the condition in step (c) becomes "greater than or equal to".
Notes:

i) In both forms, the initial value of V is tested against E3 before the statement S1 is executed even once. Therefore it is possible that, for the proper values of E1 and E3, the statement or block S1 will not be executed at all.

ii) The value of the loop variable V is well defined when the loop finishes as the value of V when the comparison in step (c) fails. (It is never the last value used, and is always the first value not used.)

iii) Branches to S1, or to labels defined within S1, are illegal and will be indicated as errors by the compiler. Branches from S1 out of the range of the loop are of course legal, and the last value of V in the loop will be retained.

iv) In both forms, the step value E2 is added to V at the end of each iteration. Therefore, it is the programmer's responsibility to ensure that E2 has a positive value when form (1.1) is used (UPTO), and a negative value when form (1.2) is used (DOWNTO).

2. THE 'WHILE' STATEMENT

The form is:

WHILE C DO S1; (followed by next statement S2).

This is translated by the compiler into code to do the following:

a) Evaluate C.

b) If C is true, execute S1; otherwise terminate the loop and start execution of S2.

c) After executing S1, return to step (a).

Notes:

i) As in the FOR statement, it is possible that S1 will not be executed at all, since the condition C is tested first.

ii) Each time around the loop, condition C is re-evaluated. Therefore, if C contains a data-expression it also will be recomputed each time around.

iii) Branches to S1 or to labels defined within S1 are illegal, and will be indicated as errors by the compiler. Branches from S1 out of the range of the loop are of course legal.

3. THE 'FOR-WHILE' STATEMENT

This form is a combination of the FOR and the WHILE statements. It has, therefore, two subforms, corresponding to the two subforms of the FOR statement.

3.1 Increasing loop-counter

The form is:

FOR V FROM E1 STEP E2 UPTO E3 WHILE C DO S1; (followed by next statement S2).

This is translated by the compiler into code to do the following:

a) Evaluate E1, E2, and E3, in that order.

b) Move the value of E1 to V.

c) If the value of V is less than or equal to the value of E3, proceed to step (d); otherwise terminate the loop and start execution of S2 (the next statement or block).

d) Evaluate C.
e) If C is true, execute S1, otherwise terminate the loop and start execution of S2 (the next statement or block).

f) After executing S1, add the value of E2 to V and return to step (c).

3.2 Decreasing loop-counter

The form is:

FOR V FROM E1 STEP E2 DOWNTO E3 WHILE C DO S1; (followed by next statement S2).

This is compiled exactly as above except that the condition in step (c) becomes "greater than or equal to".

Notes:

Since the FOR-WHILE is just a combination of the FOR and the WHILE, the same considerations will apply to the FOR-WHILE as to the FOR and WHILE separately. In particular,

a) it is possible that S1 will not be executed at all;

b) if the initial value of the loop variable V fails the first test in step (c), the condition C will not be evaluated at all.

c) when the loop finishes, the value of V is the value it had when the comparison in either step (c) or step (d) failed. As with the FOR statement, this is never the last value used, and is always the first value not used in the loop.

4. "INFINITE-LOOP" 'DO' STATEMENT

The form is:

DO S1; (followed by next statement S2).

This executes by simply evaluating S1, and, on normal completion of S1, evaluating it again, etc.

Examples:

```
FOR R0 FROM 0 STEP 2 UPTO 100 DO 0 => A(R0);
WHILE A > B : C <= D DO
   BEGIN .... END;
FOR R1 FROM 100 STEP -2 DOWNTO 2 WHILE A(R1) > B DO
   BEGIN .... END;
FOR A FROM R0+23+B STEP P UPTO X(R1-32) DO
   BEGIN .... END;
FOR X(R4) FROM 0 STEP -1 DOWNTO -100 DO
   BEGIN .... END;
FOR R1 FROM REF(A(0)) STEP -2 DOWNTO REF(A(-100)) DO
   BEGIN .... END;
FOR R1 FROM -11 STEP 1 UPTO -1 DO
   IF A(R1) > 0 THEN A(R1) => B(R1) ELSE A(R1) => B(R1) - ;
FOR R5 FROM 1 STEP 1 UPTO N DO
   FOR R4 FROM 1 STEP 2 UPTO K DO
      BEGIN .... END;
```

See also Appendix G on compiler optimizations.
5. DEGENERATE LOOP FORMS

A "FOR"-loop in PL-11 has five parts:

1) a loop control variable;
2) an initial value;
3) a STEP value;
4) a limit value;
5) a loop body.

The compiler will allow parts 2, 3, 4, and 5, or any combination thereof, to be missing. It is important to note that a missing part does not imply a default. It means that the action of that part is not done automatically as part of the loop control. This is illustrated in the following examples.

5.1 Loops without a STEP value

When iterating through arrays, it is often convenient to have the address of the next array element in a register. In this case it saves time and memory space to change this register implicitly by use of PUSH or POP rather than explicitly specifying a STEP of 2. For example, to clear an N-element integer array A we could write (assuming N is a compile-time constant):

```
FOR RI FROM REF(A) UPTO REF(A(2*N-2)) DO 0 => POP(RI);
```

Notice that the absence of an explicit STEP does not cause the compiler to assume a default value of 1 (as is the case in FORTRAN). The programmer must explicitly change the loop control variable within the loop body itself [in this example, POP(RI) adds 2 to RI].

5.2 Loops without an initial value

It often happens that the register (or variable) that will be used as the loop control variable already contains the initial value when the loop is entered. Therefore it is not necessary to specify the FROM value. For example, it would also be possible to clear the N-element integer array A in the following manner:

```
N => RI - 1 SLA;
FOR RI STEP -2 DOWNTO 0 DO 0 => A(RI);
```

This method is useful when N is a variable rather than a compile-time constant. It would also be possible to write:

```
FOR RI FROM N => RI - 1 SLA STEP -2 DOWNTO 0 DO 0 => A(RI);
```

which produces less code and is perhaps better programming style since all the elements of the loop are present.

5.3 Loops without initial or STEP values

We can combine the two examples given above as follows:

```
N => RI SLA + REF(A);
FOR RI DOWNTO REF(A(2)) DO 0 => PUSH(RI);
```

In the example, RI gets the initial value outside the loop, and the STEP is implicit in the use of PUSH.

5.4 Loops without a limit value

It often happens that a loop will be terminated by some other condition than the reaching of a certain limit by the control variable; for example, if byte array BUFFER contains a string of characters terminated by a carriage return (CR). We could have a loop to process each character with the control sequence:
FOR R1 FROM 0 STEP 1 WHILE BUFFER(R1) /= CR DO BEGIN
  ...
END;

In this case the WHILE condition replaces the limit value. Of course, care must be taken in leaving off the limit, since this loop would be infinite if BUFFER did not contain a carriage return. We could also write:

FOR R1 FROM REF(BUFFER) STEP 1 WHILE B1 /= CR DO BEGIN
  ...
END;

5.5 Loops without a body

If the purpose of the above loop is simply to count the number of characters in the line excluding the CR, but up to a maximum length of N, we would write (where N is a compile-time constant):

FOR R1 FROM 0 STEP 1 UPTO N-1 WHILE BUFFER(R1) /= CR DO;

The loop ends when either N characters are scanned or a CR is found, and then R1 will have the number of characters that precede the CR (if any), which is in the range (0,N).

5.6 Miscellaneous combinations

The last example can be written without a STEP value or a body as follows:

FOR R1 FROM REF(BUFFER) UPTO REF(BUFFER(N-1)) WHILE BPOP(R1) /= CR DO;

In this case R1 will contain the address of the byte following the CR or the last byte in BUFFER (whichever occurs first), which is perhaps not too useful.

Of course, if we are sure that BUFFER will contain a CR, then the UPTO limit can be omitted to give

FOR R1 FROM REF(BUFFER) WHILE BPOP(R1) /= CR DO;

which is a fairly "degenerate" loop.

6. GYPSY'S WARNING

The only types that are allowed as loop control variables are BYTE, INTEGER, and LOGICAL. If a LOGICAL variable is used as a control variable then the test performed in VIII.1.1 step (c) and VIII.3.1 step (c) is an unsigned test, rather than a signed test. This is important to remember, if the initial and final values are intended to be addresses, since for address values above 16K the sign bit is set -- but these addresses are not, of course, negative. The editor once spent several days tracking down a bug, before realizing that with the statements

R2 => R3+40;
FOR R1 FROM R2 STEP 10 UPTO R3 DO BEGIN ... END,

with R2 and R3 containing addresses, that if and only if R2 was less than 16K, and R3 was greater than 16K, then the loop was not executed at all. It is thus advisable in PL-11 to treat all address comparisons using LOGICAL variables. This example illustrates once again the necessity for clean and consistent programming styles.
Section IX. PROCEDURES IN PL-11

There are two important aspects of the procedure concept in any programming language: 1) the procedure definition; and 2) the procedure utilization. In FORTRAN, the procedure concept is represented by the SUBROUTINE and FUNCTION subprogram facility, where each "procedure" is defined by a separately compiled subprogram and is utilized from another program by either the CALL statement or a function reference in an expression.

In PL-11, procedures are defined by declarations in the program that intends to utilize the procedure. That is, procedures are defined and compiled as part of the main program, although the facility to compile them separately also exists (see Section IX.2).

1. LOCAL PROCEDURES

1.1 General

The method for declaring procedures is explained in Section IV.5 on Declarations. It is simply summarized here.

The form of a procedure declaration is:

(heading); (body)

where (heading) has the form:

PROCEDURE (symbolic-name)[(<register>)]

and (body) is any statement or block.

This declaration defines the (symbolic-name) as the name of the procedure, and the (body) as the code that will be executed whenever the procedure is called. The (register) is called the "return register", and is used to "remember" where the procedure was called. In PL-11, procedures are called by simply writing the name of the procedure as a statement (the word "CALL" is not used). The (register) is optional, and its omission affects the location of the return address, as is explained next.

1.1.1 Procedures declared with a link register

If a procedure declaration includes the specification of a link register, as for example in:

PROCEDURE ABC(R5):
    BEGIN
    ...
    END;

then when this procedure is entered by a call statement

ABC;

the link register R5 will contain the return address, and the old value of R5 (i.e. the value it had just before the call) will be the top element on the stack pointed to by SP.
1.1.2 Standard procedure declarations

In practice, most procedures are declared without specification of a link register, as for example:

```
PROCEDURE KLM;
BEGIN
...
END;
```

When this procedure is entered by a call statement

```
KLM;
```

the return address is simply pushed on to the stack pointed to by SP. No registers are changed, since none are involved in the procedure entry operation. It is difficult (but not impossible) to pass in-line parameters (see below) with this type of procedure linkage, but since in practice most procedures require no parameters, or pass them via registers, it is the most commonly used procedure linkage.

Examples:

```
PROCEDURE MAX (R5);
IF R1 > R0 THEN R1 => R2 ELSE R0 => R2;
```

This is called by simply writing the statement

```
MAX;
```

This call is compiled into a "jump-to-subroutine" instruction that puts the return address (i.e., the address of the next instruction after the call) into the return register specified by the procedure declaration (in this case R5). In this example the procedure (body) consists of a single IF statement. Usually, however, procedures require more than a single statement. In that case, the procedure (body) must be a block. For example, a procedure to add up the N elements of an integer array A would be:

```
PROCEDURE ADDUP;
BEGIN
0 => R1; N => R2 SLA - 2;
FOR R3 FROM 0 STEP 2 UPTO R2 DO R1 + A(3);
END;
```

This procedure would be called by the statement

```
ADDUP;
```

1.2 Scope

Since procedures are declared, the procedure declaration must appear with the other declarations at the beginning of a block. The scope of the procedure name will then be that block. In other words, the name of that procedure is "known" to the compiler only until the "END" for that block is compiled, at which time the compiler "forgets" the name of that procedure (as well as all other names declared in that block). Hence a procedure can be utilized only by statements in the block in which it is declared (or in any nested sub-blocks). For example:
BEGIN
  BEGIN INTEGER A,B,C;
  PROCEDURE P1;
  BEGIN BYTE X,Y,Z;
  END;
  P1; COMMENT THIS IS A LEGAL CALL ON P1;
  END;
  P1; COMMENT THIS CALL WILL GIVE AN ERROR MESSAGE;
  END.

1.3 Nesting

Since the (body) of a procedure can be a block, this block can have any number and kind of declarations, including other procedure declarations. The scope of these declarations will be simply the (body) of the outer procedure (i.e. they are local to that procedure). A procedure declared within the body of procedure B can only be utilized by statements within the body of B. It is "unknown" outside. For example:

BEGIN BYTE X,Y;
PROCEDURE B;
BEGIN INTEGER R,S;
PROCEDURE A;
BEGIN INTEGER C,D;
  END;
  ①
  END;
  ②
  END;
  ③
END.

At point ① a statement can access variables X,Y,R,S,C,D, and procedures B and A. At point ② only variables X,Y,R, and S are accessible, although both procedures B and A may still be called. At point ③ only variables X,Y, and procedure B may be accessed. This nesting of procedures can occur to any desired depth.

1.4 Recursion

Note that in the above example, a call on procedure A or B from point ① would be a recursive call, since ① is already in procedure A which itself is in B. This is perfectly legal. However, unlike ALGOL 60, storage for local variables (such as C and D) is not allocated dynamically at the time the procedure is called, but rather is allocated statically at the time the procedure is compiled. Hence each call will use the same copy of any local variables, regardless of the depth of recursion. (Hence, variables in PL-11 correspond directly with the "Static OWN" variables of ALGOL 60.)

1.5 Returns

When the execution of the (body) of a procedure is complete, control automatically returns to the statement following the one in which the call was made. This is due to the fact that the compiler automatically generates a "return-from-subroutine" at the end of the code generated for each procedure (body). However, it is often convenient to be able to "return" from anywhere within the procedure (body), and that is the purpose of the RETURN statement. Its form is simply

RETURN;

This causes the compiler to generate the return-from-subroutine at the place it occurs. See also Appendix G on Compiler Optimization.
1.6 Parameter passing

There are several techniques available in the PDP-11 hardware for passing parameters, and they can all be utilized in PL-11.

1.6.1 Registers

For procedures having a small number of parameters (one or two), the fastest and easiest way to pass parameters to a procedure is to load them into registers. For example, the procedure MAX defined in Section IX.1.1.2 above puts the larger of register R1 or R0 into R2. Hence to utilize this procedure to find the larger of any two values, say A and B, and store it in C, we simply write

\[ A \rightarrow R0; \quad B \rightarrow R1; \quad \text{MAX}; \quad R2 \rightarrow C; \]

Note that no mention of these parameters is made either in the procedure declaration or the call. They are "parameters" by virtue of a programming convention.

1.6.2 Storage lists

If the number of parameters is large, a common technique is to put them in a list in storage (i.e. for n parameters as the first n elements in an array), and then to simply load the address of this list (i.e. the address of the first element of the array) into a register before calling the procedure. For example, if we define a procedure MIN as

```plaintext
PROCEDURE MIN (R5);
    BEGIN
        COMMENT WE EXPECT THE LIST ADDRESS IN R1;
        IF M1(0) < M1(2) THEN M1(0) \rightarrow M1(4) ELSE M1(2) \rightarrow M1(4);
    END;
```

then it can be called as follows to put the smaller of A and B into C:

```plaintext
COMMENT ASSUME WE HAVE DECLARED 'LIST' AS ARRAY 3 INTEGER;
REF (LIST) \rightarrow R1; \quad A \rightarrow M1; \quad B \rightarrow M1(2); \quad \text{MIN}; \quad M1(4) \rightarrow C;
```

Once again neither the procedure declaration nor the procedure call make any mention of "parameters", since there are "parameters" only in the sense of a programming convention for passing information to a procedure.

1.6.3 In-Line lists

A special case of the storage list mechanism is to have the list of parameters stored immediately following the jump-to-subroutine instruction. This is the method commonly used in many systems, and is almost universally used by FORTRAN subprograms. In this method, the register that points to the parameter list is simply the return register specified in the procedure heading, which is loaded automatically by the jump-to-subroutine instruction. With this method, the address after the call is not that of the next instruction, but is the address of the first parameter in the list. In PL-11, in order to have the compiler store the parameters immediately after the jump-to-subroutine instruction, they must be specified as part of the call statement. For example, if we define procedure MAX as

```plaintext
PROCEDURE MAX (R5);
    BEGIN
        IF M5(0) > M5(2) THEN M5(0) \rightarrow R1 ELSE M5(2) \rightarrow R1;
        RETURN(4);
    END;
```

then the call to this procedure would be (for example):

```plaintext
MAX (10,15);
```

Notice that no mention of the parameters is made in the procedure declaration.
This method poses two problems not encountered with the other techniques:

i) The parameters included in a call must be constants (i.e., numbers, strings, or addresses of variables). If they were not constant, some means would have to be provided to change their value before each call. This implies that the program would have to modify itself (as opposed to its data structures). This is undesirable, for several reasons -- self-modifying programs are more difficult to understand and debug, since during execution they are not the same as the compiled listing. More important, especially for on-line work that must account for interrupts, self-modifying programs are neither re-entrant nor recursive. On larger systems with memory protection, storing into the program code is prevented by the hardware. Unfortunately such hardware is not available on all POP-11s, but the PL-11 language, as in all high-level languages, contains no mechanism for explicitly modifying the program code. Hence PL-11 "in-line" parameters must be constants.

Although this may seem a severe restriction, one must consider that addresses of variables are allowed, and this permits a subroutine, by indirect addressing, to reference any variable or array desired. This is exactly the way in which parameters are passed in FORTRAN-generated code, even though they are not written this way by the programmer.

ii) The return from subroutine becomes trickier with in-line parameters, since the return register does not point to the instruction following the call, but rather points to the first in-line parameter. Therefore the programmer must explicitly modify this register in the procedure, so that it points to the address after the last parameter in the in-line parameter list. If there are n parameters, 2n must be added to the return register in the procedure. As a programming convenience, the RETURN statement in PL-11 allows an optional expression, enclosed in parentheses, which will be added to the return register before the return-from-subroutine instruction is executed. For example, if RS is the return register,

```
RETURN(4);
```

generates exactly the same code as the sequence

```
RS + 4; RETURN;
```

An alternative method of ensuring that the return register is correctly updated involves copying the parameters into local variables. Thus, the call

```
ABC(22,99,-100);
```

will cause RS to be pointing to a list of three words containing 22, 99, and -100, respectively, at the time when the body of ABC begins execution. In order to store these values into local variables X, Y, and Z we could have:

```
PROCEDURE ABC(RS);
BEGIN
INTEGER X,Y,Z;
POP(RS) => X;
POP(RS) => Y;
POP(RS) => Z;
...
...
END;
```

This code will modify RS to point to the next cell after the third parameter in the list, which is the instruction to return to after the procedure execution is finished.

In all cases, the modifications of the return register necessitated by the in-line parameters must be explicitly programmed by the programmer (with the aid of the expression in the RETURN if he so chooses). The compiler generates no automatic modification instructions. Hence returning from a procedure is made a great deal more difficult by in-line parameters, and will probably be the source of many errors, especially for FORTRAN programmers. In addition, the difficulties encountered when addresses are passed as parameters (in which case all uses of the address within the procedure must be indirect in order to get at the variable addressed) will also be a source of error. Hence it is recommended that the Register or Storage List methods be used instead of the In-Line Lists method whenever possible.
2. **EXTERNAL AND GLOBAL PROCEDURES**

Although not required as in FORTRAN, it is sometimes useful to be able to compile procedures separately from the program that utilizes them.

This is obviously necessary if the procedure is written in a different language than the program using it, or vice versa.

In PL-11 we must consider the two aspects of the procedure concept: definition and utilization.

2.1 Definition

A procedure that is to be utilized by separately compiled programs is called a GLOBAL procedure, and is defined by prefacing the ordinary procedure declaration with the reserved word GLOBAL. For example:

```
GLOBAL PROCEDURE MAX;
BEGIN
  IF R0>R1 THEN R0 => R2 ELSE R1 => R2;
END;
```

This will cause the compiler to generate the necessary instructions to the linkage editor so that references by other programs to the name MAX will be to this procedure.

2.2 Utilization

In order to utilize a global procedure in a program other than the one in which it is compiled, we must still have a declaration for it, but the purpose of the declaration is not to define the code for the procedure but to inform the compiler that in fact that code exists in some other program external to the one being compiled. This declaration has the form:

```
EXTERNAL PROCEDURE (symbolic name)[{(reg)}] [(symbolic name)[{(reg)}], ...];
```

Examples:

```
EXTERNAL PROCEDURE MIX,MUX,MUX(R3),KL0D;
EXTERNAL PROCEDURE NEDDY;
```

These two statements define MIX, MUX, MUX, KL0D, and NEDDY as being external procedures. They all use the PC as link register, except MUX which uses R3. Note that no procedure (body) is specified, since the code has been compiled elsewhere. The call to this procedure is no different from a call to any local procedure, namely:

```
MAX;
```

Obviously the same return register must be specified in both the GLOBAL and EXTERNAL declarations of the same procedure name.

2.3 Scope

EXTERNAL and GLOBAL procedures can be declared in any block in the program. The scope of the name of these procedures in that program will be the block in which the declaration occurs. This is exactly the same as for any ordinary declaration. However, there are side effects implied by the EXTERNAL and GLOBAL concepts. In the case of EXTERNAL, this means that calls to the procedure will result in control being passed to code generated in another separately compiled program completely outside the program being compiled. In the case of GLOBAL, it means that control may be passed to this procedure by some unknown separately compiled program. This means that care must be used in declaring GLOBAL procedures nested inside inner program blocks, since it becomes possible to transfer control into the procedure from outside the program in a way that completely violates the block structure of the program.
3. 'TRAP' AND 'EMT' PROCEDURES

In the PDP-11 many standard monitor routines are entered by means of special instructions that cause a transfer through a fixed low-core cell to the monitor, which decides which routine is being called. In a PL-11 program, these monitor routines are given symbolic names by declarations of the form:

EMT PROCEDURE (symbolic name) ((litera expr.) [, (symbolic name) ((litera expr.) ...) ...];

or

TRAP PROCEDURE (symbolic name) ((litera expr.) [, (symbolic name) ((litera expr.) ...) ...];

For example:

EMT PROCEDURE XYZ(21);

defines the symbolic name "XYZ" as being EMT routine number 21 in the monitor. In order to call this monitor routine, we simply write the normal procedure call statement:

XYZ;

In this case, the compiler will generate an EMT instruction rather than a jump-to-subroutine. If the monitor routine requires in-line parameters, they are written in PL-11 exactly as for normal procedure calls, for example:

XYZ (1, REF(A), #10);

However, most parameters are passed to monitor routines by the use of registers, due largely to the difficulties of in-line parameters discussed earlier.

The PDP-11 hardware allows 256 different EMT procedures, and 256 TRAP procedures. Therefore the value of the (literal expression) specified in a TRAP or EMT declaration must be a number \( N \), \( 0 \leq N \leq 255 \).

Example:

EMT PROCEDURE READ($11), PRINT(#A8), TELETIME(170);

is equivalent to the three declarations:

EMT PROCEDURE READ($11);
EMT PROCEDURE PRINT(#A8);
EMT PROCEDURE TELETIME(170);

4. INTERRUPT PROCEDURES

4.1 Introduction

In order to be able to specify interrupt servicing in PL-11, the language contains the concept of an interrupt procedure.

When the module containing an interrupt procedure is loaded into the PDP-11, the compiler ensures that the entry point address of this procedure will be loaded into the appropriate interrupt vector in low core, so that any interrupt signal to that location will cause the interrupt procedure to "be called" asynchronously to the normal program execution.

There are three types of interrupt procedures. The first is the most basic and the most general, since it can be used to handle any possible interrupt. The other two are designed to handle the 24 CAMAC interrupts and 12 Omega EXTERNAL interrupts in a manner that is more convenient for the applications programmer.
4.2 General interrupt handling

The syntax for declaring a general interrupt procedure is:

```
INTERRUPT (address) PROCEDURE (id)[((new-PS))];
BEGIN
...
END;
```

where (address) is any even absolute byte address in memory, (id) is the programmer-defined name for the procedure, and (new-PS) is the second word of the PDP-11 interrupt vector, namely the value which will be loaded into the PS of the machine when the interrupt occurs. This part of the declaration can be omitted, as in the following example:

```
INTERRUPT $20 PROCEDURE XYZ;
BEGIN
...
END;
```

When this module is loaded, the word at absolute memory location 20 octal will be loaded with the address of the first instruction of this procedure. Location 22 octal (the second word of the interrupt vector) will not be loaded, since the (new-PS) is not specified.

If control can reach the 'END' of the interrupt procedure body, the compiler will automatically generate an RTI instruction (rather than the normal RTS instruction that is generated at the end of normal procedures). In addition, any 'RETURN' statements appearing in the body of an interrupt procedure will cause an RTI (instead of the normal RTS) to be generated. Of course, the programmer is free to write explicitly RTI (or RTT) instead of RETURN if he so wishes.

A PL-11 module can contain any number of interrupt procedures. They can appear in any order (not necessarily in order of ascending vector addresses, for example), and can be intermixed with normal declarations in any desired fashion. The same (normal) rules of block structure apply for these procedures as for any other.

An interrupt procedure cannot be called explicitly by the programmer, for the obvious reason that a "call" (JSR) pushes only one word (the return address) onto the stack, whereas an interrupt pushes two words (the return address and the old PS) onto the stack. Therefore the return from a normal procedure (RTS) must pop off only one word from the stack, whereas the return from an interrupt procedure (RTI) must pop off two words. Therefore "catastrophic" errors would occur if an interrupt procedure were called (and vice versa). The programmer may, however, take the REF of an interrupt procedure, in the same manner as with normal procedures.

4.3 CAMAC interrupt handling

The syntax for declaring a CAMAC interrupt procedure is:

```
CAMAC INTERRUPT (number) PROCEDURE (id);
BEGIN
...
END;
```

where (number) must be a numeric constant in the range (1,24) corresponding to the number of the LAM that triggers the interrupt, and (id) is the programmer-defined name for the procedure.

CAMAC INTERRUPT PROCEDUREs differ from the general INTERRUPT PROCEDURE in that the entry-point address is loaded into a table internal to the operating system, rather than directly into a low-core vector location. The actual interrupt will first transfer control to the operating system, where the current register values will be saved, and then the operating system will call this procedure in the normal call (JSR) manner. Hence the 'END' of this procedure body, and any RETURN appearing within the procedure body, will generate
the normal procedure return instruction (RTS). When this return is executed, control returns to the operating system, where the previous register values are restored and then an RTI is executed (by the operating system). Hence, to the programmer, the only difference between CAMAC interrupt procedures and normal procedures is that these may be called by the operating system at any time, asynchronously to the normal execution sequence.

The programmer may explicitly call a CAMAC interrupt procedure in the same manner as normal procedures are called. However, such a call will not transfer control to the operating system (since there is no interrupt), but will simply transfer to the first instruction of the procedure, as with normal procedures. The return is also the same as for normal procedures. Care must be exercised in calling these procedures explicitly, however, since reentrancy problems may arise if an interrupt occurs that causes the operating system to call the same procedure asynchronously.

4.4 **EXTERNAL interrupt handling**

The syntax for declaring an EXTERNAL interrupt procedure is:

```
EXTERNAL INTERRUPT (number) PROCEDURE (id);
BEGIN
...
END;
```

where (number) must be a numeric constant in the range (1,12) corresponding to the number of the Omega EXTERNAL interrupt source that triggers the interrupt, and (id) is the programmer-defined name for the procedure.

The rules for these procedures are identical to those for the CAMAC interrupt procedures, as described above. The operating system merely uses a different internal table, corresponding to the different source of the interrupt triggers.

**Note:**

This use of the word EXTERNAL should *not* be confused with its use for EXTERNAL PROCEDURE declarations. It is unfortunate, but due to historical circumstances and the desire to minimize the number of reserved words, apparently unavoidable, that different words could not be used.

4.5 **GLOBAL and EXTERNAL linkage to interrupt procedures**

All three types of interrupt procedures described above are "local" to the module in which they are declared. It is possible to declare them either GLOBAL to that module, or EXTERNAL to that module in the same manner as normal procedures are declared GLOBAL or EXTERNAL to the module in which they occur. The syntax is as follows:

```
interrupt (address) external procedure (id)[(new-ps)];
camac interrupt (number) external procedure (id);
external interrupt (number) external procedure (id);
```

As in normal EXTERNAL PROCEDURE declarations, no procedure body is given, because it is exactly this body which is external to this module. There must of course be a corresponding GLOBAL PROCEDURE declaration in some other module that is linked with this one to form the core load:
INTERRUPT (address) GLOBAL PROCEDURE (id);[(new-PS)];
BEGIN
...
END;

CAMAC INTERRUPT (number) GLOBAL PROCEDURE (id);
BEGIN
...
END;

EXTERNAL INTERRUPT (number) GLOBAL PROCEDURE (id);
BEGIN
...
END;

As in normal GLOBAL PROCEDURE declarations, these declarations must have a body. The name of the procedure (id) is made global to this module so that it can be referred to by other modules that are linked together with this one.

A single module can contain any mixture of any of the possible types of procedure declarations, in any order. The normal rules of block structure always apply.

5. "HARDWARE" PROCEDURES

The following PDP-11 control instructions have been pre-declared in PL-11 as procedure names of a special type. When these procedures are called, the corresponding hardware instruction rather than a jump-to-subroutine instruction is generated.

<table>
<thead>
<tr>
<th>PL-11 procedure call</th>
<th>Instruction code generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>HALT;</td>
<td>0</td>
</tr>
<tr>
<td>WAIT;</td>
<td>1</td>
</tr>
<tr>
<td>RTI;</td>
<td>2</td>
</tr>
<tr>
<td>BPT;</td>
<td>3</td>
</tr>
<tr>
<td>IOT;</td>
<td>4</td>
</tr>
<tr>
<td>RESET;</td>
<td>5</td>
</tr>
<tr>
<td>RTT;</td>
<td>6 (models 40 and 45 only)</td>
</tr>
<tr>
<td>SPL0 ... SPL7;</td>
<td>230 ... 237 (model 45 only)</td>
</tr>
<tr>
<td>NOP;</td>
<td>240 (octal)</td>
</tr>
</tbody>
</table>

These "procedures" do not have addresses, and there is no way for the programmer to declare them in PL-11.

6. STACK PROCEDURES ON THE MODEL 45 USING THE 'MARK' INSTRUCTION

The MARK instruction will be generated for calls to procedures that have been described as requiring the stack linkage convention. The form of such a procedure declaration is, for example:

```
PROCEDURE ABC STACK 3;
BEGIN
...
END;
```
This indicates to the compiler that all calls to procedure ABC must include three parameters which are to be pushed onto the stack along with the MARK instruction as defined in the PDP-11/45 manual. A call would be, for example:

\[ABC (R1, 257, X(R4));\]

The entire linkage sequence, including the MARK instruction and the pushing of each parameter onto the stack, is generated automatically by the compiler. An error is indicated if the number of parameters in the call does not match the number declared. A declaration such as:

\[
\text{PROCEDURE XYZ STACK; BEGIN } \ldots \text{ END;}
\]

allows a variable number of parameters to appear in any call to XYZ.
Section X. FLOATING POINT IN PL-11

1. INTRODUCTION

The PL-11 floating-point features allow the programmer to utilize the hardware floating-point options on the models 40 and 45 in an efficient and effective manner. PL-11 contains:

i) two data types: REAL, for 32-bit "single-precision" floating-point values, and DOUBLE, for 64-bit "double-precision" floating-point values;

ii) constants corresponding to each of the two data types.

These have already been described in Sections III and IV. We will now describe the operations and functions which utilize the floating-point hardware options to process variables of the new data types.

The new data types and constants can be used on any machine, but none of the operations and functions are available on the model 20. They are all assumed for the model 45. On the model 40, the [PPP card (see Section II.4) must be used to tell the compiler that the floating-point option is available.

An important note:

Since the model 40 floating-point hardware is totally different from the model 45 hardware, and since the basic philosophy of PL-11 is to accurately reflect the capabilities of the hardware, there has been no attempt to disguise the differences. A programmer must know which model he is writing for and program accordingly. It is impossible for a PL-11 program using the model 45 floating-point features to run on the model 40, and vice versa. If compatibility is desired, either avoid floating point altogether, perform all floating point in subroutines that can be replaced depending on the model, or use FORTRAN. (In fact, it appears that the model 40 floating-point hardware is explicitly designed for DEC's FORTRAN convention, since it is hard to imagine why else it would be designed in such a clumsy, inefficient, unusable manner).

2. PRE-DECLARED REGISTERS

On the PDP-11 model 45 there are six extra floating-point hardware registers which can be used to hold either single- or double-precision floating-point values. In PL-11 these registers have been given the pre-declared names: AC0, AC1, AC2, AC3, AC4, and AC5 when they are used as single-precision registers, and DAC0, DAC1, DAC2, DAC3, DAC4, and DAC5 when they are used as double-precision registers.

3. FLOATING-POINT OPERATIONS ON THE MODEL 45

3.1 Tutorial

Programming for the model 45 floating-point unit in PL-11 is done by means of arithmetic expressions that utilize the normal PL-11 operators \( \Rightarrow \) (move), \( + \) (add), \( - \) (subtract and negate), \( \times \) (multiply), \( / \) (divide), ABS (absolute value), and comparisons \( >, \geq, =, /=, \leq, < \). In general, floating-point expressions are written in the same manner as normal integer or logical expressions, except that the operands involved are real or double variables and registers. However, there are a few additional restrictions which the programmer
must be aware of, since the floating-point processor is not as general in its ability to handle operands as is the general arithmetic processor. In particular, all floating-point operators except the monadic operators (negation, absolute value, test, clear) must have one of the registers AC0, AC1, AC2, or AC3 (DAC0, DAC1, DAC2, or DAC3 for double precision) as an operand. For all dyadics except move (=>) this register must be the left operand, as it will contain the result. For move, either one of the operands, and possibly both, must be one of these registers. For example, to add the two floating-point values A and B and store the result in C we would write:

\[ A \Rightarrow AC0 + B \Rightarrow C; \]

As in all PL-11 expressions, floating-point expressions are evaluated strictly left-to-right, with no operator hierarchy and no possibility of parenthesization. Other examples of valid PL-11 floating-point expressions are:

```plaintext
REAL A,B,C,D,E,F;
A \Rightarrow AC1 * C / D - E \Rightarrow AC4;
B \Rightarrow AC0 / AC4 ABS;
AC0 \Rightarrow F;
```

Of course, floating-point operands can be subscripted in the same manner as integers. For example, to add all 30 elements of a real array X, we might write:

```plaintext
ARRAY 10 REAL X;
0 \Rightarrow AC2;  COMMENT CLEAR THE ACCUMULATOR;
FOR R1 FROM 36 STEP -4 DOWNTO 0 DO AC2 + X(R1);
```

The total is now in AC2. To compute the average, we must write in addition:

```plaintext
REAL TEN = 10.0;
AC2 / TEN;
```

If we wish to compute the average of X weighted by array W, we could write:

```plaintext
REAL AVERAGE;
ARRAY 10 REAL X,W;
0 \Rightarrow AC2;  COMMENT ACCUMULATOR FOR WEIGHTED SUM;
0 \Rightarrow AC1;  COMMENT ACCUMULATOR FOR SUM OF WEIGHTS;
FOR R3 FROM 0 STEP 4 UPTO 36 DO
  BEGIN
    X(R3) \Rightarrow AC3 * W(R3);
    AC2 + AC3;
    AC1 + W(R3);
  END;
AC2 / AC1 \Rightarrow AVERAGE;  COMMENT THE RESULT IS NOW IN AVERAGE;
```

This example could also be programmed using the indirect and auto-increment addressing modes (RIND and RPOP, respectively) for real variables as follows:
0 => AC2;  
0 => AC1;  
REF(X) => R1;  COMMENT R1 POINTS TO X ELEMENTS;
REF(W) => R2;  COMMENT R2 POINTS TO W ELEMENTS;
FOR R3 FROM 10 STEP -1 DOWNTO 1 DO
BEGIN
RPOP(R1) => AC3 * RIND(R2);  
AC2 + AC3;  
AC1 + RPOP(R2);  
END;
AC2 / AC1 => AVERAGE;

Note that RPOP(R1) increments R1 by 4, and RPUSH(R1) decrements R1 by 4. The corresponding double-precision operands are DPPOP(R2) (which increments R2 by 8) and DPUSH(R2) (which decrements R2 by 8).

3.2 Precision modes

The PDP-11 floating-point hardware instructions do not explicitly indicate whether the operation is to be performed in single or double precision. Instead, the entire floating-point processor is set up by a programmed instruction to operate either in single or double mode. In PL-11 these instructions are:

SETF; - sets the floating-point mode to single precision
SETD; - sets the floating-point mode to double precision

Once the floating-point mode is set by the execution of one of these instructions, all following floating-point operations will automatically be carried out in that precision.

In order to generate efficient code, the PL-11 compiler does not automatically generate either of these instructions under any circumstances. It is the programmer's responsibility to include these instructions where appropriate in his program, since the mode setting is a dynamic process that is best controlled by the programmer. This enables the programmer to write efficient programs, but also introduces a source of considerable error that he must be aware of. For most applications that are entirely performed in one precision only, it suffices for the programmer to simply include the mode setting instruction once at the start of his program. All following floating-point operations will be carried out in that mode regardless of whether they are written as REAL or DOUBLE operations in PL-11.

This point cannot be stressed too strongly. The PL-11 compiler checks to make sure that an expression contains only REAL or only DOUBLE operations (i.e. it checks the consistency of the operations in an expression), but there is no guarantee that these operations will be carried out in the indicated precision -- only the programmer can guarantee this by proper use of the SETF and SETD instructions. For example:

REAL X,Y,Z;
DOUBLE A,B,C;
SETF;
X => ACO + Z => Y;
A => DACO + B => C;

will compile correctly but will not execute properly, since the programmer has not explicitly changed the floating-point mode between the two expressions. Hence the second expression will not be carried out in double precision, as one would expect from reading the program. Instead the programmer must explicitly change the mode as in the following:

REAL X,Y,Z;
DOUBLE A,B,C;
SETF; X => ACO + Z => Y;
SETD; A => DACO + B => C;
Clearly the simplest course is to perform all operations in only one mode, and it is anticipated that in the vast majority of the applications this will in fact be the situation. In this case, the programmer simply sets the mode once. Since PL-11 never implicitly changes the mode, the code will not include any redundant mode-setting instructions and will operate "as efficiently as possible". If operations in both modes are desired in the same program, the programmer must be aware of the difficulty -- which, after all, is a "feature" of the DEC hardware!

3.3 REAL and DOUBLE operands

In addition to variables declared to be of type REAL and DOUBLE, which are addressed in PDP-11 hardware mode 6, and the REAL and DOUBLE registers, which are addressed in hardware mode 0, there are six other addressing modes that can be used to reference floating-point operands. These are expressed in PL-11 by the use of RPOP, RPUSH, and RIND for REAL, and DPOP, DPUSH, and DJIND for DOUBLE in exactly the same manner as POP, PUSH, and IND are used to reference INTEGER operands. RPOP, RPUSH, DPOP, DPUSH operate only on INTEGER or LOGICAL registers which contain the address of a floating-point variable. RIND and DJIND operate on any INTEGER or LOGICAL operand which contains the address of a floating-point variable. For example:

```
ARRAY 5 REAL A;
STACK 5 REAL B;
REAL C;
REF(A) => R1; REF(B) => R2; REF(C) => R3;
SETF 0 => RIND(R3);  COMMENT CLEAR C TO 0,
COMMENT COMPUTE SUM OF A(I) - B(5-I) IN C;
FOR R4 FROM 5 STEP -1 DOWNTO 1 DO 
    RIND(R3) + RPOP(R1) = RPUSH(R2);
```

**Note:**

RPOP increments the register by 4, DPOP by 8.
RPUSH decrements the register by 4, DPUSH by 8.

3.4 Move operations

The move operator (=>) can cause code to be generated for a number of different floating-point instructions, depending on the types of its two operands and which one is in a floating-point register (at least one operand of a dyadic floating-point operator must be in floating-point registers AC0, AC1, AC2, or AC3). These complications are also due to the fact that for floating-point operations, DEC has abandoned its single 'MOV' operation in favour of 'LOAD' and 'STORE' operations.

If both operands of a move are the same floating-point type (i.e. both REAL or both DOUBLE), the PL-11 compiler will translate the move operator (=>) into a LDIF (LDF) instruction if the right operand is AC0, AC1, AC2, AC3 (DAC0-DAC3), and into a STIF (STD) instruction if the left operand (but not the right) is AC0-AC3 (DAC0-DAC3). Of course it is an error if neither operand is one of these registers.

Examples:

```
REAL A, B;
DOUBLE X, Y;
SETF;  A => AC2;  COMMENT LDIF;
SETD; DAC2 => X;  COMMENT STD;
```

If the right operand is AC0-AC3 and the left operand is any operand of type DOUBLE, or if the right operand is DAC0-DAC3 and the left operand is any operand of type REAL, the move operator is translated into an LDIF (LDF) instruction; that is, a load that also performs the indicated conversion from DOUBLE to REAL or REAL to DOUBLE. However, this instruction will only operate properly if the floating-point mode has been set correctly by the programmer, as shown in the following example:
REAL  A,B;
DOUBLE  X,Y;
SETF;  X  =>  AC2;  COMMENT  CONVERT  DOUBLE  TO  SINGLE;
SETD;  A  =>  DAC1;  COMMENT  CONVERT  SINGLE  TO  DOUBLE;

If the left operand is AC0-AC3 and the right operand is any operand of type DOUBLE except DAC0-DAC3, or if the left operand is DAC0-DAC3 and the right operand is any operand of type REAL except AC0-AC3, the move operator is translated into a STCFI (STCDFI) instruction; that is, a store that also performs the indicated conversion, provided of course the programmer has set the floating-point mode bit correctly, as is shown in the following example:

REAL  A,B;
DOUBLE  X,Y;
SETF;  AC1  =>  X;  COMMENT  SINGLE  TO  DOUBLE;
SETD;  DAC3  =>  A;  COMMENT  DOUBLE  TO  SINGLE;

Note that the following perform as expected (i.e. two correct mode conversions per expression).

SETF;  X  =>  AC1  =>  Y;  COMMENT  DOUBLE  TO  SINGLE  TO  DOUBLE;
SETD;  A  =>  DAC2  =>  B;  COMMENT  SINGLE  TO  DOUBLE  TO  SINGLE;

If the right operand of a move operator (=>) is a REAL register AC0-AC3 or a DOUBLE register DAC0-DAC3, and the left operand is any operand of type INTEGER or LOGICAL, the PL-11 compiler translates the move (=>) into a LDCL (LDCT, LDCL, LDCLD) instruction. This single instruction can perform four different things, depending on how the programmer has set the mode bits in the floating-point unit (this instruction is truly a triumph for the bit-minded hardware types of the world). The following example demonstrates the possibilities:

INTEGER  I,J;
ARRAY  2  INTEGER  K;
SETF;  SETI;  I  =>  AC0;  COMMENT  16-BIT  INTEGER  TO  SINGLE;
SETD;  SETI;  I  =>  DAC1;  COMMENT  16-BIT  INTEGER  TO  DOUBLE;
SETF;  SETI;  K  =>  AC2;  COMMENT  32-BIT  INTEGER  TO  SINGLE;
SETD;  SETI;  K  =>  DAC3;  COMMENT  32-BIT  INTEGER  TO  DOUBLE;

Note that the left operand in such an operation can also be a 16-bit literal (i.e. decimal, octal, or hexadecimal number, or a 2-character string, or an address). Note also that any right operand other than AC0-AC3 or DAC0-DAC3 is illegal.

If the left operand of a move operator (=>) is a REAL register AC0-AC3 or a DOUBLE register DAC0-DAC3, and the right operand is any operand of type INTEGER or LOGICAL, the PL-11 compiler translates the move (=>) into a STCFI (STCFI, STCD1, STCDL) instruction, which also can perform four different conversions depending on how the programmer has set the mode bits in the floating-point unit. The following example shows the correct settings for each possible type of conversion:

INTEGER  I,J;
ARRAY  2  INTEGER  K;
SETF;  SETI;  AC0  =>  I;  COMMENT  SINGLE  TO  16-BIT  INTEGER;
SETD;  SETI;  DAC1  =>  J;  COMMENT  DOUBLE  TO  16-BIT  INTEGER;
SETF;  SETI;  AC2  =>  K;  COMMENT  SINGLE  TO  32-BIT  INTEGER;
SETD;  SETI;  DAC3  =>  K;  COMMENT  DOUBLE  TO  32-BIT  INTEGER;

It is also possible to load and store just the exponent part of floating-point register AC0-AC3 (DAC0-DAC3) by using the PL-11 reserved word EXPONENT, as shown in the following:
INTEGER I,J;
I => EXPONENT(AC1);  COMMENT THE LDEXP INSTRUCTION;
EXPONENT(DAC3) => J;  COMMENT THE STEXP INSTRUCTION;

The word EXPONENT can only be applied to floating-point registers ACO-AC3 (DAC0-DAC3),
and the other operand of the move must be of the type INTEGER or LOGICAL (it can, however,
be in any of the eight possible addressing modes). A constant can also be loaded (but not
stored), as for example:

$21 => EXPONENT(DAC2);

which sets the exponent part of DAC2 to 21 octal.

3.5 Other operations

3.5.1 Dyadic operators

The PL-11 dyadic operators * (add), - (subtract), * (multiply), / (divide), and the
six relational operators (>, >=, =, /=, <=, <) can be applied to floating-point operands in
the same manner as they are applied to INTEGER and LOGICAL operands, but with the hardware-
imposed restriction that, for the operators +, -, *, and /, the left operand (i.e. the one
that will contain the result of the operation) must be a register ACO-AC3 (or DAC0-DAC3).
For the relational operators, either the left or the right operand must be one of these
registers. The two operands must be both REAL or both DOUBLE in order to compile correctly,
but of course the actual mode used during execution depends on the dynamic setting of the
floating-point mode bit, which is the programmer's responsibility.

For example:

REAL  A,B,C,D;
DOUBLE X,Y,Z,W;
SETF;  AC1 + A - B * C / D;
SETD;  DAC2 * W + Y / Z - X;
SETF;  IF AC2>=B & AC3<=C THEN BEGIN ... END;
SETD;  IF DAC3/=W : Z/=DAC1 : DAC3/=DAC5 THEN
        BEGIN ... END;

In addition to these operators, the PL-11 operator MULT, when applied to REAL or DOUBLE
operands, is translated into the MODF (MODD) instruction. This instruction, which has a
fairly restricted utility, is explained in detail in the DEC manual. Its primary function
appears to be in converting REAL or DOUBLE values for printing. As with the other dyadic
operators, the left operand must be a floating-point register ACO-AC3 (DAC0-DAC3), and the
operands must be both REAL or both DOUBLE.

3.5.2 Monadic operators

The PL-11 monadic operators - (negation) and ABS (absolute value) can be applied to
any floating-point operand in the same manner as they are applied to INTEGER and LOGICAL
operands. For these operators there is no restriction to register operands. In addition,
the TSTF (TSTD) instruction is generated if any floating-point operand is compared with
zero, as in:

REAL  A,B;
DOUBLE X,Y;
SETD;  IF X=0 & Y=0 & DAC3/=0 THEN
        BEGIN ... END;
SETF;  IF A>0 : B<=0 : AC2=0 THEN
        BEGIN ... END;

and the CLRIF (CLRID) instruction is generated if zero is moved into any floating-point operand,
as in:
REAL A,B;
DOUBLE X,Y;
SETD; 0 => X; 0 => DAC1;
SETF; 0 => ACS; 0 => B;

Examples:

REAL A,B;
DOUBLE X,Y;
SETD; X ABS; Y--; DAC2--;
SETF; AC1--; AC2 ABS; B--;

As with dyadic operators, it is the programmer's responsibility to set the floating-point mode dynamically with SETD and SETF instructions in order that these monadic operations actually execute in the precision indicated by the compile-time type of their operands.

3.5.3 Functions

In addition to the "hardware procedures":

SETF;
SETD;
SETI;
SETL;
CFPC;

each of which translates directly into the hardware instruction of the same name, there are three floating-point functions:

LDFPS((operand));
STFPS((operand));
STST ((operand));

where (operand) is any non-literal INTEGER or LOGICAL operand. These functions correspond directly to the LDFPS, STFPS, and STST instructions, respectively.

4. FLOATING-POINT OPERATIONS ON THE MODEL 40

The four hardware floating-point instructions for the model 40 are generated from the following PL-11 functions:

FADD((register));
FSUB((register));
FMUL((register));
FDIV((register));

where (register) is an INTEGER or LOGICAL register that points to a stack area, the top four words of which must be set up by the programmer to contain the addresses of the operands for the operation. For example:

REAL A,B;
STACK 4 ADDRESSES;
REF(ADDRESSES) => R5;
REF(A(2)) => PUSH(R5);
REF(A(0)) => PUSH(R5);
REF(B(2)) => PUSH(R5);
REF(B(0)) => PUSH(R5);
FADD(R5); COMMENT A + B ANSWER IN A;
Section XI. THE PL-11 'COMMON' FACILITY

1. INTRODUCTION

In order to enable separately compiled program modules to access a common set of variables easily, a COMMON facility exists in the PL-11 language. This facility is very similar to the named COMMON facility in FORTRAN IV, and is strictly compatible with named COMMON blocks in FORTRAN programs. However, in order to indicate clearly where these COMMON blocks are to be initialized, PL-11 uses a slightly different syntax than FORTRAN, and to avoid any confusion with FORTRAN syntax, PL-11 uses the reserved word SEGMENT instead of COMMON. Semantically, however, they mean the same thing and are used in the same manner.

If a FORTRAN subprogram was to have access to a COMMON block called ALPHA, containing the integers X, Y, Z and the 100-word integer arrays W and T, the programmer would include the following statements in his FORTRAN deck:

```fortran
COMMON /ALPHA/ X,Y,Z,W,T
INTEGER X,Y,Z,W(100),T(100)
```

Note that in FORTRAN the order of the variables in COMMON storage is determined by the order of their appearance in the COMMON declaration.

If a PL-11 module (which might contain any number of subroutines) also was to have access to this same COMMON block, the programmer would include the following statements in his PL-11 deck:

```pl11
EXTERNAL SEGMENT ALPHA;
INTEGER X,Y,Z;
ARRAY 100 INTEGER W,T;
```

In PL-11 the order of the variables in storage is determined by their order of appearance in the declarations which follow the EXTERNAL SEGMENT declaration. If this FORTRAN module and this PL-11 module were now linked together, the X in both modules would be the same X (i.e. the same word in storage), provided of course that FORTRAN stores its integers in one 16-bit PDP-11 word.

Similarly, the Y, Z, W, and T would be the same storage locations in both modules. Hence, these two modules would be able to transmit data between them implicitly through these COMMON data variables.

Of course, such communication is not limited to only two modules, nor only between FORTRAN and PL-11 modules. For example, if several PL-11 modules, each containing the above declarations, were linked together, then the X in one module would be the same as the X in all other modules.

Any number of variables of any type can be declared in a COMMON block -- the programmer is not limited to integers.

2. INITIALIZATION

One question immediately comes to mind: What values do these COMMON variables have initially (i.e. when the program is first loaded)? Or, phrased in another way: How can the programmer initialize these variables to the values he desires?
In FORTRAN, a special BLOCK DATA subprogram must be written. The sole purpose of this is to supply initial values for variables in named COMMON blocks -- there are no instructions in this "subprogram", which therefore is not really a "program" at all. It is not possible for the FORTRAN programmer to initialize COMMON variables in the main program, nor in a subroutine or function program that uses the COMMON block. If COMMON variables are not explicitly initialized in a block data subprogram, their initial values are undefined.

In PL-11, rather than require the programmer to write a separate module just for initializing COMMON variables, it is possible to initialize them directly in the modules that use them provided the word EXTERNAL is changed to GLOBAL in the declaration of the segment. For example, one module in the group of modules having COMMON access to ALPHA might contain:

```
GLOBAL SEGMENT ALPHA;
INTEGER X=1, Y, Z=32573;
ARRAY 100 INTEGER W=100*($77), T;
```

This would ensure that the initial values of all variables in segment ALPHA are well defined. In this case, X would be loaded with 1, Y with 0 (the default value), Z with 32573, each of the 100 words of W with 77 octal, and each of the 100 words of T with 0. Note that for variables declared in a GLOBAL segment, the normal PL-11 initialization rules apply. That is, if no initial values are explicitly specified, the compiler automatically generates an initial value of 0.

Clearly if ten PL-11 modules are to have common access to the segment ALPHA, nine of them should contain the declaration of ALPHA as an EXTERNAL SEGMENT (indicating that that module will not supply any initial values for this segment) and only one of them should declare ALPHA as a GLOBAL SEGMENT and supply the initial values. Which module contains the GLOBAL declaration is irrelevant, and should be chosen for the programmer's convenience. If none of the ten modules contains a GLOBAL definition of ALPHA (i.e. all ten modules declare ALPHA to be EXTERNAL), then the initial values of these variables will be undefined. Note that this is a very sharp departure from normal PL-11 variables, which can never be undefined. If more than one module contains a GLOBAL definition of ALPHA, then the initial values will be set to some set of initial values, but which set will depend on the order in which the modules are linked together by the linker. Since this order may depend on several factors, especially if some of the modules are on libraries, the programmer simply cannot predict which set of initial values will actually get loaded. Note that the linkage editor gives no error indication when this happens, and since the modules are compiled separately, there is no way for the compiler to check for this. Hence the user must pay careful attention to ensure that each segment is declared as a GLOBAL segment in one and only one module. Good programming practice would dictate that this module be the main module, and that all segments be declared as GLOBAL in that module.

3. SCOPE AND BLOCK STRUCTURE

3.1 Description

It is clear that COMMON segments violate the normal rules of block structure, or more correctly, that they are outside the normal rules of block structure. The scope of an EXTERNAL SEGMENT or GLOBAL SEGMENT declaration is the entire module in which it is declared, but the storage associated with these variables is not only accessible from anywhere within the module, but also from outside that module. The scope of that storage is really the entire core load. To indicate this clearly in the PL-11 language, all SEGMENTS, both EXTERNAL and GLOBAL, must be declared before anything else in the module. Since these SEGMENTS are outside the normal block structure of PL-11, these declarations must appear before the block structure starts, that is, before the first BEGIN of the module (literally outside the module's blocks!). This appears to be quite unusual at first, but does represent syntactically exactly the way these variables will be treated in the total core load.

3.2 Deck set-up

A CII job to compile a PL-11 module with SEGMENT declarations would have a deck set-up as in the following example:
EXTERNAL SEGMENT ALPHA;
    INTEGER X,Y,Z;
    ARRAY 100 INTEGER W,T;

EXTERNAL SEGMENT BETA;
    BYTE A,B,C;
    LOGICAL XL,YL,ZL;
    REAL Q,P;

GLOBAL SEGMENT CHARLIE;
    INTEGER I,J,K,L, M=5, N=22;
    REAL PI=3.14159, TEN=10.0;
    ARRAY BYTE MESSAGE = 'THIS IS ALL';

EXTERNAL SEGMENT DELTA;
    ARRAY 50 BYTE AA,BB;

BEGIN COMMENT THE FIRST BLOCK STARTS HERE.
    THE NORMAL PL-11 PROGRAM GOES HERE.
    IT CAN, OF COURSE, USE ANY OF THE
    VARIABLES DECLARED ABOVE;
END.

4. USEFUL DETAILS

The size of a COMMON segment is determined by the number and size of the variables declared in declarations following the SEGMENT declaration. A segment starts with the SEGMENT declaration, and ends with the next SEGMENT declaration or a BEGIN or PROCEDURE declaration. In the above example, the size of SEGMENT ALPHA is 406 bytes, the size of BETA is 18 bytes, CHARLIE is 32 bytes and DELTA is 100 bytes.

All COMMON variable names and SEGMENT names must be unique; that is, a variable cannot be declared in two SEGMENTS, nor can a SEGMENT have the same name as a variable or another SEGMENT. These names can be referenced anywhere in the module, and within the block structure, locally declared names can duplicate (and hence take precedence over) any of the COMMON names within any program block. The name of a SEGMENT can be referenced within the program, exactly as if it had been declared as an integer array of the length of the COMMON segment.

The order in which segments are declared within a module is totally irrelevant, and can differ from module to module. EXTERNAL and GLOBAL segments can be intermixed freely. Obviously a module only needs to declare those segments it needs access to, regardless of what segments are declared in any other module in the core load. If one module declares a segment to be of size I bytes, and a second module declares the same segment to be of a different size K bytes, then the linker will reserve MAX(I,K) bytes for the segment. The CII and DOS linkage editors give no indication that I is not equal to K, but the RSX-11D linker does. Good programming practice would indicate that segments should be declared to be the same size in all modules accessing them.

Any COMMON variable or SEGMENT can have synonyms of any type, and these synonyms are then also COMMON variables belonging to the same segment.
5. IMPLEMENTATION

Each SEGMENT declaration in PL-11 is translated by the compiler into the same linkage-editor code as the PAL-11 assembler generates for a DSECT directive labelled with the same name, and as the FORTRAN compiler generates for a named COMMON block with the same name.

Hence modules in any of these languages can access the same variables, each in its own unique way. Initialization values are generated by the compiler only for variables declared in a GLOBAL SEGMENT, as mentioned above.
Section XII. MEMORY MANAGEMENT INSTRUCTIONS

The four memory management instructions are provided in the model 45 primarily for use by the operating systems that use the memory segmentation hardware. In PL-11, these are represented as follows:

\[
\begin{align*}
\text{POP(\text{SP})} & \Rightarrow \text{PREVIOUS(OPERAND)}; & \text{COMMENT MTPI;} \\
\text{POP(\text{SP})} & \Rightarrow \text{PREVIOUSDATA(OPERAND)}; & \text{COMMENT MTPD;} \\
\text{PREVIOUS(OPERAND)} & \Rightarrow \text{PUSH(\text{SP})}; & \text{COMMENT MFPI;} \\
\text{PREVIOUSDATA(OPERAND)} & \Rightarrow \text{PUSH(\text{SP})}; & \text{COMMENT MFPD;} \\
\end{align*}
\]

The address of this operand is always determined in the current address space, but is always used in the previous address space.

The model 40 does not distinguish between "instruction" and "data" space, and only has the MTPI and MFPI hardware instructions. Therefore only

\[
\begin{align*}
\text{POP(\text{SP})} & \Rightarrow \text{PREVIOUS(OPERAND)}; \\
\text{PREVIOUS(OPERAND)} & \Rightarrow \text{PUSH(\text{SP})}; \\
\end{align*}
\]

can be used in PL-11 programs for the model 40. Of course, none of these instructions exist on the models 10 and 20.
Section XIII. 'CAMAC' PROGRAMMING IN PL-11

1. DESIGN PHILOSOPHY

There are three basic requirements that must be met by any CAMAC programming system if this system is to be an easily used and understood tool for effective use of CAMAC.

1) The programmer must have the ability to give meaningful symbolic names of his choice to any CAMAC C-N-A triple. The association of a name with a triple appears exactly once in any program (in PL-11, as a declaration at the beginning of the program), and all operations with this CAMAC module are done by referring to this symbolic name. Not only does this make the program more readable, it also makes modification of a CAMAC configuration extremely simple. One has only to change a single declaration and recompile the program. The compiler will then ensure that the addresses in all operations to this module are changed properly.

2) The programmer must have the ability to give meaningful symbolic names of his choice to any CAMAC function.

3) The programmer should be able to use the same syntactic notation for CAMAC variables and functions as he uses for ordinary program variables and functions. This includes the ability to test Q responses in the same manner as normal Boolean tests are made, the ability to index through arrays of CAMAC variables in the same manner as normal arrays are indexed, etc. Such a consistent use of a high-level language syntax makes it easy for a non-professional programmer to utilize CAMAC effectively, since he is able to manipulate CAMAC data in exactly the same way as he deals with normal data.

PL-11 is designed to be syntactically as high-level a language as possible so that readable, structured programs can be written easily. At the same time, PL-11 is semantically as close to the machine architecture as necessary in order to guarantee efficiency and the ability to utilize fully all features of the hardware.

Therefore, in addition to the three requirements listed above, there is a fourth requirement satisfied by the CAMAC facilities in PL-11:

4) The compiler should produce efficient, in-line object code for every CAMAC operation. There should be no subroutine-calling overhead and no required run-time system to support the CAMAC features of the language. This implies that an entire real-time operating system can be written in PL-11 without the use of any other language and without any necessary run-time environment.

2. DECLARING 'CAMAC' VARIABLES

The purpose of CAMAC variable declarations in PL-11 is to give programmer-defined symbolic names to CAMAC registers so that all operations to CAMAC can be done by reference to these names. The syntactic form of such declarations is analogous to the PL-11 construct that enables the programmer to give a symbolic name to any machine register, memory cell, or I/O device register.

```
INTEGER PRINTBUFFER SYN MEMORY($177566),
PRINTSTATUS SYN MEMORY($177564);
```

gives the symbolic name PRINTBUFFER to the memory cell with absolute address 177566 octal, and the name PRINTSTATUS to the cell with absolute address 177564 octal.
INTEGER CRTBUFFER SYN CRATE 1 STATION 2
   SUBADDRESS 14,
Crittatus SYN CRATE 1 STATION 5
   SUBADDRESS 0 GROUP 2;

gives the symbolic name CRTBUFFER to the CAMAC register in crate 1 station 2 subaddress 14
   group 1 (by default), and the name CITTSTATUS to the group 2 register at subaddress 0 station
   5 of crate 1. In such declarations, the compiler will accept only crate numbers in the
   range (1,7), station numbers in (0,31), subaddress numbers in (0,15), and group numbers in
   (1,2).

It is important to note that both memory and CAMAC integers are 16-bit quantities. In
keeping with the philosophy of designing a fully integrated system, it was decided very early
that since a PDP-11 word is only 16 bits, all CAMAC modules used in Omega would utilize only
the low-order 16 bits of the 24 data bits provided in CAMAC. This decision has proved to
be correct: all programming errors due to a mismatch of precision are eliminated, and no
time is lost packing and unpacking PDP-11 words. Since many 16-bit CAMAC modules are com-
mercially available, no penalty was paid in higher costs or lost data. Clearly this de-
cision also made it easier to integrate CAMAC variables into PL-11.

3. USING 'CAMAC' VARIABLES

   Once a CAMAC variable has been declared, it is used just like any other PL-11 variable.
   Assuming the declarations written above,

   \[ X \rightarrow \text{PRINTERBUFFER}; \]

   moves the contents of location X into location PRINTERBUFFER, which causes a character to be
typed on the teletype owing to the definition of PRINTERBUFFER as absolute address 177566 octal
   (the teletype data register).

   \[ X \rightarrow \text{CRTBUFFER}; \]

   writes the contents of location X into the CAMAC register CRTBUFFER, which presumably causes
the module at that CAMAC location to display this character on a CRT.

   \[ \text{PRINTSTATUS} \rightarrow Y; \]

   moves the contents of PRINTSTATUS (the teletype status register) into location Y.

   \[ \text{CRTSTATUS} \rightarrow Y; \]

   reads the contents of CAMAC register CRTSTATUS into memory location Y.

   Of course it is not necessary to store these status values in memory as they can be
tested directly as follows:

   \[ \text{WHILE PRINTSTATUS=BUSY DO;} \]

   is an empty wait loop for the teletype to become "not busy".

   \[ \text{WHILE CRTSTATUS=BUSY DO;} \]

   is a wait loop for the CAMAC display modules to indicate "not busy" in the same manner. To
test for an ordinary variable X in some range we can write:

   \[ \text{IF } X>\text{LOWLIMIT} \& X<\text{HIGHLIMIT THEN} \]
   \[ \text{BEGIN COMMENT WITHIN RANGE; } \ldots \text{ END;} \]

   If SCALER is declared as a CAMAC register, we can write a similar test:

   \[ \text{IF SCALER>LOWLIMIT \& SCALER<HIGHLIMIT THEN} \]
   \[ \text{BEGIN COMMENT WITHIN RANGE; } \ldots \text{ END;} \]

   If we wish to wait until SCALER reaches some value \text{THRESHOLD}, we write:
WHILE SCALER<THRESHOLD DO;

In all of these examples, the PL-11 compiler generates for each CAMAC reference the
two in-line instructions that are required by the manufacturer's CALL interface for a CAMAC
operation. The first instruction sets up the crate and function codes, the second sets up
the station and subaddress codes and actually performs the transfer. Hence this method is
"as efficient as possible" as well as being easy to use and understand.

4. 'CAMAC' ARRAYS AND BLOCK TRANSFERS

A block of 10 CAMAC registers starting at station 1 subaddress 0 of crate 1 can be de-
clared in PL-11 as:

ARRAY 10 INTEGER SCALER SYN
CRATE 1 STATION 1 SUBADDRESS 0;

If we also define a compile-time constant SPACING to be the address increment between con-
secutive SCALER modules:

EQUATE SPACING SYN 2;

then the statement:

FOR R1 FROM 0 STEP SPACING UPTO 9*SPACING DO
  0 => SCALER(R1);

will generate code to reset to zero 10 scalers in consecutive subaddresses at the same sta-
tion. The same statement will reset to zero 10 scalers in consecutive stations if we in-
stead declare:

EQUATE SPACING SYN 32;

In order to accumulate the sum of these scalers in a variable SUM with the average in
variable AVERAGE, we can write:

  0 => SUM;
  FOR R1 FROM 0 STEP SPACING UPTO 9*SPACING DO
    SUM += SCALER(R1);
  SLUM => AVERAGE/10;

If we wish to store the 10 scaler values in a 10-element array X, we can write:

  0 => R2;
  FOR R1 FROM 0 STEP SPACING UPTO 9*SPACING DO
    BEGIN
      SCALER(R1) => X(R2); R2 + 2;
    END;

where we have to keep an extra index register R2 to account for the constant spacing of 2
between consecutive integers in memory.

This last example demonstrates a transfer of 10 words in which each word transferred
requires a CAMAC read operation. If this scaler block included a control module that allowed
the entire block to be transferred with a single command to that module, this could also be
done in PL-11 as follows:

INTEGER BLOCKSTART SYN CRATE 1 STATION 3
  SUBADDRESS 0;
  REF(X) => BLOCKSTART;
where \( \text{REF}(X) \) is the memory address of array \( X \), and we assume that initializing the \text{BLOCKSTART} control module with a memory address will cause the block of scalers to be transferred into memory starting at that address. A more common situation is to include in the \text{CAMAC} control module a block size register, to define how many words to transfer. Then the block transfer is written in PL-11 as:

\[
\text{REF}(X) \to \text{BLOCKSTART}; \ 10 \to \text{BLOCKSIZE};
\]

5. \textit{OTHER 'CAMAC' FUNCTIONS}

The preceding sections demonstrated the \text{CAMAC} read, write, and reset functions in PL-11 and showed that they could be written with the same syntactic form as normal variable operations in PL-11. There are, however, 32 possible \text{CAMAC} functions, and any or all of these can be given symbolic names by PL-11 declarations such as the following:

\[
\text{CAMAC FUNCTION ENABLE(26),DISABLE(24),}
\text{TESTLAM(8),CLEARLAM(10)};
\]

where the symbolic name is chosen by the programmer, and the function number must be in the range \((0,31)\). These functions can be applied to any \text{CAMAC} variable using the normal function notation:

\[
\text{ENABLE (SCALER)};
\text{DISABLE (CRTSTATUS)};
\text{CLEARLAM (SCALER(R1+8))};
\]

For example, if we wish to read a block of scalers into an array \( X \) such that each scaler is read only when it responds to a \text{TESTLAM} function, we can write in PL-11:

\[
0 \to \text{R2};
\text{FOR R1 FROM 0 STEP SPACING UPTO 9*SPACING DO}
\text{BEGIN}
\text{WAIT: TESTLAM (SCALER(R1));}
\text{IF NOT Q THEN GOTO WAIT;}
\text{SCALER(R1) \to X(R2);}
\text{R2 + 2;}
\text{END;}
\]

where we have used the name \( Q \) which is a pre-declared PL-11 name for a bit in memory that is set or reset by the \text{CAMAC} CALL interface according to the \( Q \)-response of a \text{CAMAC} operation.

Interrupts caused by \text{CAMAC} modules can be handled in PL-11 by declarations such as

\[
\text{CAMAC INTERRUPT 20 PROCEDURE SCALEROVERFLOW;}
\text{BEGIN ... END;}
\]

This declaration is just an extension to \text{CAMAC} of the interrupt handling mechanism present in the PL-11 language. It causes the compiler to set up transfer vectors by which the operating system can call this procedure whenever an interrupt is caused by \text{LAM} bit 20. The interrupt number must be in the range \((1,24)\), and all interrupt enabling, disabling, clearing, and resolution of multiple \text{LAM} sources can and must be programmed explicitly with PL-11 functions.

6. \textit{CONCLUSION}

The \text{CAMAC} features integrated into PL-11 are efficient, easy to use, and easy to understand. They make the full range of \text{CAMAC} functions available to the programmer in a simple symbolic notation. The PL-11 compiler is able to generate highly efficient in-line code for each \text{CAMAC} operation owing to the simple design of the CALL interface between the PDP-11...
and CAMAC. For a more complex interface a compiler would probably have to generate calls to system subroutines, but the notation used by the programmer would remain unchanged (this is the technique used for FORTRAN floating-point operations on many small computers that lack floating-point hardware). It is suggested that such an extension of a widely used language, such as ALODL, FORTRAN, or PL/I, along these lines would be a highly desirable way of providing a universal tool for programming CAMAC.
PL-11 ERROR MESSAGES

Errors Pertaining to Declarations in General

1. The size of arrays and stacks must be a positive literal.
2. Register identifiers cannot be declared as arrays or stacks.
3. Literals cannot be declared.
4. Declarations must precede all other statements in the block.
5. Declarations cannot be labelled.
6. Bit arrays and stacks are not allowed.

Errors Pertaining to Synonym and Equate Declarations

11. References cannot be synonymed to arrays or stacks.
12. Equate identifiers can be synonymed only with literal expressions.
13. Variables cannot be synonymed with literals or expressions.
14. Synonyms and external common variables cannot be initialized.
15. Bit synonyms must select a specific bit in the word or byte.

Errors Pertaining to Bit Declarations and Operations

21. Bits cannot be selected from real, double or literal operands.
22. A bit selector must be a number N, C <= N <= (8, 16).
23. This set of bits cannot be operated on simultaneously.

Errors Pertaining to Initialization within Declarations

31. Initialization list exceeds storage assigned to the variable.
32. Repetition factor must be numeric literal expression.
33. Byte and bit variables cannot be initialized with addresses.
34. Initialization values must be literal expressions only.
35. Bit variables can be initialized with a single 0 or 1 only.

Errors Pertaining to Literals

51. Decimal or octal number too large for one 16-bit PDP-11 word.
52. Hexadecimal or octal literal has too many digits for 1 word.
53. Hexadecimal or octal literal with no digits.
54. String literal too long -- max of 255 characters allowed.
55. String literal with no characters.
56. Illegal character in a numeric literal.

Errors Pertaining to Models 40 and 45 Extended Instruction Set

71. This operator not available with this Hardware Configuration.
72. The previous left operand must be a non-byte register.
73. The previous left operand must be an even, non-byte register.
74. The previous left operand must be an odd, non-byte register.
75. The right operand of a maskflip must be a non-byte register.
76. Number of parameters in a call not equal to number declared.
77. Incorrect use of previous or previous data.
79. Exponents can only move to/from integer or logical variables.
80. This instruction is not available on this Hardware.
Errors Pertaining to REAL and DOUBLE

81 REAL AND DOUBLE OPERATIONS NOT AVAILABLE ON THIS HARDWARE
82 CANNOT USE PREVIOUS OPERATOR WITH REAL OR DOUBLE OPERANDS
83 LITERALS CANNOT BE USED IN REAL OR DOUBLE OPERATIONS
84 INCORRECT TYPE MIXING IN REAL OR DOUBLE OPERATIONS
85 LEFT OPERAND OF PREVIOUS OPERATION MUST BE REAL REGISTER 0-3
86 RIGHT OPERAND OF PREVIOUS MOVE MUST BE REAL REGISTER 0-3
87 THIS FLOATING-POINT FUNCTION NOT AVAILABLE ON THIS HARDWARE
88 INCORRECT ARGUMENT FOR THIS FLOATING-POINT FUNCTION
89 EXPONENT APPLIES ONLY TO A REAL OR DOUBLE REGISTER 0-3
90 ILLEGAL EXPONENT IN A REAL OR DOUBLE CONSTANT

Errors Pertaining to Procedures and Returns

91 A PROCEDURE DECLARATION MUST SPECIFY A REGISTER FOR LINKING
92 THIS RETURN ILLEGAL IN A PROCEDURE WITHOUT A RETURN REGISTER
93 PROCEDURE CALLS CAN ONLY HAVE LITERALS AS PARAMETERS
94 TRAP AND EMT DECLARATIONS MUST SPECIFY A NUMBER 0 <= N < 256
95 A PROCEDURE DECLARATION MUST INCLUDE CODE TO DEFINE IT
96 HARDWARE FUNCTIONS DO NOT HAVE AN ADDRESS OR LENGTH
97 PARAMETER STACKING CAN ONLY BE USED WITH LINK REGISTER RS
98 INTERRUPT PROCEDURES CANNOT BE CALLED
99 ILLEGAL ADDRESS OR NEW PS VALUE IN AN INTERRUPT PROCEDURE

Errors Pertaining to Data Expressions

100 VARIABLES OF TYPE BIT CANNOT BE USED IN EXPRESSIONS
101 VALUES CANNOT BE MOVED INTO A LITERAL
102 THE PREVIOUS OPERATOR CANNOT BE USED AS A MONADIC OPERATOR
103 THE PREVIOUS OPERATOR CANNOT BE USED AS A DYADIC OPERATOR
104 THE LEFT OPERAND IN AN OPERATION CANNOT BE A LITERAL
105 OPERATORS CANNOT BE USED IN PREFIX NOTATION
106 NEITHER BYTES NOR LITERALS CAN BE REFERENCED INDIRECTLY
107 THE PREVIOUS OPERATOR CANNOT BE USED IN BYTE EXPRESSIONS
108 EXPRESSIONS, REALS, AND BITS CANNOT BE REFERENCED INDIRECTLY
109 BYTES CANNOT BE MIXED WITH OTHER TYPES IN AN EXPRESSION
110 LITERAL VALUE TOO LARGE FOR 1 POP-11 BYTE -- 8 BIT MAXIMUM
111 LITERAL VALUE TOO LARGE FOR 1 POP-11 WORD -- 16 BIT MAXIMUM
112 PUSH AND POP OPERATE ONLY ON INTEGER OR LOGICAL REGISTERS
113 REGISTERS AND INDEXED VARIABLES DO NOT HAVE A REF OR LENGTH
114 LITERALS AND EXPRESSIONS DO NOT HAVE A REF OR LENGTH
115 THE PREVIOUS OPERATOR CANNOT BE USED IN LITERAL EXPRESSIONS
116 THE PREVIOUS OPERATOR CANNOT BE PERFORMED ON ADDRESSES
117 EXPRESSIONS OF ALL LITERALS GENERATE NO RUN-TIME CODE
118 ILLEGAL USE OF PARENTHESES IN AN EXPRESSION
119 TWO OR MORE LEVELS OF INDIRECT REFERENCING ARE NOT ALLOWED
120 IDENTIFIERS DEFINED AS LABELS CANNOT BE USED IN EXPRESSIONS

Errors Pertaining to Identifiers and Labels

121 UNDEFINED IDENTIFIER -- ALL NAMES MUST BE DECLARED BEFORE USE
122 THE PREVIOUS IDENTIFIER WAS DECLARED EARLIER IN THIS BLOCK
123 THIS LABEL PREVIOUSLY DEFINED IN THIS BLOCK
124 THIS IDENTIFIER IS NOT A LABEL AND CANNOT BE USED IN A GOTO
125 LITERALS CANNOT BE USED AS LABELS

Errors Pertaining to Conditions

131 BOTH OPERANDS OF A COMPARISON OPERATOR CANNOT BE LITERALS
132 CANNOT HAVE BOTH AND AND OR IN A SINGLE COMPOUND CONDITION

Errors Pertaining to Iteration Statements

141 ONLY REGISTERS OR VARIABLES CAN BE USED TO CONTROL FOR-LOOPS
142 FOR-LOOP NOT EXECUTABLE WITH THESE INITIAL AND LIMIT VALUES
Errors Pertaining to Case Statements

151 THE INDEX IN A CASE STATEMENT MUST BE A NON-BYTE REGISTER
152 CASE STATEMENT WITH NO CASES

Errors Pertaining to Subscripts

161 THE PREVIOUS OPERATION IS NOT ALLOWED IN A SUBSCRIPT
162 ONLY 1 REGISTER CAN BE SPECIFIED IN A SUBSCRIPT EXPRESSION
163 IDENTIFIERS IN THIS MODE CANNOT HAVE SUBSCRIPTS
164 BASED VARIABLES CANNOT HAVE A REGISTER SUBSCRIPT
165 ILLEGAL OPERAND IN A SUBSCRIPT EXPRESSION
166 LITERALS CANNOT BE SUBSCRIPTED

Errors Pertaining to Syntax in General

201 END-OF-FILE ENCOUNTERED -- INCOMPLETE PROGRAM DECK
202 INPUT CHARACTER NOT RECOGNIZED BY PL-11 SYNTAX
203 THE PREVIOUS STATEMENT IS UNLABELLED AND CANNOT BE REACHED
204 ILLEGAL SYNTAX -- THIS SEQUENCE OF SYMBOLS HAS NO MEANING
205 ILLEGAL SYNTAX -- THESE SYMBOLS CANNOT BE ADJACENT
206 SYNTAX -- MISSING PUNCTUATION OR OPERATOR BETWEEN IDENTIFIERS

Errors Pertaining to Overflow in PL-11 Tables

211 NAME-TABLE OVERFLOW -- TOO MANY IDENTIFIERS IN COMPILER TABLE
212 PROGRAM-TABLE OVERFLOW -- TOO MUCH CODE GENERATED BY COMPILER
213 LABEL-TABLE OVERFLOW -- TOO MANY LABELS IN COMPILER TABLE
214 PARSE STACK OVERFLOW -- NESTING TOO DEEP FOR COMPILER TABLES
215 ARRAY OR STACK TOO BIG TO BE ALLOCATED -- 1024 BYTE MAXIMUM
216 REDEFINE-TABLE OVERFLOW -- TOO MANY GLOBAL AND EXTERNAL NAMES
217 TOO MANY INTERRUPT PROCEDURES FOR COMPILER LOADING TABLES

Errors Pertaining to CAMAC Operations

221 CAMAC CRATE NUMBER OUT OF RANGE ALLOWED WITH THIS INTERFACE
222 A CAMAC STATION MUST BE A NUMBER IN THE RANGE (0, 31)
223 A CAMAC SUBADDRESS MUST BE A NUMBER IN THE RANGE (0, 15)
224 A CAMAC GROUP MUST BE A NUMBER IN THE RANGE (1, 2)
225 MONADIC OPERATORS CANNOT BE USED WITH CAMAC VARIABLES
226 DYADIC OPERATORS CANNOT HAVE 2 CAMAC VARIABLES AS OPERANDS
227 PREVIOUS OPERATOR CANNOT HAVE A CAMAC VARIABLE ON THE LEFT
228 A CAMAC FUNCTION MUST BE A NUMBER IN THE RANGE (0, 31)
229 THE LAST ARGUMENT OF THIS FUNCTION MUST BE A CAMAC VARIABLE
230 THE FIRST ARGUMENT OF THIS FUNCTION MUST BE A CAMAC VARIABLE
231 THE ARGUMENT OF THIS FUNCTION MUST BE A CAMAC VARIABLE
232 CAMAC VARIABLES CAN ONLY BE OF TYPE INTEGER OR LOGICAL

Errors Pertaining to PL-11 Common Segments

241 COMMON SEGMENT OF 0 LENGTH
242 ALL COMMON MUST BE DECLARED BEFORE ANY BEGINS OR PROCEDURES
243 TOO MANY COMMON SEGMENTS -- MAXIMUM OF 102 PER COMPILED
# Appendix B

## Reserved Words in PL-11

<table>
<thead>
<tr>
<th>WORD</th>
<th>WHERE USED</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>monadic &quot;absolute value&quot; operator in expressions</td>
</tr>
<tr>
<td>ANY</td>
<td>comparison operator in bit tests (conditional expressions)</td>
</tr>
<tr>
<td>ARRAY</td>
<td>keyword to reserve blocks of storage in declarations</td>
</tr>
<tr>
<td>BEGIN</td>
<td>delimiter to open a new program block</td>
</tr>
<tr>
<td>BIND</td>
<td>modifier for indirect referencing of byte data</td>
</tr>
<tr>
<td>BIT</td>
<td>keyword used to declare bit variables</td>
</tr>
<tr>
<td>BPPOP</td>
<td>modifier to remove a byte datum from the top of a stack</td>
</tr>
<tr>
<td>BPUSH</td>
<td>modifier to add a byte datum to the top of a stack</td>
</tr>
<tr>
<td>BYTE</td>
<td>keyword used to declare byte variables</td>
</tr>
<tr>
<td>CAMAC</td>
<td>modifier used to declare CAMAC functions or interrupt procedures</td>
</tr>
<tr>
<td>CASE</td>
<td>keyword for the multi-way &quot;case&quot; selection statement</td>
</tr>
<tr>
<td>CLEAR</td>
<td>operator used to set to zero a specified list of bits</td>
</tr>
<tr>
<td>COMMENT</td>
<td>keyword to start a comment</td>
</tr>
<tr>
<td>COMP</td>
<td>monadic &quot;ones complement&quot; operator in expressions</td>
</tr>
<tr>
<td>CRATE</td>
<td>modifier defining the crate number in CAMAC variable declarations</td>
</tr>
<tr>
<td>DIND</td>
<td>modifier for indirect referencing of double data</td>
</tr>
<tr>
<td>DIV</td>
<td>dyadic &quot;programmed&quot; divide operator in expressions</td>
</tr>
<tr>
<td>DO</td>
<td>connective that precedes the body of all loops</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>keyword used to declare double-precision floating-point variables</td>
</tr>
<tr>
<td>DOWTO</td>
<td>connective that precedes the limit expression in a decrementing FOR-loop</td>
</tr>
<tr>
<td>DPPOP</td>
<td>modifier to remove a double datum from the top of a stack</td>
</tr>
<tr>
<td>DPUSH</td>
<td>modifier to add a double datum to the top of a stack</td>
</tr>
<tr>
<td>ELSE</td>
<td>delimiter that precedes the false part of an IF statement</td>
</tr>
<tr>
<td>EMT</td>
<td>modifier to define an &quot;EMT&quot; procedure in declarations</td>
</tr>
<tr>
<td>END</td>
<td>delimiter to close a program block</td>
</tr>
<tr>
<td>EQUIATE</td>
<td>keyword for declaring symbolic compile-time constants</td>
</tr>
<tr>
<td>EXPONENT</td>
<td>keyword to refer to the exponent part of a REAL or DOUBLE register</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>modifier to define an external procedure in declarations</td>
</tr>
<tr>
<td>FOR</td>
<td>keyword for FOR-loop iteration statements</td>
</tr>
<tr>
<td>FROM</td>
<td>connective that precedes the initial value expression in FOR-loops</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>keyword used in declaring CAMAC functions</td>
</tr>
<tr>
<td>GLOBAL</td>
<td>modifier to define a global procedure in declarations</td>
</tr>
<tr>
<td>GOTO</td>
<td>keyword for branch or jump statements</td>
</tr>
<tr>
<td>GROUP</td>
<td>modifier defining the group number in CAMAC variable declarations</td>
</tr>
<tr>
<td>IF</td>
<td>keyword for conditional (IF) statements</td>
</tr>
<tr>
<td>IND</td>
<td>modifier for indirect referencing of integer data</td>
</tr>
<tr>
<td>INTEGER</td>
<td>keyword used to declare integer variables</td>
</tr>
</tbody>
</table>
interrupt modifier to define an interrupt procedure in declarations

lind modifier for indirect referencing of logical data

logical keyword used to declare logical variables

longshift dyadic operator indicating long arithmetic shift on the models 40 and 45

lpop modifier to remove a logical datum from the top of a stack

lpush modifier to add a logical datum to the top of a stack

maskflip dyadic operator indicating exclusive-OR on the models 40 and 45

maskoff dyadic operator to set selected bits to 0 in expressions

maskon dyadic operator to set selected bits to 1 in expressions

mult dyadic "programmed" multiply operator in expressions

none comparison operator in bit tests (conditional expressions)

not modifier to reverse the outcome of a status test (conditional expressions)

of connective that precedes the block in a case-selection statement

pop modifier to remove an integer datum from the top of a stack

previous keyword used in memory management operations

previusdata keyword used in memory management operations

procedure keyword to declare procedures

push modifier to add an integer datum to the top of a stack

real keyword used to declare single-precision floating-point variables

ref modifier to define address literals

rem dyadic "macro" remainder operator in expressions

return keyword for return-from-procedure statements

rind modifier for indirect referencing of real data

rotate dyadic operator indicating circular right shift on the models 40 and 45

rpop modifier to remove a real datum from the top of a stack

rpush modifier to add a real datum to the top of a stack

segment keyword to declare COMMON blocks

set operator used to set to one a specified list of bits

sha dyadic "macro" arithmetic shift operator in expressions

shifta dyadic "programmed" arithmetic shift operator in expressions

shiftl dyadic "programmed" logical shift operator in expressions

shl dyadic "macro" logical shift operator in expressions

sla monadic shift left arithmetic operator in expressions

slc monadic shift left circular operator in expressions

sra monadic shift right arithmetic operator in expressions

src monadic shift right circular operator in expressions

stack keyword to reserve blocks of storage in declarations

station modifier defining the station number in CAMAC variable declarations

step connective that precedes the increment value in FOR-loops

subaddress modifier defining the subaddress number in CAMAC variable declarations

snap monadic operator to exchange the bytes in a word

syn connective used in synonym and equate declarations

then delimiter that precedes the true part of an IF statement

trap modifier to define a "trap" procedure in a declaration

upto connective that precedes the limit expression in an increment FOR-loop

while keyword for while-cause iteration statements
APPENDIX C

PRE-DECLARED VARIABLES AND REGISTERS

These names can be used in a PI-11 program without being declared by the programmer.

1. REGISTERS

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>R0</td>
<td>integer register 0</td>
</tr>
<tr>
<td>R1</td>
<td>&quot; &quot; 1</td>
</tr>
<tr>
<td>R2</td>
<td>&quot; &quot; 2</td>
</tr>
<tr>
<td>R3</td>
<td>&quot; &quot; 3</td>
</tr>
<tr>
<td>R4</td>
<td>&quot; &quot; 4</td>
</tr>
<tr>
<td>R5</td>
<td>&quot; &quot; 5</td>
</tr>
<tr>
<td>SP</td>
<td>stack pointer -- integer register 6</td>
</tr>
<tr>
<td>PC</td>
<td>program counter -- integer register 7</td>
</tr>
</tbody>
</table>

2. MEMORY ADDRESSED BY REGISTERS

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>M0</td>
<td>integer at the address contained in R0</td>
</tr>
<tr>
<td>M1</td>
<td>&quot; &quot; &quot; &quot; &quot; &quot; &quot; R1</td>
</tr>
<tr>
<td>M2</td>
<td>&quot; &quot; &quot; &quot; &quot; &quot; &quot; R2</td>
</tr>
<tr>
<td>M3</td>
<td>&quot; &quot; &quot; &quot; &quot; &quot; &quot; R3</td>
</tr>
<tr>
<td>M4</td>
<td>&quot; &quot; &quot; &quot; &quot; &quot; &quot; R4</td>
</tr>
<tr>
<td>M5</td>
<td>&quot; &quot; &quot; &quot; &quot; &quot; &quot; R5</td>
</tr>
<tr>
<td>B0</td>
<td>byte at the address contained in R0</td>
</tr>
<tr>
<td>B1</td>
<td>&quot; &quot; &quot; &quot; &quot; &quot; &quot; R1</td>
</tr>
<tr>
<td>B2</td>
<td>&quot; &quot; &quot; &quot; &quot; &quot; &quot; R2</td>
</tr>
<tr>
<td>B3</td>
<td>&quot; &quot; &quot; &quot; &quot; &quot; &quot; R3</td>
</tr>
<tr>
<td>B4</td>
<td>&quot; &quot; &quot; &quot; &quot; &quot; &quot; R4</td>
</tr>
<tr>
<td>B5</td>
<td>&quot; &quot; &quot; &quot; &quot; &quot; &quot; R5</td>
</tr>
</tbody>
</table>

3. ABSOLUTE MEMORY

<table>
<thead>
<tr>
<th>Memory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEMORY</td>
<td>32768-word integer array, starting at absolute address 0</td>
</tr>
</tbody>
</table>

4. EXTENDED ARITHMETIC UNIT 'REGISTERS'

<table>
<thead>
<tr>
<th>Register</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>Accumulator -- integer at absolute address FEC2 (hex)</td>
</tr>
<tr>
<td>MQ</td>
<td>Multiplier-Quotient -- integer at absolute address FEC4 (hex)</td>
</tr>
<tr>
<td>SC</td>
<td>Step-Count -- byte at absolute address FEC8 (hex)</td>
</tr>
<tr>
<td>SR</td>
<td>Status-Register -- byte at absolute address FEC9 (hex)</td>
</tr>
</tbody>
</table>
5. **CONDITION CODE BIT NAMES**

<table>
<thead>
<tr>
<th>PL-11 name</th>
<th>PDP-11 hardware condition bit</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARRY</td>
<td>C-bit</td>
</tr>
<tr>
<td>OVERFLOW</td>
<td>0-bit</td>
</tr>
<tr>
<td>MINUS</td>
<td>N-bit</td>
</tr>
<tr>
<td>ZERO</td>
<td>Z-bit</td>
</tr>
</tbody>
</table>

6. **FLOATING-POINT REGISTERS FOR THE MODEL 45**

\[
\begin{align*}
AC0 \\
AC1 \\
AC2 \\
AC3 \\
AC4 \\
AC5 \\
\{ \\
DAC0 \\
DAC1 \\
DAC2 \\
DAC3 \\
DAC4 \\
DAC5
\end{align*}
\]

The registers have these names when used to hold single-precision quantities.

The floating-point registers have these names when used to hold double-precision quantities.
APPENDIX D

PRE-DECLARED 'EMT' PROCEDURES IN PL-11

In order to facilitate communication with the PDP-11 monitor, the following names have been pre-declared in the PL-11 compiler. These names may be used to call monitor functions from a PL-11 program without the necessity of declaring them. It is, however, the programmer's responsibility to ensure that the corresponding code to perform the desired functions has been loaded into the PDP-11 at run-time. These routines are automatically included in the standard PDP-11 monitor written by S. Lauper for the Omega Project. (See the "Omega On-line PDP-11 User's Manual").

1. I/O ROUTINES

1.1 READ

This is the standard input routine, and is defined as EMT code 11 octal. It can be used for either teletype or high-speed paper tape input, depending on the parameters.

1.1.1 Standard teletype input

This is expected to be the most common usage of this routine. The input unit is the teletype (or low-speed paper-tape reader that is connected to the teletype), and the end-of-message character is the "carriage return" key.

Parameters

\[\begin{array}{ll}
\text{R1} & \text{address of the input buffer} \\
\text{R0} & 0
\end{array}\]

Calling sequence

If we have declared an array called BUFFER as:

\[
\text{ARRAY 80 BYTE BUFFER;}
\]

then to read a line of at the most 80 characters that ends with a carriage return, we would simply write:

\[
0 \Rightarrow \text{R0}; \text{REF}(
\text{BUFFER}) \Rightarrow \text{R1}; \text{READ};
\]

Results

Control is not returned from the READ until a message, followed by a carriage return, has been typed. This message, as a sequence of 8-bit ASCII characters, is returned in the input buffer.

1.1.2 Non-standard teletype input

If it is desired to make some key other than "carriage return" indicate "end-of-message", the non-standard teletype input calling sequence must be used.
Parameters

R1   address of the input buffer
R0   address of a word containing the end-of-message character

Calling sequence

If we wish the end-of-message character to be a semicolon (;), we would declare in the
PL-11 program:

    INTEGER TERMINATOR = ';$';

Then to read a line of at the most 80 characters that ends with a semicolon, we write:

    REF(TERMINATOR) => R0; REF(BUFFER) => R1; READ;

Results

The results are exactly as in the standard teletype input case.

1.1.3 High-speed paper tape input

There is only a single method of reading a paper tape with the high-speed reader.

Parameters

R1   address of input buffer
R0   address of a word containing the negation of the end-of-message character

Calling sequence

If the end-of-message character is to be a semicolon (;), it must be in two's complement
form in order to indicate paper tape rather than teletype as the input device. This can be
done with a declaration such as:

    INTEGER TAPEMARK = ','- ;

The call is then:

    REF(TAPEMARK) => R0; REF(BUFFER) => R1; READ;

Results

The next record on the high-speed paper tape reader is read into BUFFER (the record is
terminated when the first semicolon is read), and then control returns to the calling program.

Warning:

In all uses of the READ routine, there is no check made during the read to see if there
is enough space in the buffer to hold the message. It is the programmer's responsibility
to allocate enough space in the buffer for the longest possible input message.

1.2 PRINT

This is the standard teletype output routine, and is defined as BMT code 10 octal.
There is at present no way to punch a paper tape using the high-speed paper tape punch.

Parameters

R1   address of the ASCII character string to be printed
R0   the number of bytes in the string
Calling sequence

To print the 11-character message HERE WE GO! we might declare in the PL-11 program:

```
ARRAY 11 BYTE MESSAGE = 'HERE WE GO!';
```

the call would then be:

```
11 => R0; REF(MESSAGE) => R1; PRINT;
```

Results

The message is typed on the teletype, followed automatically with a carriage return and line feed. Control is returned to the caller as soon as the typing has been initiated. In order to wait for the message to be typed, simply call PRINT again with a character count of 0. There should in general be no need to perform such a wait unless the buffer is to be re-used immediately after the call to PRINT.

1.3 **TELETIME**

This routine prints the current time on the teletype, and is defined as BMT code 12 octal.

Parameters

None.

Calling sequence

```
TELETIME;
```

Results

The time is printed in the form hh:mm (i.e. 13H30).

2. **DATA CONVERSION ROUTINES**

There are four data conversion routines to convert from 16-bit values to octal or decimal ASCII character strings and back.

2.1 **BINTODEC**

This routine converts one 16-bit integer (15 bits plus leading sign bit) into the equivalent 7-byte string of decimal digits. It is pre-declared as BMT code 15 octal.

Parameters

- R1 address of the binary word to be converted
- R0 address of the buffer for the character string

Calling sequence

To convert the integer X into a string in buffer BUFFER, we declare:

```
INTEGER X; ARRAY 7 BYTE BUFFER;
```

and then use the call:

```
REF(BUFFER) => R0; REF(X) => R1; BINTODEC;
```

Results

The character string returned in BUFFER will consist of up to five decimal digits, with leading zeros suppressed, preceded by a minus sign if X was negative, and followed by a single blank.
2.2 **BINTOCT**

This routine converts one 16-bit word into the equivalent 7-byte string of octal digits. It is pre-declared as EMT code 21 octal.

**Parameters**

- R1 address of the binary word to be converted
- R0 address of the buffer for the character string

**Calling_sequence**

To convert the logical bit pattern in Y into a string in BUFFER, we declare:

```
LOGICAL Y; ARRAY 7 BYTE BUFFER;
```

and then use the call:

```
REF(BUFFER) => R0; REF(Y) => R1; BINTOCT;
```

**Results**

The character string returned in BUFFER consists of six octal digits followed by a single blank. Leading zeros are not suppressed.

2.3 **DECTOBIN**

This routine converts an ASCII character string of decimal digits into one 16-bit integer (15 bits plus leading sign bit). It is defined as EMT code 22 octal. All leading and trailing blanks in the string are ignored, but embedded blanks are illegal. The string can be of any length, and must be terminated by a comma or carriage return. A leading sign (+ or -) is optional.

**Parameters**

- R1 address of the string of decimal digits
- R0 address of the word for the binary result

**Calling_sequence**

To convert the decimal characters in BUFFER into a binary value stored in X, we can declare:

```
INTEGER X; ARRAY 8 BYTE BUFFER = '-2751,';
```

and then use the call:

```
REF(X) => R0; REF(BUFFER) => R1; DECTOBIN;
```

**Results**

The value returned in X is the binary equivalent of the decimal number represented by the character string in BUFFER. The OVERFLOW status bit in the PDP-11 hardware is set if there are any illegal characters in the string, or if the value is not in the range [-32768, +32767].

2.4 **OCTTOBIN**

This routine converts an ASCII character string of octal digits into one 16-bit word. It is defined as EMT code 24 octal. All leading and trailing blanks in the string are ignored, but embedded blanks are illegal. The string can be of any length, and must be terminated by a comma or carriage return.
Parameters

R1  address of the string of octal digits
R2  address of the word for the binary result

Calling sequence

To convert the octal characters in BUFFER into the equivalent bit pattern in Y, we can declare:

\[
\text{LOGICAL } Y; \text{ARRAY 5 BYTE BUFFER} = '001,' ;
\]
and then use the call:

\[
\text{REF}(Y) \rightarrow \text{RO}; \text{REF(BUFFER)} \rightarrow \text{RI}; \text{OCTTOBIN};
\]

Results

The value returned in Y is the bit pattern represented by the octal digits in BUFFER. The OVERFLOW status bit in the POP-11 hardware is set if there are any illegal characters in the string, or if the pattern represents more than 16 bits of information. The bit pattern is right-justified in the word, with leading zeros supplied if necessary. Extra leading zeros in the character string BUFFER are ignored.
PROGRAMMING EXAMPLES

The following examples serve to demonstrate the utility of some PDP-11 features. They may not represent the most efficient method of solving a problem on the PDP-11.

1. To search an N-element integer array A to find the first element that matches a variable Y:

   FOR R1 FROM 0 STEP 2 UPTO N => R2 SLA - 2 WHILE Y /= A(R1) DO;

   Note that in this case the body of the loop is empty -- all the work is done by the iteration constructs of the language. The loop terminates when either a match is found or the table is exhausted, whichever happens first.

2. Another method of doing this, using stack operations, is:

   N => R1 SLA + REF(A);
   WHILE PUSH(R1) /= Y AND R1 > REF(A) DO;

   Again the loop body is empty, since the iteration mechanism does all the work. Note that if there are two elements in A that match Y, this method will find the one with the highest subscript, whereas the first method will find the one with the lowest. The first method, rewritten to find the highest subscript element would be:

   FOR R1 FROM N => R1 SLA - 2 STEP - 2 DOWNTO 0 WHILE Y /= A(R1) DO;

   which is more efficient since i) the limit is 0, ii) only one register is used.
### APPENDIX F

#### 63 ASCII CHARACTERS RECOGNIZED IN PL-11 STRINGS

<table>
<thead>
<tr>
<th>Keypunch symbol</th>
<th>CII print symbol</th>
<th>ASCII internal dec</th>
<th>hex</th>
<th>oct</th>
<th>EBCDIC internal dec</th>
<th>hex</th>
<th>oct</th>
</tr>
</thead>
<tbody>
<tr>
<td>space</td>
<td></td>
<td>32  20  040</td>
<td></td>
<td></td>
<td>64     40  100</td>
<td></td>
<td></td>
</tr>
<tr>
<td>:</td>
<td></td>
<td></td>
<td>33  21  041</td>
<td></td>
<td></td>
<td>79     4F  117</td>
<td></td>
</tr>
<tr>
<td>&quot;</td>
<td></td>
<td>34  22  042</td>
<td></td>
<td></td>
<td>127    7F  177</td>
<td></td>
<td></td>
</tr>
<tr>
<td>#</td>
<td>#</td>
<td>35  23  043</td>
<td></td>
<td></td>
<td>123    7B  173</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>36  24  044</td>
<td></td>
<td></td>
<td>91     5B  133</td>
<td></td>
<td></td>
</tr>
<tr>
<td>%</td>
<td>%</td>
<td>37  25  045</td>
<td></td>
<td></td>
<td>108    6C  154</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>&amp;</td>
<td>38  26  046</td>
<td></td>
<td></td>
<td>80     50  120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>'</td>
<td>'</td>
<td>39  27  047</td>
<td></td>
<td></td>
<td>125    7D  175</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td>40  28  050</td>
<td></td>
<td></td>
<td>77     4D  115</td>
<td></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>)</td>
<td>41  29  051</td>
<td></td>
<td></td>
<td>93     5D  135</td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>42  2A  052</td>
<td></td>
<td></td>
<td>92     5C  134</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>43  2B  053</td>
<td></td>
<td></td>
<td>78     4E  116</td>
<td></td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>,</td>
<td>44  2C  054</td>
<td></td>
<td></td>
<td>107    6B  153</td>
<td></td>
<td></td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>45  2D  055</td>
<td></td>
<td></td>
<td>96     60  140</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.</td>
<td>.</td>
<td>46  2E  056</td>
<td></td>
<td></td>
<td>75     4B  113</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td>47  2F  057</td>
<td></td>
<td></td>
<td>97     61  141</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>48  30  060</td>
<td></td>
<td></td>
<td>240    F0  360</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>49  31  061</td>
<td></td>
<td></td>
<td>241    F1  361</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>50  32  062</td>
<td></td>
<td></td>
<td>242    F2  362</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>51  33  063</td>
<td></td>
<td></td>
<td>243    F3  363</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>52  34  064</td>
<td></td>
<td></td>
<td>244    F4  364</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>53  35  065</td>
<td></td>
<td></td>
<td>245    F5  365</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>54  36  066</td>
<td></td>
<td></td>
<td>246    F6  366</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>55  37  067</td>
<td></td>
<td></td>
<td>247    F7  367</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>56  38  070</td>
<td></td>
<td></td>
<td>248    F8  370</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>57  39  071</td>
<td></td>
<td></td>
<td>249    F9  371</td>
<td></td>
<td></td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>58  3A  072</td>
<td></td>
<td></td>
<td>122    7A  172</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>;</td>
<td>59  3B  073</td>
<td></td>
<td></td>
<td>94     5E  156</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>60  3C  074</td>
<td></td>
<td></td>
<td>76     4C  114</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>=</td>
<td>61  3D  075</td>
<td></td>
<td></td>
<td>126    7E  176</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>&gt;</td>
<td>62  3E  076</td>
<td></td>
<td></td>
<td>110    6E  156</td>
<td></td>
<td></td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>63  3F  077</td>
<td></td>
<td></td>
<td>111    6F  157</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>a</td>
<td>64  40  100</td>
<td></td>
<td></td>
<td>124    7C  174</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Keypunch symbol</td>
<td>CII print symbol</td>
<td>ASCII internal</td>
<td></td>
<td></td>
<td></td>
<td>EBCDIC internal</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------</td>
<td>----------------</td>
<td>------</td>
<td>------</td>
<td>------</td>
<td>----------------</td>
<td>------</td>
</tr>
<tr>
<td>A</td>
<td>A</td>
<td>65 41 101</td>
<td></td>
<td></td>
<td></td>
<td>195 C1 301</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>66 42 102</td>
<td></td>
<td></td>
<td></td>
<td>194 C2 302</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>67 43 103</td>
<td></td>
<td></td>
<td></td>
<td>195 C3 303</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>68 44 104</td>
<td></td>
<td></td>
<td></td>
<td>196 C4 304</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>69 45 105</td>
<td></td>
<td></td>
<td></td>
<td>197 C5 305</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>70 46 106</td>
<td></td>
<td></td>
<td></td>
<td>198 C6 306</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>G</td>
<td>71 47 107</td>
<td></td>
<td></td>
<td></td>
<td>199 C7 307</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>H</td>
<td>72 48 110</td>
<td></td>
<td></td>
<td></td>
<td>200 C8 310</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>I</td>
<td>73 49 111</td>
<td></td>
<td></td>
<td></td>
<td>201 C9 311</td>
<td></td>
</tr>
<tr>
<td>J</td>
<td>J</td>
<td>74 4A 112</td>
<td></td>
<td></td>
<td></td>
<td>209 D1 321</td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>K</td>
<td>75 4B 113</td>
<td></td>
<td></td>
<td></td>
<td>210 D2 322</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>L</td>
<td>76 4C 114</td>
<td></td>
<td></td>
<td></td>
<td>211 D3 323</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>M</td>
<td>77 4D 115</td>
<td></td>
<td></td>
<td></td>
<td>212 D4 324</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>78 4E 116</td>
<td></td>
<td></td>
<td></td>
<td>213 D5 325</td>
<td></td>
</tr>
<tr>
<td>O</td>
<td>O</td>
<td>79 4F 117</td>
<td></td>
<td></td>
<td></td>
<td>214 D6 326</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>P</td>
<td>80 50 120</td>
<td></td>
<td></td>
<td></td>
<td>215 D7 327</td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>Q</td>
<td>81 51 121</td>
<td></td>
<td></td>
<td></td>
<td>216 D8 330</td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>R</td>
<td>82 52 122</td>
<td></td>
<td></td>
<td></td>
<td>217 D9 331</td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>83 53 123</td>
<td></td>
<td></td>
<td></td>
<td>226 E2 342</td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>84 54 124</td>
<td></td>
<td></td>
<td></td>
<td>227 E3 343</td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>85 55 125</td>
<td></td>
<td></td>
<td></td>
<td>228 E4 344</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>V</td>
<td>86 56 126</td>
<td></td>
<td></td>
<td></td>
<td>229 E5 345</td>
<td></td>
</tr>
<tr>
<td>W</td>
<td>W</td>
<td>87 57 127</td>
<td></td>
<td></td>
<td></td>
<td>230 E6 346</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>88 58 130</td>
<td></td>
<td></td>
<td></td>
<td>231 E7 347</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>89 59 131</td>
<td></td>
<td></td>
<td></td>
<td>232 E8 350</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>Z</td>
<td>90 5A 132</td>
<td></td>
<td></td>
<td></td>
<td>233 E9 351</td>
<td></td>
</tr>
<tr>
<td>[</td>
<td></td>
<td>91 5B 133</td>
<td></td>
<td></td>
<td></td>
<td>74 4A 112</td>
<td></td>
</tr>
<tr>
<td>\</td>
<td></td>
<td>92 5C 134</td>
<td></td>
<td></td>
<td></td>
<td>224 E0 340</td>
<td></td>
</tr>
<tr>
<td>]</td>
<td></td>
<td>93 5D 135</td>
<td></td>
<td></td>
<td></td>
<td>90 5A 132</td>
<td></td>
</tr>
<tr>
<td>^</td>
<td></td>
<td>94 5E 136</td>
<td></td>
<td></td>
<td></td>
<td>95 5F 137</td>
<td></td>
</tr>
<tr>
<td>—</td>
<td></td>
<td>95 5F 137</td>
<td></td>
<td></td>
<td></td>
<td>109 6D 155</td>
<td></td>
</tr>
</tbody>
</table>
COMPILER OPTIMIZATIONS

In general, of course, it is meaningless to talk of "optimization" for a language of the type such as is PL-11, since for example the sequence:

\[ A + R1 \Rightarrow B \times \text{IND}(D) + E = \text{COMP}; \]

translates directly and uniquely into machine instructions. However, as any machine language programmer knows, there are cases where the programmer himself does the optimization; for example, when testing if the result of a calculation is zero it is not necessary to do a comparison, since the condition codes will already be set. Detailed here are the specific optimizations that the compiler makes.

1. 'GOTO' STATEMENTS

i) The compiler attempts to generate short (1 word) branches whenever possible, and employs a fairly elaborate "fix-up" scheme so that both forward and backward references will be short if possible.

ii) When used in IF statements, the compiler generates conditional branches that transfer control directly to the label (rather than conditional branches which transfer control to an unconditional branch that transfers control to the label). Moreover, these will be short wherever possible under the "fix-up" scheme of the compiler.

2. COMPARISON STATEMENTS

If the operand specified on the right side of any comparison is the literal zero (0), the compiler will not generate a "compare" instruction, but instead will generate the more efficient "test" instruction. Further, whenever the data expression on the left of the comparison requires evaluation at run-time (i.e. it contains arithmetic operators on non-literals) and the right operand is literal zero, neither a "compare" nor a "test" instruction will be generated, since the expression evaluation will set the condition code properly.

Thus it is slightly more efficient for the programmers to compare against zero than against any other value. For example, to take advantage of this:

a) instead of \( A \geq 1 \), use \( A > 0 \)

b) instead of \( B \leq -1 \), use \( B < 0 \)

3. 'CASE' STATEMENTS

i) A RETURN statement appearing explicitly in a CASE statement or an IF statement suppresses the automatic generation of the immediately following branch statement that normally follows statements within the CASE or IF. This also holds for an RTI or RIT instruction.

ii) If the \( k \)th case of a CASE statement is an unlabelled GOTO, the compiler places the destination address of that GOTO directly in the \( k \)th position of the branch table created for that CASE, and generates no code for that statement (i.e. there is no jump to a jump in such cases).
iii) If the $k$th case of a CASE statement is empty (either an empty BEGIN ... END or simply a semi colon), no code is generated and the address of the statement following the CASE statement is placed in the $k$th position of the branch table created for that CASE.

iv) If all the cases of a CASE statement are procedure calls without parameters and with the same link register, the procedure addresses are entered directly into the branch table created for that CASE and the jump indirect instruction is changed into an indirect jump-to-subroutine instruction.

4. 'FOR' STATEMENTS

i) If the step value $E2$ is the literal 1, the compiler generates an INC instead of an ADD.

ii) If the step value $E2$ is the literal -1, the compiler generates a DEC instead of an ADD.

iii) If the DOWNTO limit value $E3$ is the literal 0 or 1, the compiler will not generate the unnecessary compare instruction.

iv) If the UPTO limit value $E3$ is the literal 0 or -1, the compiler will not generate the unnecessary compare instruction.

v) If both the initial value $E1$ and the limit value $E3$ are literals, the compiler will not generate the extra branch necessary to test the loop the first time, but will issue a warning message if the loop can never be performed.

vi) The hardware SOB instruction will be generated on the PDP-11 40 and 45 models, for PL-11 loops that have the following form:

```
   FOR REG FROM EXPR STEP -1 DOWNTO 1 DO
       BEGIN
           ...
           ...
       END;
```

(where REG is any integer or logical register and EXPR is any expression) provided that the loop body does not contain more than $2^4 = 64$ instruction words.

5. PROCEDURE 'RETURN's

A return statement is automatically generated at the end of a procedure only if control can reach the end. In particular, if the user explicitly writes RETURN as the last statement of his procedure, a second RETURN will not be generated automatically.

Finally, any 'no-op' branches (i.e. a branch to the next instruction) appearing anywhere in the program are automatically detected and eliminated by the compiler.
APPENDIX H

DIFFERENCES BETWEEN DIFFERENT PDP-11 MODELS

The following remarks summarize the difference between the PDP-11/20 and the PDP-11/40 and PDP-11/4S as reflected in PL-11 operators.

1) Operators for the models 40 and 45 only. No equivalents for these operators exist on the model 20:

   LONGSHIFT
   MASKFLIP
   ROTATE

2) Operators that are used differently on the models 40 and 45 than on the model 20 with EAU:

   *
   /  
   DIV  
   MULT  
   REM  
   SHIFTA  
   SHA

3) Operators that do not exist on the models 40 and 45. These operators do exist on the model 20 with EAU:

   SHIFTL  
   SHL