PROTEL

Procedure Oriented Type Enforcing Language

Introductory Manual

Version 2026

Derived from the April 1987 revision

Resurrected and enhanced for the 21st century

by

Vincente D'Ingianni
vincente@binary-systems.com

former Advisor and Manager,
Bell-Northern Research / Nortel DMS-500 Development

Table of Contents

Preface 6

1.0 Introduction 8

Part I: Basic Structure of the Language 8

1.1 Input Format 8

1.2 Comments 8

1.3 Types of Symbols 8

1.3.1 Keywords 9

1.3.2 Special Symbols 9

1.3.3 Identifiers 10

1.3.4 Numbers 10

1.4 Constants 11

1.4.1 Simple Constant Expressions 11

1.4.2 Aggregate Constants (Tuples) 11

1.4.3 Character Strings 12

Part II: Rules of the Language 13

1.5 General Overview of Language Rules 13

2.0 Declarations 14

2.1 Type Declarations 14

2.1.1 Numeric Range 15

2.1.2 Symbolic Range 16

2.1.3 Boolean 16

2.1.4 Pointer 16

2.1.5 Sets 17

2.1.6 Tables 18

2.1.7 Structures 19

2.1.8 Areas 21

2.1.9 Descriptors 22

2.1.10 Procedures 23

2.1.11 Pack Attribute 24

2.2 Data Declarations 25

2.2.1 Variable Declarations 25
2.2.2 Constant Declarations 27
3.0 Executable Statements 29
3.1 Expressions 29
3.1.1 Operators 31
3.1.1.1 Assignment Operator 31
3.1.1.2 Arithmetic Operators 31
3.1.1.3 Comparison Operators 32
3.1.1.4 Logical Operators 32
3.1.1.5 Unary Operators 34
3.1.2 Operands 35
3.1.2.1 Identifiers 36
3.1.2.2 Selection 36
3.1.2.3 Indexing 36
3.1.2.4 Multiple 37
3.1.2.5 Dereference 37
3.2 Control Statements 38
3.2.1 RETURN Statement 39
3.2.2 IF Statement 40
3.2.3 Nested IF Statements 41
3.2.4 CASE Statement 43
3.2.5 SELECT Statement 44
3.2.6 Iterative Statements 45
3.2.7 The OVER Specification 47
3.2.8 Iterative Statement with the WHILE Condition 49
3.2.9 EXIT Statement 50
4.0 Parameter Passing Mechanisms 51
4.1.1 Formal and Actual Parameters 51
4.1.2 The VAL Attribute 54
4.1.2.1 Common Errors 55
5.0 PROTEL Program Organization 56
5.1 Introduction 56

5.1.1 Declaration Versus Definition 56
5.2 Sections, Modules, and Programs 58
5.2.1 Sections 58
5.2.2 Imbedding and Outbedding 58
5.2.3 Modules 58
5.2.4 Programs 59
5.3 Compilation Order 60
5.3.1 Compilation Order for Programs 60
5.3.2 Compilation Order for Modules 62
5.4 Imbedding 64
5.4.1 Differences Between USES and OF Imbedding 64
5.4.2 Imbedding of Non-Top-Level Sections 66
5.4.3 Private Interfaces 66
5.5 PROTEL Language 67
5.5.1 Design Goals 67
5.5.2 A Sample Program 69
5.5.3 Program Organization: Points to Remember 70
6.0 Summary 71
7.0 PROTEL 2026 Development Tools 72
7.1 The PROTEL Compiler - Pc 72
7.2 The PROTEL Beautifier - Pb 72
7.2 EMACS PROTEL Mode 73
7.3 Foreign Language Interoperability 76
7.3.1 Strings and the C boundary 76
7.3.2 Calling foreign code from PROTEL (EXTERNAL) 76
7.3.3 Publishing PROTEL routines for foreign callers (EXPORT) 78
7.3.4 Complete minimal examples 80
7.3.5 ENTRY vs EXPORT 82
7.3.6 Quick reference 82
7.3.7 Troubleshooting 82
Appendix A. Keywords 84
Appendix B. Special Symbols 85

Preface

From 1988 through 1998, I had the privilege of developing cutting edge real time software on state of the art communication systems. I regularly sifted through millions of lines of code with extremely primitive tools by today's standards. As the team leader for Carrier AIN, we had claim to the first feature delivered to the field using the object oriented extensions in PROTEL 2.

When I taught Introduction to PROTEL and DMS SOS classes at BNR/Nortel in the early 1990s, I would often get questions from new hires asking, "Why don't we use C and Unix?".

Besides the fact that C and Unix were created at our competitor AT&T, there were many excellent features of PROTEL that made it superior to C. These features protected developers from stupid mistakes and enforced good programming structure and readability without losing efficiency.

Modern programming languages in the 21st century have lost their way with overuse of classes and inheritance, inefficient code, and ridiculous overhead requirements. This is evident in the resurgence of procedural programming over object oriented programming methodology. Strong typing aspects allow errors to be caught at compile time, not run time.

Its all in the name, Procedure Oriented Type Enforcing Language.

This is not retro computing, it is smart computing.

This manual was recreated to resurrect a fantastic language that was left for dead through bankruptcy and corporate mismanagement. Through in-depth web searches, foggy memory, and a lot of Artificial Intelligence, this new PROTEL Introductory Manual will give the next generation of software engineers the opportunity to experience a programming language that was ahead of its time.

This new PROTEL Introductory Manual is dedicated to the BNR Compiler Team and all of the Nortel developers that produced over 50 million lines of PROTEL code that have been hidden from the public eye. Much of this code is still in use in the 21st century. They created the modern communications network that we enjoy today, but get very little public recognition. These guys did the heavy lifting with the creation and maturation of this programming language.

Finally, this PROTEL Introductory Manual and the PROTEL Reference Manual are the foundation of a new public domain compiler implementation that will finally give those new hires what they needed — a modern version of PROTEL that fully integrates with C/C++ and UNIX / Linux operating systems.

Key updates in PROTEL 2026 include;

- Case sensitivity by default for identifiers (types, variables, procedures, etc.), with keywords required in UPPERCASE

- Backward compatibility via compiler option --classical for case-insensitivity (matching original Nortel DMS behavior)

- Modern compilation using GCC backend for Mac OS X and UNIX / Linux, with interoperability for calling/linking C code

This manual is structured similarly to the original OFLPI for familiarity, with additions for object oriented extensions and modern features.

1.0 Introduction

PROTEL is a high level language developed specifically for use in switching systems. This manual is an introduction to PROTEL. It provides an overview of the language and is designed to be read in conjunction with the PROTEL Reference Manual.

In particular, the following aspects of PROTEL are not discussed in this document

- The special declarative keywords PROTECTED, SHARED and PRIVATE
- The scope rules
- The type compatibility rules
- Overlays
- The CAST construct
- BIND and WITH declarations
- Module types

The manual is divided into two parts. The first of these describes the basic structure of the language while the second details the language rules. The latter section is divided into subsections corresponding to different types of PROTEL statements. Program examples are provided to illustrate the points being discussed.

Part I: Basic Structure of the Language

1.1 Input Format

The language source is free form input between columns 1 and 110 (inclusive) of the input record. Embedded blanks are permitted anywhere except within numbers, identifiers, special symbols or keywords. Character strings may span several input records, although this is not recommended.

Compiler directives are denoted by a dollar sign ($) in column one followed by a command word (e.g., $EJECT). Therefore identifiers which begin with a dollar sign must not occur beginning on column one.

1.2 Comments

Comments are prefixed by a percent sign (%) and extend to the end of the input record. They may begin anywhere in the record and may be preceded by source text on the same line.

1.3 Types of Symbols

Four basic types of symbols are used in PROTEL. They are:

1. Keywords
2. Special Symbols

3. Identifiers
4. Numbers

1.3.1 Keywords

The keywords of the language are reserved, and may not be used as identifiers. Each keyword must be delimited by non-alphanumeric characters.

Some examples of keywords are:

```
BLOCK
BOOL    (Boolean)
DCL     (Declare)
DO
IF
INIT    (Initialize)
STRUCT  (Structure)
TRUE
```

Figure 24 contains a complete list of PROTEL keywords. Keywords will be introduced as required while discussing various features of the PROTEL language. Throughout the rest of this manual keywords will appear in upper case. This makes the programs easier to read and does not imply that keywords must be entered in upper case. PROTEL is indifferent to case, except within character strings.

In the following PROTEL statement, keywords are in **bold** font:

```
DCL factorial PROC(i integer) RETURNS integer IS
BLOCK
    :
    :
    :
ENDBLOCK;
```

1.3.2 Special Symbols

An alphanumeric character is either a letter or a digit. The alphanumeric characters in PROTEL are:

A, B, C, ..., Z, 0, 1, 2, ..., 9.

Sequences of one or more non-alphanumeric characters constitute the special symbols of the language.

The following are examples of special symbols.

- < less than
- = equals
- ; semicolon

| or
-> assignment

A complete list of special symbols can be found in Figure 25. The use of these symbols will become clear as various aspects of the language are discussed.

1.3.3 Identifiers

An identifier is a sequence of characters used to name a data type, constant, variable, procedure, statement label, or section. Identifiers consist of a letter, the dollar sign ($), or the underscore character (_) followed by up to 31 alphanumeric characters, dollar signs, or underscore characters.

Some examples of identifiers are:

```
weekday
n
AMOUNT_IN_$
NUMBER_35
end_of_file
$AMOUNT
```

In the following PROTEL statement the identifiers are underlined:

```
DCL factorial PROC(i integer) RETURNS integer IS
BLOCK
   :
   :
   :
ENDBLOCK;
```

1.3.4 Numbers

Decimal numbers are represented by a string of decimal digits. Hexadecimal numbers are denoted by the symbol # followed by a string of hexadecimal digits (digits 0 to 9, letters A to F). Negative numbers are prefixed by a minus sign.

EXAMPLES

```
#41     % hexadecimal number equal to decimal 65.

65     % decimal number.

-17     % negative decimal number.
```

1.4 Constants

1.4.1 Simple Constant Expressions

A simple constant expression is an expression, as described in "Expressions", which can be evaluated at compile time and which results in a single value.

EXAMPLES

```
5

TRUE

2 * (size - 1)      % size is a constant
```

1.4.2 Aggregate Constants (Tuples)

Tuples are used to represent structured constants which include several values. A constant tuple consists of a sequence of constant expressions enclosed by braces. The constant expressions may themselves be tuples; thus nesting of tuples is allowed.

A tuple may be preceded by a repetition factor which must be a numeric compile time constant. Such a factor indicates that what follows is to be treated as a single tuple where the elements are repeated the given number of times.

EXAMPLES

```
{1, 3, 5, 7, 9}

{size, 3 * size, length - 1}

3{2, 4}         % Equivalent to {2, 4, 2, 4, 2, 4}

{2, {3,4}}      % Outer tuple has two elements;
                % inner tuple has two elements.
```

1.4.3 Character Strings

A character string is a sequence of characters enclosed in single quotes (apostrophes). A single quote within a character string is represented as a pair of adjacent single quotes. A number preceding a character string indicates that the string is to be repeated a given number of times.

EXAMPLES

```
'error'

'out of range'

'don''t'     % The string is "don't".

2 'ab'       % Equivalent to 'abab'.
```

A character string is treated as a tuple of numeric values corresponding to the characters of the string.

For example, on a machine which uses the ASCII character set, the following three tuples are identical:

```
'AB'          % ASCII
{#41, #42}    % hex
{65,66}       % decimal
```

Part II: Rules of the Language

1.5 General Overview of Language Rules

A PROTEL program consists of a number of statements constructed from the basic symbols described previously.

The program in Figure 1 illustrates some of the statement types used in PROTEL. Each statement type will be discussed in more detail in subsequent sections.

NOTE: The numbers preceding the statements are for reference purposes only and are not part of the PROTEL program.

The types and variables used in the program are defined in statements 1,5 and 7. The procedure for calculating the factorial of a number is defined in statements 5 through 16.

In statement 9, the initial value of n is assigned to variables i and result.

The value of i is decremented by one and the new value assigned to i (statement 10). The new value is compared with 1. If it is greater, then the variable i is multiplied by the variable result and the new value assigned to result (statement 13).

Statements 10 through 14 are repeated as long as the value of i-1 is greater than 1 (statement 10). When the value of i-1 is no longer greater than 1, the value of result (i.e., the factorial of n) is returned to the point of invocation of the procedure (statement 15).

```
  1   TYPE integer {-32768 TO 32767};
  2
  3   % calculates the factorial of a number n
  4
  5   DCL factorial PROC(n integer) RETURNS integer IS
  6     BLOCK
  7        DCL result, i integer;
  8
  9        n -> i -> result;
 10        WHILE (i - 1 -> i) > 1
 12           DO
 13             i * result -> result;
 14           ENDDO;
 15        RETURN result;
 16     ENDBLOCK;
```

Figure 1. Simple Program

2.0 Declarations

A PROTEL program consists of type declarations and data declarations Since every datum has a type, we call a datum an instance of its type.

Named types are declared using TYPE statements. Named data are declared using DCL statements.

2.1 Type Declarations

EXAMPLES

```
TYPE byte {0 TO 255};

TYPE integer {-32768 TO 32767};

TYPE line_status {on_hook, off_hook};

TYPE end_of_file BOOL;

TYPE time_interval
   STRUCT
      value {0 TO 255},
      unit  {tenms, sec, mins, hours}
   ENDSTRUCT;

TYPE line_table TABLE [0 TO 99] OF BOOL;

TYPE small_non_negative_number byte;     % a defined type
```

DESCRIPTION

Type declarations are used to describe the characteristics of data. The fundamental characteristic of a type is a definition of the set of values which an instance of the type may assume. We refer to this as the value-set of the type. Other characteristics include the amount of storage it requires and whether its value may be updated.

Types can be defined in terms of existing primitive types in the language or in terms of other types defined by the programmer. The primitive types consist of the simple types, the aggregate types and the procedure type.

small_non_negative_number is defined in terms of user-defined types, not in terms of primitive types. It is an example of a defined type.

Once a type is declared, it can be used in any context where a type is required. The declaration of data in terms of types allows the compiler to do extensive type checking at compile time.

The simple types are used to declare data representing "atomic", or single values.

The simple types are:

- numeric range
- symbolic range
- Boolean
- pointer

Aggregate types are used to define data constructs representing collections of data values which can be treated as a single named object.

The aggregate types are:

- sets
- structures
- areas
- tables
- descriptors

2.1.1 Numeric Range

A numeric range type defines a range of numeric values within which instances of the type may assume. The range is used to determine the amount of storage required by an instance of the type.

EXAMPLES

```
TYPE positive_integer {0 TO 32767};
    % Defines the type positive_integer as
    % a number in the range 0 to 32767.

TYPE integer {-32768 TO 32767};
    % Defines integer as a number in the
    % range -32768 to 32767. Note that
    % integer is not a built-in type.

TYPE digit {0 TO 9};
    % Defines the type digit as a number in
    % the range 0 to 9.
```

2.1.2 Symbolic Range

A symbolic range type specifies by enumeration the allowable constant values that may be assumed by an instance of that type. That is, the value-set of a symbolic range type is "enumerated".

EXAMPLES

```
TYPE digit_protocol  {mf, dp, dt};
    % Specifies that the type digit_protocol
    % consists of the values mf, dp, or dt.

TYPE weekday {mon, tues, wed, thurs, fri} ;
    % Specifies that the type weekday is comprised
    % of the values mon, tues, wed, thurs, or fri.
```

2.1.3 Boolean

The value-set of type Boolean contains two values: TRUE and FALSE. These correspond to truth values, and are the result of evaluating any Boolean expression.

EXAMPLE

```
 TYPE can_be_serviced BOOL;
    % Specifies that a variable of type
    % can_be_serviced may be either TRUE or FALSE.
```

2.1.4 Pointer

An item of type pointer contains an absolute address and is used to indirectly reference data.

EXAMPLES

```
TYPE pntr PTR TO digit_protocol;
    % Defines pntr as a pointer to instances
    % of type digit_protocol.

TYPE ptr_day PTR TO weekday;
    % Defines ptr_day as a pointer to instances
    % of type weekday.
```

Pointers to storage references may be obtained by using the PTR operator as described in "Unary Operators". To access the storage reference pointed to by a pointer, one dereferences the pointer. Dereferencing of pointers is described in "Dereference".

2.1.5 Sets

A set is used to define a data type whose value-set consists of any subset of the values from a specified symbolic or numeric range. Within an instance of a set, one bit is associated with each element of the base symbolic or numeric range.

EXAMPLES

```
TYPE bit_word SET {0 TO 15};
    % Specifies that an instance of the type
    % bit_word may be used to represent any
    %
    %         16
    % of the 2   (two to the sixteen) subsets of
    % the set {0,1,2,...,15}.
    % Each of the 16 bits in the set is associated
    % with one of the values in the numeric range.

TYPE special_features SET {abbreviated_dial,
                           add_on,
                           call_transfer,
                           digitone};
    % Specifies that an instance of the type
    % special_features can assume any of the
    %
    %  4
    % 2  (two to the four) subsets of the
    % set {abbreviated_dial, add_on, call_transfer,
    % digitone}.

TYPE feature {abbreviated_dial,
              add_on,
              call_transfer,
              digitone};

 TYPE special_features SET feature;
    % Same as example 2. An instance of type
    % special_features can assume any subset
    % of the set of feature.
```

DISCUSSION

The type special_features has a storage requirement of 4 bits. Each bit is associated with an element in the symbolic range.
For example, in any subset of type feature, the element call_transfer is either in the subset or it is not. We represent these two conditions by setting the bit corresponding to call_transfer to one or zero, respectively.

The maximum number of elements in a set is the number of bits in a word (16 in PROTEL/DMS, 32 in PROTEL/370).

2.1.6 Tables

A table type is used to build a named collection of homogeneous items, i.e., elements of the same type.

EXAMPLES

1.

```
TYPE digit_register TABLE[0 TO 15] OF digit;

                                |          |
                                |          |
                            bounds      type of entries
```

2.

```
TYPE boc_type {boc1, boc2, boc3, boc4};

TYPE switch_count TABLE[boc_type] OF integer;
```

The type declaration in example 1 specifies that digit_register is a table of 16 elements, the elements of which are of type digit. An index range specifies the legitimate set of indices into the table. Informally, we think of the index range as specifying the "bounds" of the table. In example 1, the "lower bound" of a digit_register table is 0, and the "upper bound" is 15.

The bounds specification of a table can be a numeric or symbolic range. It may also be a defined type, if the base type of the defined type is either numeric or symbolic range. In example 2, a table of type switch_count has 4 elements, one for each value in its index type (boc_type).

In all cases, the number of elements in the table is equal to the number of elements in the index range.

The type of entries in a table may be any type, including table types. Thus, a two dimensional table may be declared as a TABLE OF TABLE.

An item within a table is referenced by suffixing the name of the table instance with a subscript expression in square brackets. The value (as described in "Indexing") of the subscript expression corresponds to the desired table element.

2.1.7 Structures

A structured type is used to build a named collection of non-homogeneous elements, i.e., elements of different types.

Examples

1.

```
TYPE terminal_id
    STRUCT
       terminal_controller {0 TO 4095},
       terminal_number {0 TO 255}
    ENDSTRUCT;

    % Specifies that the type terminal_id is made
    % up of two fields, a terminal_controller in
    % the numeric range 0 to 4095 and a terminal_number
    % in the numeric range 0 to 255.
```

2.

```
TYPE time_interval
    STRUCT
       value {0 TO 255},
       units {tenms, secs, mins, hrs}
    ENDSTRUCT;

    % Type time_interval has two fields,
    % value in the numeric range 0 to 255,
    % and units in the symbolic range
    % {tenms, secs, mins, hrs}.

DCL call_time time_interval;
    % An instance of time_interval.
```

3.

```
TYPE experiment_results
    STRUCT
       runs integer,
       time time_interval
    ENDSTRUCT;

 DCL
    exp1 experiment_results,
    exps TABLE[1 TO 10] OF experiment_results;
```

Each element of a structure has a name, e.g., "value" and "units" in type time_interval (example 2). Each element of a structure is called a field of the structure. "value" is a field of type time_interval. A field specification consists of a name and a type. To reference the field value of an instance of type time_interval, say call_time, we use the name of the instance and the name of the field, joined by a period.

EXAMPLES

```
call_time.value
call_time.units
```

Fields within a structure may be of any type, including structure types. Thus, structures may be nested. In example 2 above, a variable of type experiment_results has a field "time", which is a structure of type time_interval.

EXAMPLES

```
exp1.runs
exp1.time.value
exp1.time.units
```

A field within a structure is referenced by specifying the name of the structure instance and the name of the field itself, as described in "Selection".

2.1.8 Areas

The area type is used to build collections of items similar to those built by the structure type. Areas, however, provide a facility to postpone declaration of fields in the structure to a later time.

An area type is either:

- a father area. This is an initial type declaration for an area; or
- a refinement of a father area.

A father area declaration indicates the amount of storage required for the area and may specify some of the fields in the area. A refinement specifies the names and types of one or more fields in the area.

Several refinements may be associated with the same father area. In such cases, the refinements are overlayed in storage.

EXAMPLES

1.

```
TYPE father AREA(3 * 16)
   i integer
ENDAREA;

% Specifies that father is an area 48 bits in width.
% father's first field is i, an integer. There are
% 32 bits unallocated in the type declaration for
% father.
```

2.

```
TYPE son1 AREA REFINES father
    t TABLE[0 TO 15] OF BOOL
ENDAREA;

% Specifies that son1 is a refinement of father. The
% table t is allocated storage, after i, in the space
% reserved for father.
```

3.

```
TYPE son2 AREA REFINES father
   j int,
   k colour
ENDAREA;

% Specifies that son2 is a refinement of father. The
% integer j and k of type colour are both allocated
% storage, after i, in the space reserved for father.
% Note that j and t share the same storage.
```

Area types exist to allow for information hiding and data abstraction. These concepts are discussed in "Using PROTEL in DMS" in the section on AREAS (DIS OFLPA).

2.1.9 Descriptors

Descriptors are used to indirectly reference tables. Conceptually, a descriptor consists of two parts: A descriptor part and a table part. The descriptor part contains the number of elements in the table, the size of these elements and a pointer to the base of the table. The table part contains the elements of the table.

At run time, the information in the descriptor part is modified to reflect changes in the table pointed to.

EXAMPLE

```
TYPE features_table DESC OF integer;
   % Defines features_table as a descriptor which
   % points to a table whose elements are integers.
```

An item within a descriptor is referenced in the same manner as an item within a table. The lower bound of a descriptor is assumed to be zero.

The TDSIZE and UPB descriptor operators are described in "Unary Operators".

There are two primary uses of descriptors:

1. To allow the implementation of generalized code which can deal with tables of arbitrary size.

2. To allow dynamic allocation of tables. A descriptor can be "filled in" by some storage allocator at run-time; conversely, storage pointed to by a descriptor can be reclaimed dynamically, at which point the descriptor would be set to NIL, thus disconnecting it from the storage it previously pointed to. In PROTEL all local descriptors are initialized to NIL.

2.1.10 Procedures

The procedure type is used to define the formal parameters and return type for a procedure.

EXAMPLE 1

```
TYPE get_channel_proc PROC(terminal terminal_id)
    RETURNS integer;

DCL new_get_channel get_channel_proc;
```

The name of the type is get_channel_proc. An instance of get_channel_proc is a procedure; new_get_channel is an instance of get_channel_proc. It has one parameter, terminal, of type terminal_id. It returns a value of type integer. The RETURNS clause is omitted for procedures which do not return a value.

The parameter is passed by value for parameters defined as terminal is in example 1. This means that the value of the actual parameter will be copied into the formal parameter on procedure entry. A value parameter may be altered within the procedure, but doing so will not change the value of the corresponding argument in the calling routine.

The REF keyword may be associated with a parameter. When REF is specified, the address of the actual parameter will be passed to the called procedure. This can be used, for example, when the parameter is a large aggregate which should not be copied. REF parameters are read-only.

Alternatively, keyword UPDATES may be associated with a parameter. In this case the parameter is passed in the same way as a REF parameter but updating is allowed and will cause the actual parameter in the calling procedure to change.

EXAMPLE 2

```
TYPE data_table TABLE[1 TO 50] OF BOOL;

TYPE transfer_data PROC(REF     features_table_1 data_table,
                        UPDATES features_table_2 data_table);
```

The name of the procedure type in example 2 is transfer_data. It has two parameters, features_table_1 and features_table_2. Each is a table containing fifty entries of type BOOL. The addresses of both parameters are passed to the called procedure. Updating of features_table_2 is allowed; features_table_1 cannot be altered.

A detailed discussion of parameter passing mechanisms is given in "Parameter Passing Mechanisms".

2.1.11 Pack Attribute

The width in bits of a data item is determined from the item's type. In the definition of a type, the PACK attribute can be used to specify a different width. It can be used, for example, to align fields on byte boundaries.

EXAMPLE 1

```
TYPE card_number
       STRUCT
          card_in_shelf PACK(8)  {0 TO 15},
          shelf_number  PACK(16) {0 TO 4095}
       ENDSTRUCT;
```

Normally, the field card_in_shelf would be allocated only 4 bits. The PACK(8) specification overrides this, and forces 8 bits to be allocated. Similarly the field shelf_number requires only 12 bits. However, 16 bits are allocated, since PACK(16) is specified.

A PACK attribute can specify a bit width greater than or equal to the default width for a type; it cannot force a type to occupy less space than it would otherwise. Its effect, therefore, is to pad the width normally allocated for a type.

Scalar types have maximum allowed packing widths. Aggregate types can be packed to arbitrary widths.

EXAMPLE 2

```
TYPE line_table TABLE[0 TO 99] OF PACK(8) BOOL;
```

An instance of type BOOL only requires 1 bit. However, since PACK(8) is specified, 8 bits are allocated for each element of line_table.

2.2 Data Declarations

EXAMPLES

```
DCL remote_make_busy BOOL;

DCL inter_digit_time  time_interval;

DCL calculate_checksum PROC(start_address store_address)
                       RETURNS integer IS
        BLOCK
          :
          :
          :
        ENDBLOCK;
```

Data declarations are used to define the data accessible to a program. They can be divided into two categories:

1. Variable declarations;
2. Constant declarations.

2.2.1 Variable Declarations

Figure 2 gives examples of data declaration statements being used to declare variables.

The DCL statement declares one or more variables and their type. The type used in a variable declaration can be a defined type created with a type definition or a primitive type. For example, the variable "result" is defined in terms of the type integer.

integer is a defined type. In the first three declarations the variables are defined in terms of the primitive types, numeric range, Boolean, and symbolic range.

The value following the keyword INIT specifies the initial value to be assumed by a variable at the time of its allocation to storage. In the last example the variable result is initialized to i. The INIT clause is optional.[1]

[1] In the DMS implementation, variables declared within a procedure can be initialized using an INIT clause. Global variables cannot generally (there are exceptions) be initialized in this fashion. The IBM/370 implementation allows global variables to be initialized as well.

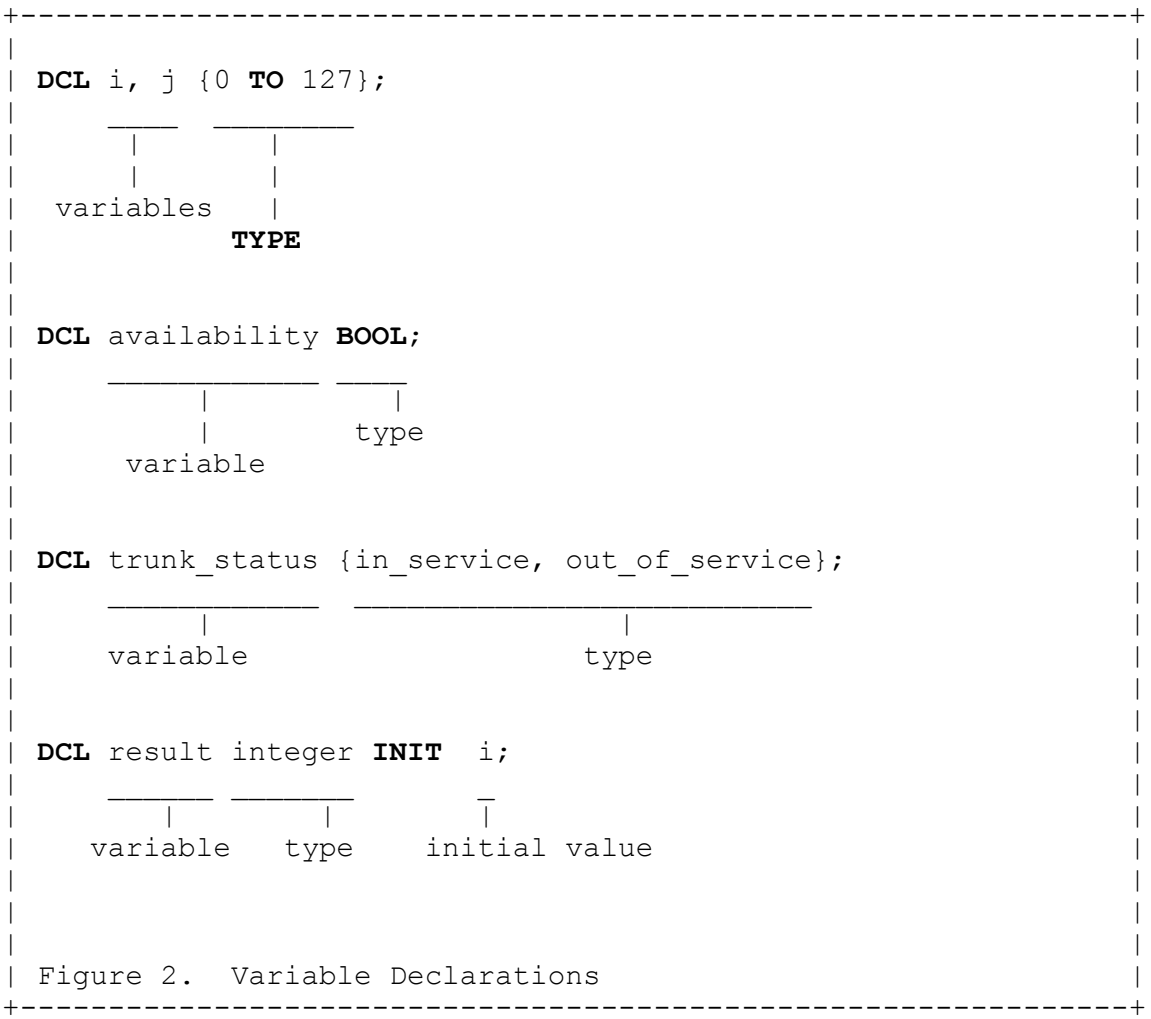

Figure 2. Variable Declarations

2.2.2 Constant Declarations

The data declaration statement can be used to define symbolic names for scalar, aggregate, or procedure constants. The value of a declared constant is fixed at compile time and may not be modified during program execution.

The value(s) following the keyword IS in the declaration specifies the constant value(s).

In Figure 3 the value of the constant seconds_per_hour is fixed at 60 * 60, and the primes table contains the entries 2, 3, 5, 7, 11, and 13.

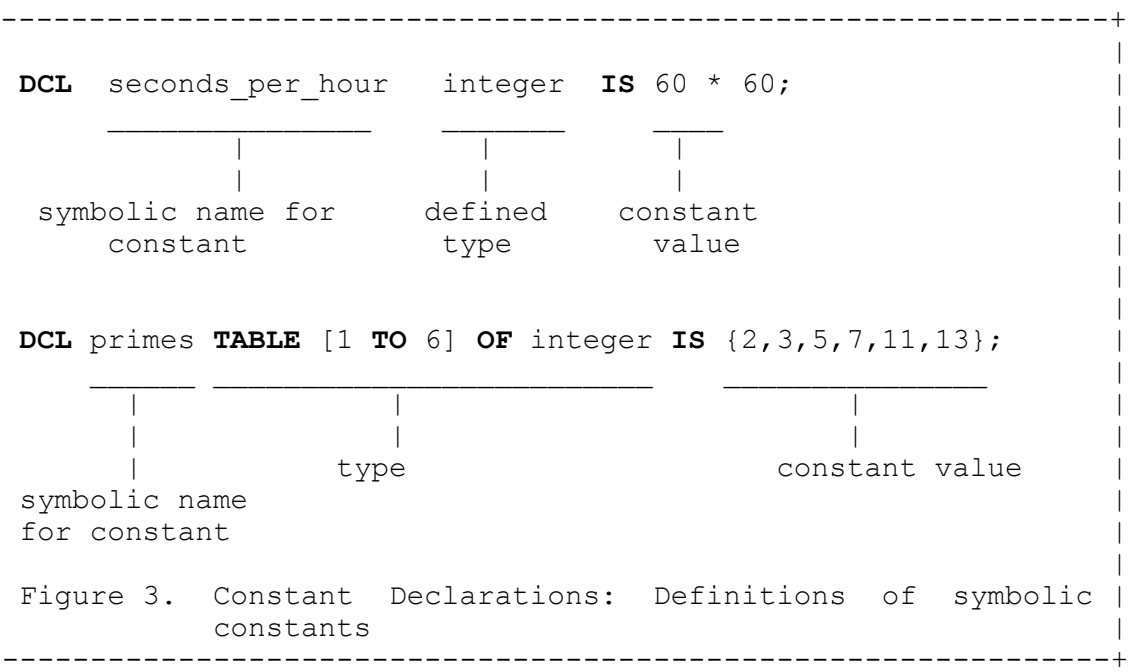

Figure 3. Constant Declarations: Definitions of symbolic constants

Constant declarations can also be used to define the type and body of a procedure. As an example, consider Figure 4. Statements 1 through 8 comprise the constant declaration statement. The name of the procedure, the parameters, and the return value are specified in statements 1 through 2. Calculate_average is the procedure name. This is followed by the keyword PROC and the procedure parameters and their types enclosed in parentheses. In the example the parameters are number_1 and number_2, each of type integer.

The portion of the statement

```
RETURNS integer
```

is called the "returns clause". It specifies that the procedure returns a value of type integer.

The block statement following IS and delimited by the BLOCK and ENDBLOCK keywords defines the body of the procedure calculate_average. We call the DCL statement, up to the IS keyword, the declaration of procedure "calculate_average". We call the body of a procedure its definition, or its implementation.

Data declarations local to the procedure may follow the keyword BLOCK. In this example, the local variable "average" is declared in statement 4.

Executable statements (statements 6 and 7) related to the procedure are located between the local variable data declaration(s) and the keyword ENDBLOCK.

```
 1  DCL calculate_average PROC(number_1, number_2 integer)
 2                                          RETURNS integer IS
 3  BLOCK
 4     DCL average integer;
 5
 6     (number_1 + number_2) / 2 -> average;
 7     RETURN average;
 8  ENDBLOCK;
```

Figure 4. Constant Procedure Declaration

3.0 Executable Statements

Executable statements operate on the data defined in the declaration statements. We think of executable statements as "doing things" or manipulating objects. Formally, executable statements are those PROTEL statements which cause sequences of machine language instructions to be generated.

Executable statements can be divided into two basic categories:

1. Expressions;
2. Control Statements.

3.1 Expressions

EXAMPLES

```
i * result -> result;

get_idle_trunk();
```

Expressions are executable statements which create new data values from existing ones. Expressions can be categorized by those which evaluate to a value, and those which do not. An expression which does not evaluate to a value is a procedure call. An expression is a statement if it is either an assignment statement, or a procedure call.

The assignment operator in PROTEL is "->".[2]

The value of the expression

```
23 -> x
```

is 23.

To see that an assignment statement is an expression, one must understand that the assignment operator is a binary operator which causes a side-effect.

Consider the following assignment statement:

```
23 + 1 -> x / 2 -> y;
```

[2] '->' is pronounced "guzinta", a derivative of "goes into". Thus a -> b reads "a goes into b".

Evaluation of arithmetic operators in PROTEL is left-to-right. The value of "23 + 1 -> x" is 24, with the side-effect that x has been assigned the value 24. 24 / 2 is then evaluated and the right-most assignment operator is evaluated, causing y to be assigned the value 12. The order of evaluation of operators in this example is +, ->, /, ->.

Assignment expressions contain the assignment operator in the expression. However, an assignment expression is a statement only if the assignment operator is the last evaluated operator in the expression.

EXAMPLES

```
i + 1 -> i;       % legal assignment statement.

(i + 1 -> i);     % legal assignment statement.

(i -> i + 1);     % not legal; -> not the last operator.

p(i + 1 -> i);    % legal if p does not return a value.
```

Procedure calls may be used either by themselves or as a component of an assignment statement. When used as part of an assignment statement, a procedure must have been declared with a RETURNS clause, indicating the type of the value to be returned. When used by itself the procedure must have been defined without the RETURNS clause. A procedure call consists of the procedure name followed by the procedure parameters enclosed in parentheses.

In the first example above, the value of i is multiplied by the value of result and the new value assigned to result.

In the second example above, the procedure get_idle_trunk is called. Note that this procedure has no parameters as is indicated by the empty parentheses.

The basic components of an expression are operators, operands, and parentheses.

The operands and operators are indicated in the statement in Figure 5.

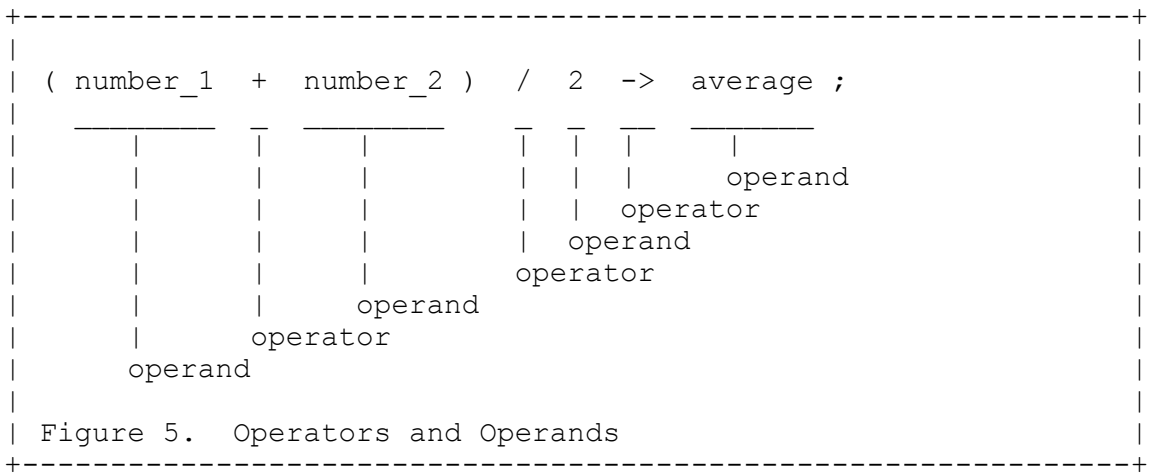

Figure 5. Operators and Operands

3.1.1 Operators

The types of operators used in PROTEL are:

- assignment operator
- arithmetic operators
- comparison operators
- logical operators
- unary operators

3.1.1.1 Assignment Operator

The assignment operator -> causes the value on the left of the operator to be assigned to the variable on the right. The operands and the result of an assignment statement must be type-compatible. The type compatibility rules are discussed in detail in the PROTEL Reference Manual.

3.1.1.2 Arithmetic Operators

The PROTEL arithmetic operators are:

*	multiplication
/	integer division
+	addition
-	subtraction
MOD	a MOD b gives the integer remainder of a divided by b
<<	shift left
>>	shift right

Arithmetic operators operate on and produce values of type numeric range.

3.1.1.3 Comparison Operators

The PROTEL comparison operators are:

```
a < b     % a is less than b

a > b     % a is greater than b

a <= b    % a is less than or equal to b

a >= b    % a is greater than or equal to b

a = b     % a is equal to b

a ^= b    % a is not equal to b
```

These operators compare (algebraically) the relative magnitudes of their two operands and yield a Boolean (true or false) value as a result. The valid operand types for the <, >, <=, and >= operators are the types numeric range and symbolic range. For the = and ^= operators, any type of operand is valid. The result of these operators is of type BOOL.

Tables or structures of identical type may also be compared. This is not a good idea with structures, because there may be unused bits with random values inside a structure. The detailed rules for allowable comparisons are given in the PROTEL Reference Manual.

3.1.1.4 Logical Operators

The logical operators used in PROTEL are:

Symbol	Meaning
&	AND
|	OR
!	Exclusive OR
^	NOT
INCL	Set inclusion ("is included in")
NOTINCL	Set exclusion ("is not included in")

The valid operand types and result types for the &, |, and ! operators are BOOL, numeric range, and SET.

The function of the logical operators depends on their operands. When both operands are of type BOOL, they perform the logical operations. When both operands are of type SET or numeric range, they perform the bitwise AND, OR, and exclusive OR functions (set intersection, set union, and exclusive OR). For these cases, the result is of the same type as the operands.

For example, the set s is defined in the statement

```
DCL s SET {x, y, z};
```

The OR operator can be used to set the x bit in the set s.

```
{x} | s -> s;
```

Similarly, the AND operator can be used to clear the x bit in the set s.

```
^ {x} & s -> s;
```

The operand type for the INCL and NOTINCL operators is the type SET. The result is of type BOOL.

```
{y,x} INCL {x,z,w,y}   % TRUE. {y,x} is included in {x,z,w,y}.
```

Compound Boolean expressions can be formed from one or more Boolean expression through the use of logical operators. Consider, for example, the Boolean expressions:

```
a >= 2

a = 1

b <= 3
```

A compound Boolean expression composed from these three expressions is shown in Figure 6.

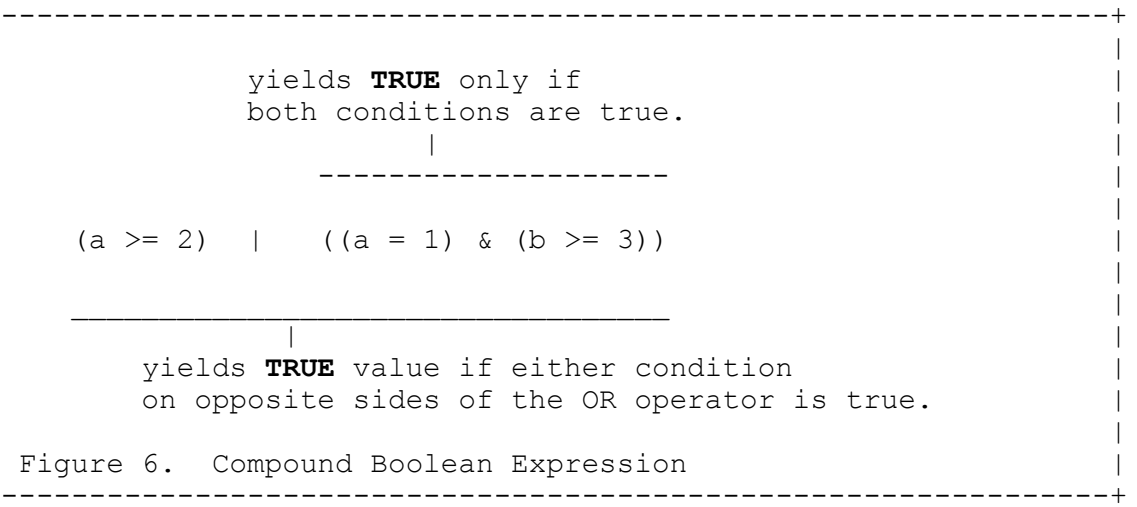

Figure 6. Compound Boolean Expression

3.1.1.5 Unary Operators

The unary operators used in PROTEL are:

Symbol	Meaning
-	negation (arithmetic)
^	NOT (logical)
UPB	upper bound
TDSIZE	Size ("Table or Descriptor SIZE")
DESC	Descriptor
PTR	Pointer

The valid operand and result type for the minus operator ('-') is numeric range.

For the NOT operator ('^'), the operand type must be one of BOOL, numeric range, or SET. The result is of the same type as the operand.

The operand type for the UPB and TDSIZE operators must be either DESC or TABLE. UPB returns the highest value in the bounds of the table; TDSIZE returns the number of elements in the table. The result type for the UPB operator is the type of the bounds; for TDSIZE, it is numeric range.[3]

The DESC operator requires a table or descriptor as an operand and returns a descriptor to that table. The PTR operator accepts a storage reference as an operand and returns a pointer to the storage location where the reference is contained.

Expressions are evaluated from left to right. However, parentheses can be used to change the order of evaluation. For example,

```
sum_1 + sum_2 / sum_3
```

sum_1 is added to sum_2 and the result is divided by sum_3. However, in the expression

```
sum_1 + (sum_2 / sum_3)
```

sum_1 is added to the result of dividing sum_2 by sum_3.

EXAMPLES

```
DCL t TABLE[0 TO 10] OF integer;

UPB t          % value is 10.

TDSIZE t       % value is 11.
```

[3] Actually, TDSIZE yields a result of type unsignedint, a positive subset of numeric range.

3.1.2 Operands

The operands in an expression are the objects operated on by the operators.

Operands in an expression can be of three types:

1. constants
2. storage references
3. procedure calls

Examples of each of these operand types are illustrated in Figure 7.

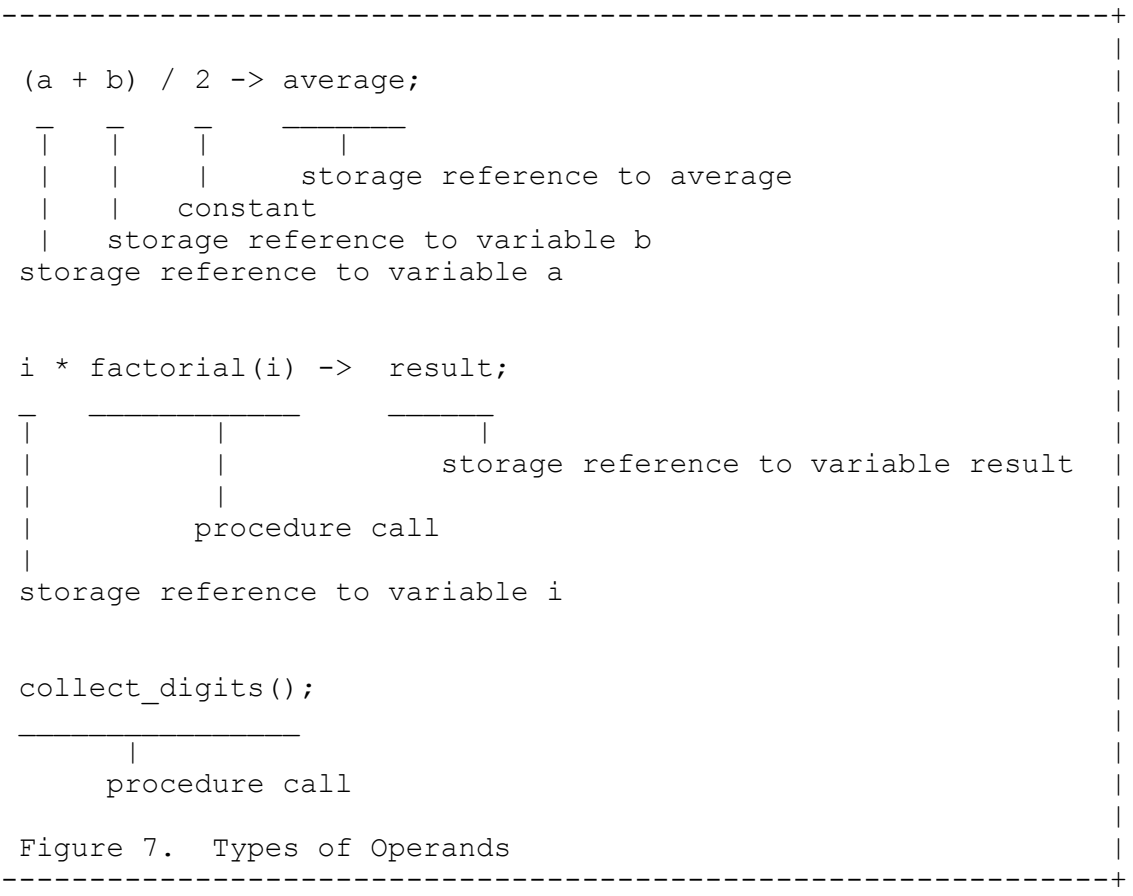

Figure 7. Types of Operands

"Constants" may be either self-defining or user-defined. A self-defining constant is a number, tuple or string occurring in an expression. A user-defined constant is an identifier denoting a constant declared in a constant declaration (i.e., a DCL statement).

Storage references may be of five types:

1. Identifiers
2. Selection
3. Indexing
4. Multiples
5. Dereferences

3.1.2.1 Identifiers

The "identifier" form of a storage reference may be applied when dealing with variables of a simple type or all of the fields of a structured type. The effect of such a reference is to yield the value of the variable associated with the identifier.

3.1.2.2 Selection

Selection is used to specify a field of a structure or area. A "selection" consists of a storage reference to the structure followed by a dot, followed by the name of the selected field.

For example, consider the data declaration statement

```
DCL trunk_id  terminal_id;
```

It specifies that the variable trunk_id is of the type terminal_id (as defined in example 1 in “Structures”.)

The field terminal_number of the structure is accessed in the following way:

```
trunk_id . terminal_number
--------   ---------------
     |              |
     |              |
     |            field referenced
     |
   an instance of type terminal_id
```

3.1.2.3 Indexing

Indexing is used to select an element of a table or descriptor. An "indexing" consists of a storage reference to the table or descriptor followed by a square bracketed expression. The expression should be of the same type as the table bounds and its result indicates the position of the desired element within the table.

For example, a reference to the 4th value in an instance
digit_register_x of type digit_register would be written

```
digit_register_x[3]
```

(Remember that digit_register is indexed from zero.)

3.1.2.4 Multiple

The multiple form of a storage reference is used to index more than one element of a table, or even a whole table, simultaneously. A "multiple" consists of a storage reference to the table followed by a square bracketed subrange expression. The subrange expression indicates, by specifying its first and last elements, the portion of the table to be referenced.

The subrange expression is optional. If omitted, the whole table is referenced.

EXAMPLES

```
digit_register_x[0 TO 4]
   % specifies the first 5 elements of digit_register_x.

digit_register_x[11 TO 15]
   % specifies the last 5 elements of digit_register_x.

digit_register_x[]
   % specifies the whole table digit_register_x.

digit_register_x[0 TO UPB digit_register_x]
   % specifies the whole table digit_register_x.
```

Note that brackets are required when specifying all elements in a table.

Forming a descriptor over only part of a table is sometimes referred to as taking a "slice" of the table.

3.1.2.5 Dereference

Dereferences denote items pointed to by pointer variables. A "dereference" consists of a storage reference to a pointer followed by an at ("@") symbol.

For example, consider the following declarations

```
DCL p PTR TO {0 TO 255};

DCL i, j {0 TO 255};
```

The following two-statement sequence causes the value of i to be assigned to j

```
PTR i -> p;
p@    -> j;
```

3.2 Control Statements

EXAMPLES

```
RETURN result;


IF a > 0 THEN
  b + 1 -> b;
ENDIF;


WHILE (i - 1 -> i) > 1
DO
  i * result -> result;
ENDDO;
```

These examples illustrate three types of control statements:

1. the RETURN statement;
2. the IF statement;
3. the ITERATIVE statement.

Control statements are used to modify the normal sequential execution of statements. PROTEL has different types of control statements. These are:

- RETURN statement;
- IF statement;
- CASE statement;
- SELECT statement;
- ITERATIVE statement;
- EXIT statement.

Each of these will be discussed in detail in the following sections.

3.2.1 RETURN Statement

```
RETURN result;
```

The RETURN statement, when used in a procedure, causes control to be returned to the point of invocation of the procedure.

Procedures defined with a RETURNS clause must specify after the keyword RETURN the expression whose value is to be returned.

Consider the statements in Figure 8.

```
DCL factorial PROC(i integer)  RETURNS integer IS
                               ---------------
                                       |
                                  RETURNS clause
                                  specifies the type
                                  of the value returned

BLOCK
   DCL result integer INIT i;
      :
      :
      :
      :
   RETURN result;
          ------
           |
           expression to be returned
ENDBLOCK;
```

Figure 8. Return Statement: A PROTEL procedure may return a value of any type.

Note that result is of the type integer as defined in the procedure declaration.

3.2.2 IF Statement

The IF statement is used in PROTEL when a test or a decision is to be made between alternatives. The IF statement has the following format:

```
IF expression_1
   THEN
      statement_list_1;
   ELSE
      statement_list_2;
ENDIF;
Next_Statement;
```

The expression following the keyword IF must always yield a value of type BOOL. If this value is TRUE, the statement list following THEN is executed. If the value is FALSE, the statement list following ELSE is executed. The ELSE clause of the IF statement is optional. If it is not present and the value of the expression following IF is FALSE the statement following ENDIF is executed.

Both statement_list_1 and statement_list_2 are optional.

EXAMPLE

```
IF x > y
   THEN
      x - y -> x;
   ELSE
      y - x -> x;
ENDIF;
Next_ statement;
```

In example 1 above one of the expressions

x - y -> x

or

y - x -> x

will be executed. The choice depends on the values of x and y. If x is greater than y, x - y is assigned to x. If, however, x is not greater than y, then y - x assigned to x.

3.2.3 Nested IF Statements

The THEN and ELSE clauses of the IF statement can be expanded to govern the execution of any number of alternative statements. This type of statement contains IF statements within IF statements and is referred to as a nested IF statement.

The general form of such statements is shown in Figure 9. Note that each IF has a corresponding ENDIF.

```
IF expression_1
   THEN
      IF expression_2
         THEN
            statement_list_1;
         ELSE
            statement_list_2 ;
      ENDIF;

   ELSE
      IF expression_3
         THEN
            statement_list_3 ;
         ELSE
            statement_list_4 ;
      ENDIF;
IF;
Next Statement;
```

Figure 9. IF Statement: general form.

In Figure 10, only one of the three statements

```
x -> biggest_number;

y -> biggest_number;

z -> biggest_number;
```

will be executed. Statement 12 logically follows statements 3, 7, and 9.

If x is greater than both y and z then statement 3 is executed. If, however, x is less than or equal to either y or z, control is transferred to statement 5. Now if y is greater than z, then statement 7 is executed. If, however, y is not greater than z, then statement 9 is executed.

```
 1   IF (x > y) & (x > z)
 2      THEN
 3         x -> biggest_number;
 4      ELSE
 5         IF y > z
 6            THEN
 7               y -> biggest_number;
 8            ELSE
 9               z -> biggest_number;
10         ENDIF;
11   ENDIF;
12   next_statement_1;
```

Figure 10. IF Statement Example

3.2.4 CASE Statement

The CASE statement is another type of control statement. It is used to selectively transfer control to one group of statements out of several.

The general form of the CASE statement is:

```
CASE expression IN
   {expression_value_1}:  statement_list_1;
   {expression_value_2}:  statement_list_2;
        .
        .
        .
   {expression_value_n}:  statement_list_n;
   OUT statement_list_n_plus_1;
ENDCASE;
```

Each group of statements is uniquely labelled with one or more of the values in the range of the expression following the keyword CASE. The value of expression must be of type numeric range, symbolic range, SET, or BOOL. When the CASE statement is executed control is transferred to the group of statements whose label matches the value of expression. If no match is found, control is transferred to the group of statements following OUT.

When the last statement in a group has been executed control passes to the statement following the keyword ENDCASE, providing that no statement in the group transfers control out of the CASE statement (e.g., RETURN or EXIT).

```
TYPE digit {0 TO 9};

DCL n digit;

CASE n IN  % n is first digit dialled
   {0} : TRUE -> operator;

   {1} : TRUE -> ddd_call;
         set_area();

   {4} : TRUE -> assist;
         check_11();

 {2,6} : TRUE -> wrong_number;   % sorry, ...

   OUT   TRUE -> local_call;
ENDCASE;

Next_statement;
```

Figure 11. Case Statement

In Figure 11 n lies in the range 0 to 9. Values in this range are used to label statements enclosed by CASE, ENDCASE keywords.

Note that not all possible values for n (0 to 9) occur as case selector labels.

When the CASE statement is executed, control is transferred to the group of statements whose label matches or includes the value of the expression. For example, if n is 0, control is transferred to the statement

```
{0} : TRUE -> operator;
```

If n does not equal 0, 1, 2, 4 or 6, control is transferred to the statement following OUT. In all cases, execution will resume at next_statement.

3.2.5 SELECT Statement

The SELECT statement is similar to the CASE statement. Their syntax is identical except that SELECT and ENDSELECT are substituted for CASE and ENDCASE.

The action of CASE and SELECT are the same. The difference is in efficiency. CASE is usually faster than SELECT but may require more space. This is particularly true if the range of the label values is large but the actual labels are sparse within that range.

3.2.6 Iterative Statements

Another type of control statement is the "iterative" statement.

```
FOR i FROM initial UP TO limit BY modification
DO
   statement_1;
      :
      :
      :
   statement_n;
ENDDO;
```

The iterative statement facilitates the definition and control of repetitions of a group of statements.

The statements which perform these repetitions are collectively called an iterative statement or loop.

The FOR statement defines the number of times the iterative statement will be executed.

The components of this statement are:

- the control variable. It is implicitly declared by the FOR statement and is not accessible outside the loop. It is read only.
- the initial value. It is optional; if omitted, FROM 0 is assumed.
- the limit value. It is optional; if omitted, DOWN TO 0 is assumed.
- the modification value. It is optional; if omitted, BY 1 is assumed.

They are labelled in the statement in Figure 12.

```
FOR i FROM 0 UP TO 10 BY 1
     _       _        __     _
     |       |        |      |
     |       |        |      modification
     |       |        limit    value
     |       |        value
     |       initial
     |       value
     |
     control
     variable
```

Figure 12. FOR Loop

The sequence of events involved in executing the iterative statement is:

1. Initialize the control variable.

This variable (i in our example) will be set equal to the initial value (0 in our example). It cannot be modified within the loop.

2. Test the control variable.
   If the control variable is less than or equal to the limit value, the statements following DO will be executed. Otherwise a transfer to the statement following ENDDO is effected and the iteration terminated. In the example, as long as I is less than or equal to 10 the statements following DO will be executed.

3. Execute the statements following DO.

   The statements headed by the DO and terminated by the ENDDO are executed.

4. Modify the control variable.

   The modification value is added to the control variable. In the example above the constant 1 is added to the contents of the control variable i.

5. Return to step 2.

   Consider the following example

```
FOR i FROM 2 UP TO 10 BY 3
   DO
      :
      :
      :
   ENDDO;
```

The first time through the loop, i will have value 2. The second time through i will have value 5. Recall that the modification value, in this case 3, is added to the control variable (2 + 3 = 5). The third time through the loop, i will have value 8. On the fourth test of "i <= 10", i will have value 11, the test will fail, and the body of the loop will not be executed.

3.2.7 The OVER Specification

Another way of specifying the number of times the iteration is to be performed is by using the OVER keyword.

EXAMPLE 1

```
TYPE weekday {mon, tues, wed, thurs, fri};

FOR i OVER weekday
  DO
      :
      :
      :
  ENDDO;
```

The number of elements in the symbolic range weekday is 5. Therefore, the above FOR statement performs the same number of iterations as

EXAMPLE 2

```
FOR j FROM 0 UP TO 4
  DO
      :
      :
      :
  ENDDO;
```

The two loops differ in the type of the control variable. i has type weekday, while j has type "numeric range".

In example 2 the control variable j is initialised to 0 and the loop is repeated until j is greater than 4. Each time the value of j is incremented by 1.

In example 1 the control variable i is initialised to mon, and increments through the values tues, wed, thurs, and fri.

The program in Figure 13 sums the elements of a table.

```
 1     DCL t TABLE [0 TO 9] OF integer;
 2
 3     DCL sum_t PROC () RETURNS integer IS
 4     BLOCK
 5        DCL sum integer INIT 0;
 6
 7        FOR i FROM 0 UP TO UPB t
 8           DO
 9              t[i] + sum -> sum;
10           ENDDO;
11        RETURN sum;
```

```
| 12    ENDBLOCK;                                                |
|                                                                |
| Figure 13.  Iterative Statement:  This program sums the        |
|             elements in a table.                              |
+----------------------------------------------------------------+
```

The table t, the elements of which are to be summed, has 10 entries all of type integer.

Note that no parameters are required for the sum_t PROC. This is indicated by the empty parenthesis in statement 3.

Statements 7 through 10 form the iterative statement, or loop. Note that in statement 7 the limit value is UPB t, meaning the upper bound of table t. Thus the loop is repeated until i is greater than 9. The preceding examples show loops with positive modification values. It is possible to specify a negatively incremented loop, in which case a 'count down' operation is effected. This is down by specifying DOWN TO instead of UP TO in the iteration statement.

Note that the modification value is still positive.

EXAMPLE 3

```
FOR k FROM 5 DOWN TO 0 BY 1
  DO
      :
      :
      :
  ENDDO;
```

Each time through the loop the value of k is decremented by 1.

3.2.8 Iterative Statement with the WHILE Condition

The WHILE condition is a modification of the iterative statement. This form of iterative statement specifies that the body of the loop is to be repeated as long as a specified 'condition' is true. The condition is evaluated prior to each pass through the loop.

```
1    0 -> a;
2    1 -> i;
3
4    WHILE (i <= 10)
5       DO
6          a + 1 -> a;
7          i + 1 -> i;
8       ENDDO;
```

Figure 14. WHILE Loop

In this case as long as i is less than or equal to 10 the loop is repeated.

The first time through the loop statement 7 assigns 2 (= i + 1) into i, since i was initialised to value 1 in statement 2.

On each iteration through the loop i is incremented by 1 until i = 11. At this point execution of the loop is terminated.

The program for calculating the factorial of a number in Figure 1 uses an iterative statement with the WHILE condition.

Both WHILE and FOR clauses may be mixed in the same statement.

EXAMPLE 1

```
FOR i FROM 0 UP TO 20 BY 1 WHILE (a < b) DO
   :
   :
   :
ENDDO;
```

A result of the default values for all clauses yields

EXAMPLE 2

```
FOR i FROM 0 DOWN TO 0 BY 1 WHILE TRUE DO
   :
   :
   :
ENDDO;
```

Example 2 describes an infinite loop. Therefore, with all optional fields removed, a loop-forever construct in PROTEL reduces to:

EXAMPLE 3

```
DO
    :
    :
    :
ENDDO;
```

3.2.9 EXIT Statement

The EXIT Statement is another type of control statement. It is used to branch out of a sequence of statements within a labelled BLOCK, DO, CASE, SELECT or IF statement.

The general format of the EXIT statement is:

```
e : DO
       statement_1;
       IF expression_1 THEN
             EXIT e;
       ENDIF;
       statement_2;
    ENDDO e;
    Next_statement;
```

Note in the example the DO statement is labelled e. The identifier e following the EXIT statement matches this label. If expression_1 is true then the EXIT is executed. This transfers control to the statement following the terminating keyword of the block whose label matches the identifier. In this case, control is transferred to the statement following ENDDO e;

If expression_1 is false, statement_2 is executed, and control returns to the statement labelled e.

4.0 Parameter Passing Mechanisms

4.1.1 Formal and Actual Parameters

A parameter declared in a procedure declaration is called a formal parameter. In Figure 15, f is a formal parameter of p.

The parameter in a procedure call which matches a formal parameter is called an actual parameter. On line 11 of Figure 15, i is an actual parameter for f. Actual parameters are referred to colloquially as "arguments".

```
    1 | DCL p PROC(f int) IS
    2 | BLOCK
    3 |    23 -> f;
    4 | ENDBLOCK;
    5 |
    6 | DCL alpha PROC () IS
    7 | BLOCK
    8 |    DCL i int;
    9 |
   10 |    75 -> i;
   11 |    p(i);
   12 |    typeint('i = ', i);
   13 | ENDBLOCK;
```

Figure 15. Parameter Passing: calling by value

There are numerous methods of matching actual and formal parameters. Only two are generally used in PROTEL. The first is called call-by-value. This involves evaluating the actual parameter, and passing its value to the formal parameter. The formal parameter is a stack variable which is initialised to the value of its actual parameter. It disappears when the procedure finishes executing. Inside of the procedure it is a read-write variable like all of the other variables local to that procedure. The only way it is distinguished is that it has an initial value supplied by the caller. Because the argument is a value, we consider the argument to be an expression, which is evaluated at run-time.

In Figure 15, the value of i at line 12 is 75. Note that procedure p cannot change the value of any argument to a call-by-value formal parameter. We may think of a call-by-value actual parameter as an initial value. Call-by-value is the default parameter passing mechanism in PROTEL.

The second parameter passing mechanism is called call-by-reference. Here a pointer to the argument is generated by the compiler as the actual parameter. The formal parameter is implemented as a pointer to the actual parameter. All accesses to the value of the formal parameter are indirect. The compiler must generate the indirect addressing modes required, even though the references inside the procedure look as though they are direct.

For example, if the formal parameter is a table t, then t[i] is an indirect reference through the table pointer t.

Note that for a call-by-reference parameter, the argument must be a storage reference.[4]

```
+-------------------------------------------------------------+
|                                                             |
|   1 | DCL h PROC(UPDATES t TABLE [1 TO 5] OF int) IS        |
|   2 | BLOCK                                                 |
|   3 |    78 -> t[3];                                       |
|   4 | ENDBLOCK;                                             |
|   5 |                                                       |
|   6 | DCL k PROC() IS                                       |
|   7 | BLOCK                                                 |
|   8 |    DCL w TABLE[1 TO 5] OF int;                        |
|   9 |                                                       |
|  10 |    {10,20,30,40,50} -> w[];                           |
|  11 |    h(w);                                              |
|  12 |    typeint('w[3] = ', w[3]);                          |
|  13 | ENDBLOCK;                                             |
|                                                             |
| Figure 16.  Parameter Passing:  call by reference (efficient) |
+-------------------------------------------------------------+
```

The value of t[3] in statement 12 is 78, not 30.

[4] A storage reference is generally a variable, but may also be a constant. A self-defining tuple, such as 'Quebec', is both a constant an a storage reference. If this is confusing, the reader is directed to the reference manual for an explanation of storage reference.

```
    1 | DCL g PROC(UPDATES x int) IS
    2 | BLOCK
    3 |    123 -> x;
    4 | ENDBLOCK;
    5 |
    6 | DCL k PROC() IS
    7 | BLOCK
    8 |    DCL v int;
    9 |
   10 |    67 -> v;
   11 |    g(v);
   12 |    typeint('v = ', v);
   13 | ENDBLOCK;
```

Figure 17. Parameter Passing: call by reference (inefficient)

Using procedure g in Figure 17, a call such as g(i+67) is illegal, since one cannot take a pointer to the value of 'i+67'. The argument to a call-by-reference formal parameter must be a storage reference. A storage reference is a data object which is stored in memory, and therefore has an address.

When a procedure intends to alter the value is an argument, the formal parameter must be call-by-reference. In PROTEL, we invoke call-by-reference by the keyword UPDATES. In line 12 of Figure 17, the value of v is 123, not 67.

Now we look at the 'work' involved in each of these parameter passing mechanisms. The number of bytes of data which must be copied to match an argument to a call-by-value parameter is equal to the width of the formal parameter. Hence, when f (Figure 15) gets an initial value, one word is passed. If the formal parameter is a table of 5 ints (such as f in Figure 16) then 5 words will be copied as the initial value.

A pointer, however, is always the same width. In PROTEL/DMS a pointer is two words. A call-by-reference parameter always requires two words to be passed. Obviously it is inefficient to pass an int or byte by reference. However, it is even less desirable to pass a large table by value. If we don't wish to have read-write access to an argument, we should use call-by-value.

For the sake of efficiency, PROTEL allows us to invoke call-by-reference through two different keywords. One, UPDATES, asks for the usual call-by-reference, i.e., is a demand for read-write capability. REF, on the other hand, says "this should be a call-by-value parameter, but because the bit width of the value is large, use call-by-reference with read-only capability.". Both REF and UPDATES generate identical object code, however a REF parameter cannot be used as the target of an assignment.

4.1.2 The VAL Attribute

The keyword VAL may also be used in formal parameter declarations. This is used when a level of indirection to data is passed, and the data itself is to be considered, but is not subject to modification by the procedure.

Consider parameter d3 of procedure pv in Figure 18. d3 is passed call-by-value, so pv gets a local copy of the argument. d3 is not a constant; pv is free to change the value of its local copy (though this would likely be unwise). However, pv is not permitted to modify the storage pointed to by d3. Hence line 7 is an error, since within procedure pv, the int type which d3 points to has been redeclared to be a read-only type.

Similarly p1 points to a read-only int type. p1 is not, itself, read-only, but again changing the value of p1, i.e., causing it to point somewhere other than its initial value would likely be a programming error, since no updated value can be returned through p1.

```
+-------------------------------------------------------------+
|                                                             |
|   1 |DCL pv PROC(d3 DESC OF VAL char, p1 PTR TO VAL char) IS  |
|   2 |BLOCK                                                  |
|   3 |   DCL                                                 |
|   4 |      c char,                                          |
|   5 |      capital DESC OF CHAR IS 'Ottawa';                |
|   6 |                                                       |
|   7 |   'G' -> d3[1];    % illegal. d3[1] is read-only.      |
|   8 |   'G' -> p1@;      % illegal. p1@ is read-only.        |
|   9 |   capital -> d3;   % legal. d3 is read-write.         |
|  10 |   PTR c   -> p1;   % legal. p1 is read-write.         |
|  11 |ENDBLOCK;                                              |
|                                                             |
| ERROR LINE      7                                           |
| *** ERROR: ATTEMPT TO ASSIGN VIA READ ONLY TYPE INT ON LINE 1 |
| ERROR LINE      8                                           |
| *** ERROR: ATTEMPT TO ASSIGN VIA READ ONLY TYPE INT ON LINE 1 |
|                                                             |
| Figure 18.  Parameter Passing:  use of VAL attribute        |
+-------------------------------------------------------------+
```

One last remark on procedure pv: being a two-word object, p1 should not be passed by REF for reasons of efficiency. It rarely makes sense to pass a pointer by passing a pointer to it. d3, on the other hand, is an object larger than two words, but since it contains a pointer it is also sensible to pass it by value.

Otherwise all references through d3 will require double dereferencing.

4.1.2.1 Common Errors

One sometimes finds formal parameter declarations such as

```
DCL px PROC(REF d4 DESC OF char) IS FORWARD;
```

where the author believes that REF protects the data pointed to by d4. It does not. d4 would, of course, have to be passed as an UPDATES parameter to allow it to be modified, so d4 itself is read-only. Inside the body of procedure px, the statement

```
'Canada' -> d4;
```

would be illegal; however, the statement

```
'Canada' -> d4[];
```

would be quite legitimate.

If px were called

```
:
px(some_desc);
:
```

then the caller can rest assured that some_desc has not been modfied, i.e., it still points to the same table, and that table still contains the same number of elements. There is no guarantee that the values of the elements have been unaltered.

The correct declaration to protect the table pointed to by d4 is:

```
DCL px PROC(d4 DESC OF VAL int) IS FORWARD;
```

5.0 PROTEL Program Organization

5.1 Introduction

No explanation of PROTEL program organization would be complete without a discussion of the design goals of the language - nor would it be easily understood. It also requires a firm understanding of the terms, declaration and definition, especially with respect to procedures.

5.1.1 Declaration Versus Definition

In general, a declaration declares the type of an object, while the definition gives the object a value. The declaration statement below declares variable I. The assignment statement is a definition of I. The last two statements both declare and define J and K, respectively. In every case, the declaration part of the statement ends after the type. The definition parts of the last two statements follow the keyword INIT or IS.

```
DCL I Int;
28 -> I;
DCL J Int INIT 29;
DCL K Int IS 30;
```

We say that a procedure is declared when it is first encountered in a DCL statement.

A declaration of a procedure defines the name of the procedure, its formal parameters (if any), and its return type (if any).

A definition specifies the "value" of the procedure, i.e., the executable statements which will be executed when the procedure is invoked. The value of a procedure is called its body.

EXAMPLES

1.
```
DCL P1 PROC() IS FORWARD;
```

2.
```
DCL Get_status PROC(P Port) RETURNS Status IS ...
```

3.
```
DCL Init_device PROC(D Device);
```

4.
```
TYPE Cc_partitioned_proc PROC() RETURNS BOOL;
DCL  Gated_use_partitioned_cc_translations
                            Cc_partitioned_proc;
```

5.
```
DCL Nil_partitioned_translations Cc_partitioned_proc IS
     RETURN FALSE;
```

6.

```
TYPE Cibincom PROC(
            UPDATES Pv Parval,       % result of evaluation
            UPDATES Pstr String,     % result if string type
            UPDATES Pd Did,          % dir containing name
            UPDATES Pn Parname,      % name of evaluated par
            UPDATES Rc Ciretcode);   % result of the proc.
```

Example 1 declares P1 to be a procedure with no parameters and no return value. Example 2 declares Get_status as a procedure; the declaration ends with the identifier Status.

Examples 3 and 4 both declare procedure variables. Their definitions will come through assignment statements. Example 3 declares procedure variable Init_device having a formal parameter D of type Device, and having no return type.

Example 5 illustrates a procedure declaration and definition, where the definition is a single statement.

In example 6, a procedure type is declared. Any procedure declared of type Ccibincom will have the parameter list in the Cibincom type declaration.

5.2 Sections, Modules, and Programs

Let us begin our discussion of program structure by defining the objects with which we are dealing. We shall define the terms section, module, and program.

5.2.1 Sections

A SECTION is the smallest compilable unit. It is stored in a file. The name of the section is the name of the file which contains it. The first statement of every section is called the imbed statement. The imbed statement contains the section name, which must be the same as the file name. It also indicates other section-type and module-type information. We will consider only INTERFACE and SECTION section types. The imbed statement contains imbedding information if it contains either of the keywords

USES or **OF**.

EXAMPLES

```
SECTION Prog1ui USES Mydefs, Formato;
SECTION Prog1imp OF Prog1ui;
INTERFACE Formato USES Mydefs, Myio;
INTERFACE Mydefs;
INTERFACE Myio;
```

Prog1ui, Prog1imp and Formato are all sections. Formato is an interface section.

5.2.2 Imbedding and Outbedding

Imbedding a section means opening the object file for that section, and reading the "imbed data". The imbed data for a section consists of the symbols for all data types, and data objects which are global to that section. When the object file for a section y contains such data, we say that y outbeds these symbols. If section x imbeds section y, it can imbed any subset of the symbols outbed by y.

5.2.3 Modules

A module is a collection of one or more sections, connected by OF links. The name of a module is the name of its top-most section.

The top-most section is the only section in a module which is not OF another section. In the above example, Prog1ui is a module consisting of the sections Prog1ui and Prog1imp.

5.2.4 Programs

Let us now examine the USES graph for Prog1ui (see Figure 19). When module x uses module y, we draw an arrow from x to y. Prog1ui uses Mydefs and Formato. Formato also uses Mydefs.[5]

The USES list for Prog1ui is {Formato, Mydefs}, represented by directed edges from Prog1ui to Formato and Mydefs. The USES list for Formato is {Myio, Mydefs}, and is represented by directed edges from Formato to Myio and Mydefs.

Note that Prog1ui uses Formato directly, while it uses Myio indirectly. It uses Myio by virtue of the fact that it uses Formato, which uses Myio. We say that Prog1ui uses Myio transitively.

Prog1ui uses Mydefs both directly and transitively.

The set of all modules used by Prog1ui, either directly or transitively, is called the "transitive closure of its USES list." The definition of a PROTEL program is a module, an entry point (i.e., an ENTRY procedure), and the transitive closure of the USES list of the module.[6]

It is not required that the top-most section of every module be an interface. Of course no module without an interface can be used by any other module. A module which contains an entry point, and thus defines a program, does not generally have an interface.

Note that Prog1ui is such a module. Note also that the lack of an interface in Prog1ui does not prevent it from using other modules.

---

[5] For the mathematically inclined, a USES graph is a “directed acyclic graph" (DAG). There exist no cycles in the directed edges.

[6] It is not required that the entry point be within the module whose transitive closure defines the program. In CC software, however, SOS does require that this be so.

5.3 Compilation Order

A compilation order for PROTEL is an ordering of the sections to be compiled, such that all sections are compiled before they are imbedded.

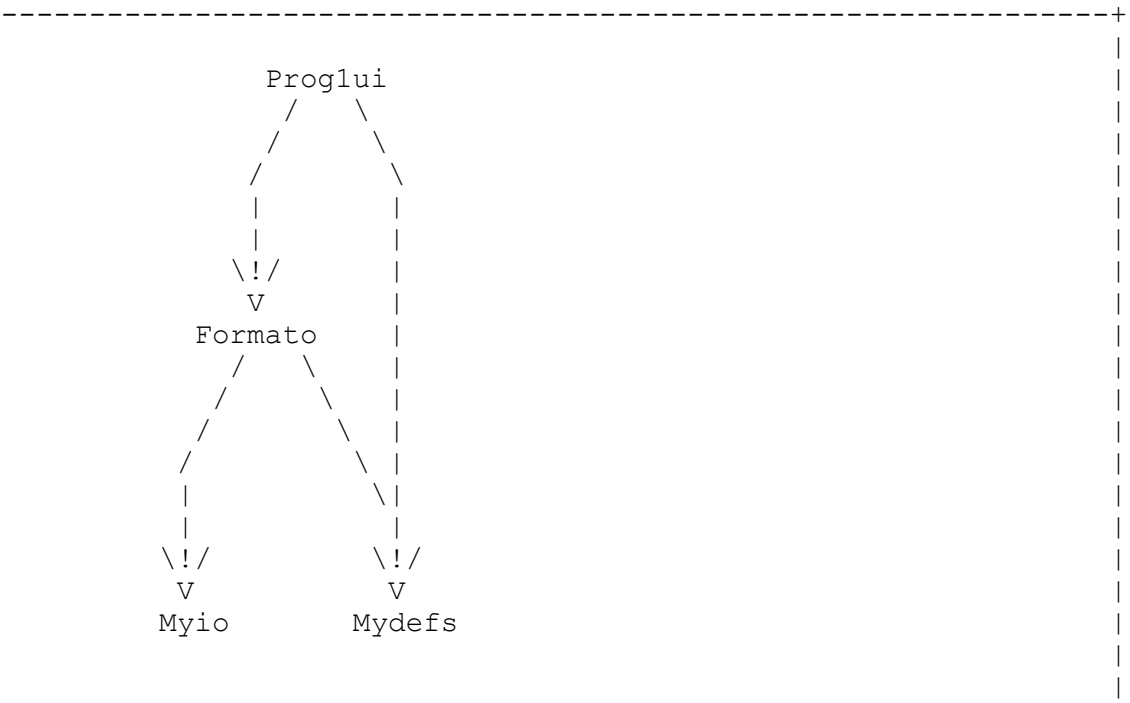

Figure 19. USES Graph for program Prog1ui: a USES graph is a directed acyclic graph.

5.3.1 Compilation Order for Programs

Suppose we wish to compile Prog1ui. To do so requires that the object code for Formato and Mydefs be available, so they must both have been compiled before Prog1ui. Furthermore, Mydefs must have been compiled before Formato, since Formato uses Mydefs. Myio must have been compiled before Formato, as well. Furthermore, every module used by Myio, if any, must have been compiled before Myio.

One can see that determining a correct compilation order can be difficult. The derivation of a compilation order is recursive. As a first attempt at its definition, let us say that "for every module to be compiled, the modules in its USES list must first be compiled."

The above definition is not quite correct. Imbedding is performed not on a per-module basis, but on a per-section basis. The USES list of a module contains a section list. The restriction on the sections occurring in a USES list is that they must be INTERFACE sections. Therefore,

a second attempt at defining compile order might be: "for every module to be compiled, the sections in its USES list must first be compiled."

This means, for example, that even though module Prog1ui may use module Formato, only the interface section, Formato, must be compiled before Prog1ui is compiled.

A glance at Figure 20 should make the reason for this clear. Module Prog1ui uses the identifiers Int and Typeint, which are not declared in Prog1ui. In order to check that identifiers are used correctly, the declaration of all identifiers must be available when they are used. Typeint is declared in Formato, and Int is declared in Mydefs.

Typeint is a FORWARD procedure, which means it is implemented somewhere in module Formato.[7] The purpose of imbedding is to permit full compile-time type checking. The USES list of a given module X specifies those sections which define all of the external symbols used in module X. Since the USES list is composed only of interface sections, only the interface section of each used module needs to be previously compiled.

To type check the use of a procedure, we need to know only the number and types of its formal parameters, and the type of its RETURN value, if any. This information is provided by a FORWARD declaration. Thus a forward declaration generally appears in an interface section, while the body of the procedure occurs in an implementation section. By implementation section, we generally mean a section which is not an INTERFACE section, and which contains the bodies of forward-declared procedures.

---

[7] Since Typeint is not implemented in section Formato, it must be implemented in some section OF Formato.

5.3.2 Compilation Order for Modules

The compilation order for the sections of a module must satisfy the same rule as those for a program: for every section to be compiled, all of its imbedded sections must have been previously compiled.

A section imbeds every section which it is OF or USES, either directly or transitively. Thus Prog1imp imbeds Prog1ui directly, and imbeds indirectly those sections in the USES list of Prog1ui. By definition, the USES list for every section in a module is the USES list of the top-level section. This is sensible since only the top-level section can have a USES list.

As a concrete example, consider the module X in Figure 21.

Section X must be compiled first, followed by X1. For the remaining sections any compile order is valid, as long as X2 is compiled before X3. There are no compile order dependencies between X2, X4, and X5, since they are neither OF each other, nor USE each other.

```
+-------------------------------------------------------------+
| +-----------------------------------------------------+     |
| | SECTION Prog1ui USES Mydefs, Formato;               |     |
| |                                                     |     |
| | DCL Print_and_reset PROC(UPDATES i Int) IS FORWARD; |     |
| +-----------------------------------------------------+     |
|                                                             |
| +-----------------------------------------------------+     |
| | SECTION Prog1imp OF Prog1ui;                        |     |
| |                                                     |     |
| | DCL Print_and_reset PROC(UPDATES i Int) IS          |     |
| | BLOCK                                               |     |
| |                                                     |     |
| |    Typeint('Previous value of i was', i);           |     |
| |    0 -> i;                                          |     |
| | ENDBLOCK;                                           |     |
| +-----------------------------------------------------+     |
|                                                             |
| +-----------------------------------------------------+     |
| | INTERFACE Formato USES Mydefs, Myio;                |     |
| |                                                     |     |
| | DCL Typeint PROC(format DESC OF VAL Char,           |     |
| |                  data Int) IS FORWARD;              |     |
| +-----------------------------------------------------+     |
|                                                             |
| +-----------------------------------------------------+     |
| | INTERFACE Mydefs;                                   |     |
| |                                                     |     |
| | DCL                                                 |     |
| |    Wordsize {16 TO 16} IS 16;                       |     |
| |                                                     |     |
| | TYPE                                                |     |
| |    Int          PACK(Wordsize) {-32768 TO 32767},   |     |
| |    Unsignedint  PACK(Wordsize) {0 TO #FFFF},        |     |
| |    Byte         PACK(Bytesize) {0 TO #FF},          |     |
```

```
| |    Char         Byte;                                    |       |
| +--------------------------------------------------------+       |
|                                                                    |
| Figure 20.  Imbedding  for  Prog1ui:  module  Prog1ui  imbeds |
|             interfaces Formato and Mydefs.                     |
+-------------------------------------------------------------------+
```

To use our previously defined terminology, we call X the "interface", and X2, X3, X4 and X5 the "implementation" sections.

```
+-------------------------------------------------------------------+
|                                                                   |
|   INTERFACE X;                                  X                 |
|   SECTION X1 OF X;                              |                 |
|   SECTION X2 OF X1;                             |                 |
|   SECTION X3 OF X2;                             X1                |
|   SECTION X4 OF X1;                            /| \               |
|   SECTION X5 OF X1;                           / |  \              |
|                                              /  |   \             |
|                                              |  |    \            |
|                                             X2  X4   X5           |
|                                              |                    |
|                                              |                    |
|                                             X3                    |
|                                                                   |
| Figure 21.  Hierarchy  for  module  X:  The  OF  relationship |
|             yields a tree structure.                          |
+-------------------------------------------------------------------+
```

5.4 Imbedding

5.4.1 Differences Between USES and OF Imbedding

When a section X3 OF X2 imbeds X2, it gains access to all global identifiers in X2. This includes types, variables, and constants.[8] We say then, that all global identifiers within the parent section are within the scope of the OF section. This scope rule holds true for sections imbedding through all OF relations, whether direct or transitive.

For example, consider the module X in Figure 21. X3 has access to all global identifiers declared in X2, X1 and X. X3 can alter any global variable declared in X1 or X2.

With USES imbedding, there are slightly different scope rules between direct and transitive imbedding. Constants and variables imbed only directly; types imbed both directly and transitively.

Consider Figure 22. Procedure Update_counter in module Zed updates variable Counter in module Alpha. It is not reasonable that any module which uses Zed should have direct access to Counter through Zed. Note, however, that module Zed does not use module Beta, so the type of Counter (Smallint), is imbedded transitively through Alpha. Suppose Zedimp declared a variable of type Hexdigit.

Would the type Hexdigit be transitively imbedded into Zed? No, there is a catch to transitively imbedding types. Zed imbeds the type Smallint only because it was attached to the variable Counter. Alpha did not reference type Hexdigit, so it did not outbed it.

The next question is, if the type of Counter in Alpha were changed to Posint, would Zedimp still compile? The answer is no, because it would no longer imbed type Smallint. This is not a particularly safe situation. The problem arose because Zed imbedded Smallint "for free." Since the identifier Smallint was referenced inside of module Zed, Zed should have had Beta in its USES list.

```
+-----------------------------------------------------------+
| +-------------------------------------------------+       |
| | INTERFACE Zed USES Alpha;                       |       |
| |                                                 |       |
| | DCL Update_counter PROC() IS FORWARD;           |       |
| +-------------------------------------------------+       |
|                                                           |
| +-------------------------------------------------+       |
| | SECTION Zedimp OF Zed;                          |       |
| |                                                 |       |
| | DCL Local_counter Smallint;                     |       |
| |                                                 |       |
| | DCL Update_counter PROC() IS                    |       |
| | BLOCK                                           |       |
| |   Counter + 1 -> Counter;                       |       |
| |   Local_counter + 1 -> Local_counter;           |       |
```

[8] This includes procedures as well. Remember that in PROTEL a procedure is either a constant or a variable. A procedure implementation a always a constant.

```
| | ENDBLOCK;                                          |        |
| +----------------------------------------------------+        |
|                                                               |
| +----------------------------------------------------+        |
| | INTERFACE Alpha USES Beta;                         |        |
| |                                                    |        |
| | SHARED Counter Smallint;                           |        |
| +----------------------------------------------------+        |
|                                                               |
| +----------------------------------------------------+        |
| | INTERFACE Beta;                                    |        |
| |                                                    |        |
| | DCL                                                |        |
| |    Wordsize {16 TO 16} IS 16;                      |        |
| |                                                    |        |
| | TYPE                                               |        |
| |    Smallint     {0 TO 4095},                       |        |
| |    Posint       {0 TO 32767},                      |        |
| |    Hexdigit     {0 TO #F};                         |        |
| +----------------------------------------------------+        |
|                                                               |
| Figure 22.  Imbedding  for  module  Zed:  type  Smallint   is |
|             transitively imbedded.                            |
+---------------------------------------------------------------+
```

5.4.2 Imbedding of Non-Top-Level Sections

It is possible for an interface to be OF an interface. Such a section may then appear in the USES list of some other module. Suppose section X1 in Figure 21 had been declared as

```
INTERFACE X1 OF X;
```

Now consider a module Y which uses X1.

```
INTERFACE Y USES X1;
```

Since X1 is OF X, Y transitively imbeds X. Furthermore, Y imbeds X as though it were OF X, so it imbeds not only types, but variables, and constants as well. The presence of both X, and X1 in the uses list for Y is equivalent to the presence of X1 alone.

Similarly, had X2 in Figure 21 been declared an INTERFACE, Y could have used X2, and would have imbedded X2, X1, and X. In general, when a used interface is not the top-most section in a module, a user of the interface will imbed all sections of the module from the used interface up to the top.

5.4.3 Private Interfaces

It often happens that an interface is required within a module, for the purpose of sharing types, variables, and constants within the module. X1 in Figure 21 is such a section. In general, whenever a section has more than one section OF itself, it is an interface for the sections below it. If no other modules are to be permitted to use this private interface, it will be declared as a SECTION, not as an INTERFACE.

There is no keyword in PROTEL to denote an interface which is not usable by all other modules.

5.5 PROTEL Language

5.5.1 Design Goals

PROTEL is designed to support large-scale programming projects. The term "embedded systems" applies well to the types of programs implemented in PROTEL. Some of the requirements of such systems are: modularity, re-usability, and safety. We will discuss each of these concepts drawing examples from PROTEL.

Modularity refers to the ability to localize software features. A PROTEL module should provide some well-defined set of features. The interface to a module should provide documentation of the features, and should contain only those types and procedures which a user of the module must have access to. The procedures should be forward-declared, and the bodies of the forward-declared procedures should not appear in the interface. A user of the procedures need not have access to the implementation of the procedures.

Consider the compilation order of module Figure 21, and a module Y which uses X.

```
INTERFACE Y USES X;
```

To compile all of modules X and Y, one possible compilation order is {X,X1,X2,X3,X4,X5,Y}. Another possible compilation order is {X,Y,X1,X2,X3,X4,X5}.

We now ask the question: if section X3 were modified, which sections would need to be recompiled? The answer is, only section X3.

The reason is that no section imbeds X3, so no section requires recompilation because of a change to X3. In general, the implementation of a procedure is modified more often than its declaration. If a procedure p were implemented in interface section X, and the body of p required modification, then the sections requiring recompilation would be all those sections which imbed X. In our current example, this would be {X,X1,X2,X3,X4,X5,Y}.

The restriction of procedure bodies to implementation sections has implications on the replaceability of modules, an issue we shall not discuss further. We point out, however, that it is desirable that the number of sections requiring recompilation due to a change in a given module be kept as small as possible.

In this context, our module also becomes reusable. When the use of a module depends only upon the interface it presents, and not upon its implementation, it is highly likely that the module will be used by many other modules. That it, it will appear in many USES lists.[9] Consider the modules Myio, Mydefs and Formato, of Figure 19. Prog1ui is some program which requires formatted I/O. Mydefs provides some basic types, Myio implements an I/O package, and Formato provides a facility for performing formatted I/O of these types.

The likelyhood is high that a second program will be written which requires formatted I/O of the types in Mydefs. If Formato appears in the USES graph of a second program, say Prog2ui,

---

[9] When a module appears in the transitive closure of different modules which contain entry procedures, it has become imbedded in each of these programs. This is a nice example of reusable software.

it is reused. If Formato has implementation sections which require modification even due to hardware changes, the users of Formato will not even require recompilation.

Safety, in terms of program organization, involves structuring the modules of a program in such a way as to minimize the effects of changes to USED modules on one's own modules. We saw earlier[10] how a module X can reference an imbedded type from module Y without having the module Y in its USES list. This exposes module X to effects from modifications to module Y which are irrelevant to the facilities provided by Y.

Using a private interface is not a safe technique, since private interfaces tend to be implementation-dependent. An important use of private interfaces is to declare procedure variables, otherwise known as aspects, a topic discussed in DIS OFLPI, Using PROTEL in DMS.

One rule of imbedding is that an INTERFACE may only be OF another INTERFACE. This prevents an interface from being hung from an implementation and thus allowing another module to imbed the implementation section. In this respect implementation sections are safe from extra-modular imbedding (i.e., being imbedded from outside the module).

An interface module provides either (a) a set of functions, such as Formato, or (b) a set of types. There exist DEFINITIONS modules, which are interface modules which contain nothing but type declarations.

Imbedding statements provide static means of communication between modules and programs, via shared variables and procedures; they also privide a consistent environment via shared types.[11]

---

[10] See "Differences between USES and OF Imbedding".

[11] Dynamic communication is not provided by the PROTEL language, as it is in some other languages. In the DMS context, it is provided by the operating system.

5.5.2 A Sample Program

Prog3ui Figure 23 is a module with an entry procedure. Together with its USES list it defines a program. Prog3ui prints out a list of the primes less than 100. An explanation of the statements in section Prog3ui follows.

Statement and Meaning

1. Imbed statement. Prog3ui is a program (no INTERFACE) which uses modules Mydefs and Formato.
2. Comment.
3. Declares Start as the entry procedure and as forward.
4. Declares Start procedure body.
5. Begins the block defining the procedure body.
6. Begins a local declaration statement.
7. Declares an Int constant with value 100. Since Int is not declared in Prog3ui, it must be declared in either Mydefs or Formato.
8. Declares an table variable of 100 BOOL elements.
9. Initialises all elements of prime to TRUE.
10. A loop which will iterate 99 times.
11. An if statement which is entered if prime[i] is TRUE.
12. A print statement which outputs i. Since Typeint is not declared in Prog3ui, it must be declared in either Mydefs or Formato.
13. An if statement which iterates over all multiples of i less than or equal to limit.
14. Assignment statement sets prime[multiple_of_i] to FALSE.
15. Ends the loop begun on statement 16.
16. Ends the if statement begun on statement 14.
17. Ends the loop begun on statement 13.
18. Ends the block begun on statement 6.

There is no ambiguity when we say that statement 3 declares Start as both the entry procedure and as a forward procedure. An entry procedure must de defined in the module in which it is declared.

An entry procedure, as well, may not have either a formal parameter list or a RETURNS type; hence the IS ENTRY declaration fixes the procedure's type as "PROC();" and serves as the FORWARD declaration as well.

```
+-------------------------------------------------------------+
|                                                             |
|     1 |SECTION Prog3ui USES Mydefs, Formato;                |
|     2 |% print out prime numbers.                           |
|     3 |DCL Start PROC() IS ENTRY;                           |
|     4 |                                                     |
|     5 |DCL Start PROC() IS                                  |
|     6 |BLOCK                                                |
|     7 |   DCL                                               |
|     8 |      limit Int IS 100,                              |
|     9 |      prime TABLE[1 TO limit] OF BOOL;               |
|    10 |                                                     |
|    11 |   limit {TRUE} -> prime[];                          |
|    12 |                                                     |
|    13 |   FOR i FROM 2 UP TO limit DO                       |
```

```
|     14 |       IF prime[i] THEN                                          |
|     15 |          Typeint('next prime = ', i);                           |
|     16 |          FOR multiple_of_i FROM i UP TO limit BY i DO           |
|     17 |             FALSE -> prime[multiple_of_i];                      |
|     18 |          ENDDO;                                                 |
|     19 |       ENDIF;                                                    |
|     20 |    ENDDO;                                                       |
|     21 |                                                                 |
|     22 |ENDBLOCK;                                                        |
|                                                                          |
|                                                                          |
| Figure 23.  Prog3ui:  This  program  prints  the  primes less            |
|             than 100.                                                    |
+--------------------------------------------------------------------------+
```

5.5.3 Program Organization: Points to Remember

1. Imbedding is done on a per-section basis. A section imbeds sections, not modules.

2. Previous to compiling a section, every section which it imbeds, either directly or transitively, must have been compiled.

3. There is no PROTEL construct which tells a section which sections are OF it. Therefore, no section in a module can imbed from below itself.

4. Only INTERFACE sections can appear in a USES list.

5. The name of a section is the name of the file in which it is stored.

6. The name of a module is the name of its top-most section.

7. Types imbed transitively. Variables, constants, and procedures can only be imbedded directly. Types referenced within a module should be imbedded directly.

6.0 Summary

Identifiers are used in PROTEL to allow the programmer to name data types, constants, variables, procedures, statement labels, and sections.

PROTEL allows the definition of types in terms of existing primitive types in the language or in terms of other types defined by the programmer. Once a type is defined, it can be used in any context where a type is required. All data used in PROTEL programs must be defined in terms of a type. This allows the compiler to perform extensive type checking at compile time.

PROTEL provides single-entry, single-exit control structures. This enables good structured programming techniques, which yield programs which are more easily implemented, understood, and maintained.

7.0 PROTEL 2026 Development Tools

The PROTEL 2026 toolchain ships with a small set of command-line and editor tools intended for MacOS and UNIX / Linux hosts with minimal dependencies.

7.1 The PROTEL Compiler - Pc

The PROTEL Compiler is can be invoked from a MacOS or UNIX / Linux command line with:

```
> Pc [-o outfile] sourcefile(s) [—classical]
```

`-o outfile`: Specify output (executable or object file). Defaults to a.out.
sourcefile(s): One or more .P files (PROTEL source).

`--classical`: Enables case-insensitivity for identifiers (compatibility with original Nortel DMS). Keywords remain UPPERCASE.

Pc is actually a t*ranspiler*. The transpiler uses GCC backend: Generates C intermediates, compiles to native code. Supports -g (debug), -c (compile only), etc., passed to GCC. PROTEL PROCs are C-callable; use extern in C for vice-versa.

7.2 The PROTEL Beautifier - Pb

Pb is a lexical beautifier (or “pretty-printer”) for PROTEL source code. It reads PROTEL text, performs a simple lexical analysis, and rewrites every reserved keyword in UPPERCASE. With the --bold option it additionally surrounds each keyword with ANSI SGR escape sequences (\033[1m … \033[0m) so that the keywords appear in bold on terminals and in editors that support ANSI escape sequences.

Pb knows the complete set of PROTEL reserved words given in Appendix A of this manual, together with the compiler directives ($EJECT, $LI, $OBJECT, …). A keyword is only uppercased (and optionally bolded) when it appears as a token outside a comment or string literal. Comments (introduced by % and running to the end of the line) and string literals (delimited by apostrophes, with '' representing an embedded apostrophe) are left completely untouched. All other tokens — user identifiers, numeric literals, operators (including ->), punctuation, and whitespace — are preserved exactly as they were written.

The program is intentionally a pure filter. It can be used with redirection, pipes, or explicit file-name arguments for both input and output. If the input contains very few PROTEL keywords, Pb emits a warning on standard error but still produces the (minimally modified) output.

The original Pb was created in the early 1990s by Vincente D’Ingianni using C and Lex; however, the original source code was lost. This new version for PROTEL 2026 is written in Python for ease of portability.

Invocation

Pb may be invoked in any of the following ways:

```
> Pb < infile > outfile
> Pb infile outfile
> Pb --bold < infile | less
> cat infile | Pb
> ./Pb myprog.P > myprog.pretty.P
```

After chmod +x Pb the program can be executed directly because it begins with the standard Python 3 shebang (#!/usr/bin/env python3).

Arguments and Options

```
usage: Pb [-h] [-b] [--version] [input] [output]
```

- `input`
  Input PROTEL file (default: read from standard input). Use “-” to read explicitly from stdin.

- `output`
  Output file (default: write to standard output). Use “-” to write explicitly to stdout.

- `-b, --bold`
  Wrap each uppercased keyword in ANSI bold escape sequences (\033[1m … \033[0m).

- `--version`
  Print the program identification string and exit.

- `-h, --help`
  Print a short usage message and exit.

Version string

```
Pb 1.0 (PROTEL Beautifier)
```

The keyword list used by Pb is taken directly from this PROTEL Reference Manual (Appendix A).

7.2 EMACS PROTEL Mode

The PROTEL 2026 distribution includes EMACS PROTEL mode for GNU Emacs in honor of Bill Conn, original author of EMACS PROTEL mode at Nortel/BNR.

It provides:

- Keyword highlighting for PROTEL 2026 reserved words (UPPERCASE keywords per the Reference Manual)
- Comment recognition (`%` to end of line)

• String highlighting (single-quoted PROTEL strings)
• Pascal-style indentation for `BLOCK` / `ENDBLOCK`, `IF` / `ENDIF`, and related structures

After Emacs loads this block, `protel-mode` is available and **file-name detection is automatic** for the following extensions;

• .P
• .protel
• .pt
• .ptl
• .AA01 … .ZZ99. (legacy PLS version extensions)

The PLS pattern matches any extension of the form **`.LLNN`** where `L` is a letter (`A`–`Z` or `a`–`z`) and `N` is a decimal digit. Examples: `vdi.aa01`, `module.AB18`, `Hello.aa01`.

PROTEL Mode provides capitalization, text highlighting, and proper indentation for EMACS users via Pb. In Emacs, run Pb on the current buffer with **`C-c C-f`** (`M-x protel-beautify`)

No extra `auto-mode-alist` configuration is required in `.emacs`; `protel-mode.el` registers these extensions when it is loaded.

The PROTEL mode file is `emacs/protel-mode.el` in the PROTEL project tree

(setq protel-user-root "/path/to/PROTEL")

`protel-mode` uses Emacs **Font Lock** to highlight keywords, comments, and strings. The mode does **not** hard-code RGB colors in the buffer; it assigns syntax classes to named **faces**, and Emacs (or your color theme) supplies the actual display attributes.

• Reserved keywords - `protel-keyword-face` - **Bold white** (via `face-remap` from `font-lock-keyword-face`)
• `%` comments - `protel-comment-face` - *Green italic*
• `#!` shebang (line 1) - `protel-comment-face` - *Green italic*
• `'...'` strings - `protel-string-face` - Orange (inherits theme string color)

Font Lock applies the standard `font-lock-*` faces; `protel-mode` remaps them to `protel-*` faces so highlighting works reliably and you can customize colors with `set-face-attribute` or `M-x customize-face`.

The project `.emacs` in the PROTEL distribution loads `protel-mode`, enables global Font Lock, and sets the `protel-*` face colors (bold keywords, green italic comments).

Customizing colors in `.emacs`

Override any face after loading the mode:

```
(with-eval-after-load 'protel-mode
  ;; Bold keywords (default)
  (set-face-attribute 'protel-keyword-face nil :weight 'bold)

  ;; Green italic comments (default)
  (set-face-attribute 'protel-comment-face nil
                      :foreground "green"
                      :slant 'italic)

  ;; Optional: softer green on dark backgrounds
  ;; (set-face-attribute 'protel-comment-face nil :foreground "dark olive green"))
```

Use `M-x customize-face RET protel-comment-face RET` to adjust interactively.

7.3 Foreign Language Interoperability

PROTEL 2026 programs can call routines written in C, C++, Swift, Rust, Python (via a thin C embedding shim), Java (via JNI), and other languages that support the **C calling convention**. Likewise, foreign programs can call procedures you publish from PROTEL. This is the recommended approach for I/O, platform APIs, and performance-critical libraries: PROTEL does not use INTRINSIC for ordinary host I/O in the 2026 toolchain.

All foreign interop goes through two declarations:

```
| Direction | PROTEL keyword | Foreign side |
|-----------|----------------|--------------|
| Foreign → PROTEL | **EXPORT** | Import C symbol (e.g. `extern "C"`, Swift
`@_silgen_name`, Rust `extern "C"`, Python `ctypes.CDLL`, Java JNI bridge) |

| PROTEL → Foreign | **EXTERNAL** | Export C symbol (e.g. C compilation unit,
`#[no_mangle]`, `@_cdecl`, Python C API shim, JNI JVM embed) |
```

Working examples live under `examples/interop/` in the PROTEL distribution.

7.3.1 Strings and the C boundary

PROTEL character strings are **tuples of bytes** with a compile-time length. They are **not** automatically NUL-terminated when pa
ssed to C.

When a C, Swift, Rust, Python (via C), or Java (via JNI/C) function expects a `char*` / C string, append an explicit zero byte in a char tuple:

```
writeln({'Hello', 0});
swift_greet({'PROTEL', 0}, 42);
```

The compiler emits a `(const char[]){ ... }` array; the trailing `0` is your responsibility.

7.3.2 Calling foreign code from PROTEL (EXTERNAL)

Step 1 — Declare the foreign routine

```
DCL writeln PROC(msg PTR TO char) EXTERNAL;
```

Step 2 — Implement it in the foreign language

C (`examples/interop` pattern / `protel_io.c`):

```
void writeln(const char* msg) {
    /* write(2) based implementation */
```

```
}
```

Rust (`rust_greet.rs`):

```
#[no_mangle]
pub extern "C" fn rust_greet(name: *const i8, value: i16) { /* ...
*/ }
```

Swift (`swift_greet.swift`):

```
@_cdecl("swift_greet")
public func swift_greet(_ name: UnsafePointer<CChar>, _ value: Int32)
{ /* ... */ }
```

Python (via C embedding shim `python_greet.c`):

```
void python_greet(const char* name, int16_t value) {
    /* Py_Initialize, import greet_helper, call greet(name, value) */
}
```

PROTEL does not call the Python interpreter directly. A small **C** translation unit exposes `python_greet` with C linkage; it embeds Python and forwards to `greet_helper.py`. Link against `libpython` (Python development headers required).

Java (via JNI embedding shim `java_greet.c`):

```
void java_greet(const char* name, int16_t value) {
    /* JNI_CreateJavaVM, FindClass("GreetHelper"),
CallStaticVoidMethod greet */
}
```

PROTEL does not call the JVM directly. A **C** shim exposes `java_greet` with C linkage, embeds the JVM, and forwards to `GreetHelper.java`. Link against `libjvm` (`$JAVA_HOME/include`, JDK required).

Step 3 — Call from PROTEL

```
DCL Start PROC() IS ENTRY;

DCL Start PROC() IS
   BLOCK
      rust_greet({'PROTEL', 0}, 42);
```

```
      writeln({'done', 0});
   ENDBLOCK;
```

Step 4 — Build and link

```
> protel myprog.protel --keep -o build/myprog.cpp
> clang++ -std=c++20 -c build/myprog.cpp -o build/myprog.o
> rustc --crate-type staticlib rust_greet.rs -o build/librust_greet.a
> clang++ build/myprog.o build/librust_greet.a src/runtime/protel_io.c -o myprog
```

See `examples/interop/build_interop.sh`, `build_interop_rust.sh`, `build_interop_python.sh`, and `build_interop_java.sh`.

7.3.3 Publishing PROTEL routines for foreign callers (EXPORT)

Step 1 — Forward EXPORT declaration

```
INTERFACE protel_math;

TYPE integer {-32768 TO 32767};

DCL protel_add PROC(a integer, b integer) RETURNS integer EXPORT;
```

Step 2 — Procedure body (same section)

```
DCL protel_add PROC(a integer, b integer) RETURNS integer IS
   BLOCK
      RETURN a + b;
   ENDBLOCK;
```

The transpiler places the implementation in an `extern "C"` block so the symbol name is `protel_add`.

Step 3 — Import from foreign code

C:

```
#include <stdint.h>
int16_t protel_add(int16_t a, int16_t b);
```

Rust (`rust_calls_protel.rs`):

```
extern "C" {
    fn protel_add(a: i16, b: i16) -> i16;
}
```

Swift (`swift_calls_protel.swift`):

```
@_silgen_name("protel_add")
func protel_add(_ a: Int16, _ b: Int16) -> Int16
```

Python (`python_calls_protel.py`):

```
import ctypes

lib = ctypes.CDLL("build/interop/libprotel_for_python.dylib")  # .so
on Linux
lib.protel_add.argtypes = (ctypes.c_int16, ctypes.c_int16)
lib.protel_add.restype = ctypes.c_int16
sum = lib.protel_add(40, 2)
```

Build the PROTEL EXPORT module as a **shared library** (`.dylib` / `.so`), then load it with **`ctypes.CDLL`**. No Python extension module is required on the PROTEL side.

Java (`JavaCallsProtel.java` + `java_calls_protel.c`):

```
public final class JavaCallsProtel {
    static { System.loadLibrary("javacallsprotel"); }
    private static native short protelAdd(short a, short b);
}
```

Compile the EXPORT module and a thin **JNI** bridge into `libjavacallsprotel.dylib` (`.so` on Linux). `System.loadLibrary` loads the combined library; the JNI stub forwards to the EXPORTed `protel_add` symbol.

Step 4 — Link

```
> protel protel_for_rust.protel --keep -o build/protel_for_rust.cpp
> clang++ -std=c++20 -c build/protel_for_rust.cpp -o build/protel_for_rust.o
> rustc rust_calls_protel.rs build/protel_for_rust.o -o rust_calls_protel
```

Python (shared library + ctypes):

```
> protel protel_for_python.protel --keep -o build/protel_for_python.cpp
> clang++ -std=c++20 -fPIC -c build/protel_for_python.cpp -o build/
protel_for_python.o
> clang++ -shared -o build/libprotel_for_python.so build/protel_for_python.o
> python3 examples/interop/python_calls_protel.py
```

Java (shared library + JNI):

```
> protel protel_for_java.protel --keep -o build/protel_for_java.cpp
> clang++ -std=c++20 -fPIC -c build/protel_for_java.cpp -o build/protel_for_java.o
> clang -fPIC -c java_calls_protel.c -o build/java_calls_protel.o -I$JAVA_HOME/include
> clang++ -shared -o build/libjavacallsprotel.so build/protel_for_java.o build/
java_calls_protel.o
> java -Djava.library.path=build/interop -cp build/interop JavaCallsProtel
```

7.3.4 Complete minimal examples

PROTEL calls Rust

File: `examples/interop/protel_calls_rust.protel`

```
SECTION protel_calls_rust;

TYPE char {0 TO 255};
TYPE integer {-32768 TO 32767};

DCL rust_greet PROC(name PTR TO char, value integer) EXTERNAL;
DCL writeln PROC(msg PTR TO char) EXTERNAL;

DCL Start PROC() IS ENTRY;

DCL Start PROC() IS
   BLOCK
      rust_greet({'PROTEL', 0}, 42);
      writeln({'PROTEL called Rust', 0});
   ENDBLOCK;
```

PROTEL calls Python

File: `examples/interop/protel_calls_python.protel`

```
SECTION protel_calls_python;

TYPE char {0 TO 255};
TYPE integer {-32768 TO 32767};

DCL python_greet PROC(name PTR TO char, value integer) EXTERNAL;
```

```
DCL writeln PROC(msg PTR TO char) EXTERNAL;

DCL Start PROC() IS ENTRY;

DCL Start PROC() IS
   BLOCK
      python_greet({'PROTEL', 0}, 42);
      writeln({'PROTEL called Python', 0});
   ENDBLOCK;
```

Rust calls PROTEL

File: `examples/interop/protel_for_rust.protel` (same EXPORT library as Swift example)

```
INTERFACE protel_math;

TYPE integer {-32768 TO 32767};

DCL protel_add PROC(a integer, b integer) RETURNS integer EXPORT;

DCL protel_add PROC(a integer, b integer) RETURNS integer IS
   BLOCK
      RETURN a + b;
   ENDBLOCK;
```

PROTEL calls Java

File: `examples/interop/protel_calls_java.protel`

```
SECTION protel_calls_java;

TYPE char {0 TO 255};
TYPE integer {-32768 TO 32767};

DCL java_greet PROC(name PTR TO char, value integer) EXTERNAL;
DCL writeln PROC(msg PTR TO char) EXTERNAL;

DCL Start PROC() IS ENTRY;

DCL Start PROC() IS
   BLOCK
      java_greet({'PROTEL', 0}, 42);
      writeln({'PROTEL called Java', 0});
   ENDBLOCK;
```

Python calls PROTEL

File: `examples/interop/python_calls_protel.py` loads `protel_for_python.protel` as a shared library via **ctypes**.

Java calls PROTEL

File: `examples/interop/JavaCallsProtel.java` loads `protel_for_java.protel` via **JNI** (`java_calls_protel.c` bridge).

7.3.5 ENTRY vs EXPORT

- **IS ENTRY** — marks the **program** entry point. The transpiler emits a host `int main()` that calls your ENTRY procedure. Required for standalone executables.

- **EXPORT** — marks a **library** entry point for foreign languages. Does not create `main`. Use in INTERFACE sections intended to be called from Swift, Rust, Python, Java, C, etc.

A single module may have **one ENTRY** and **many EXPORT** procedures.

7.3.6 Quick reference

```
| Task | PROTEL | Build driver |
|------|--------|--------------|
| Hello World (PROTEL only) | `protel Hello.protel --run` | `protel` + `clang++` |
| Hello World (direct shebang) | `./examples/Hello.protel` | §7.0.5 — `export PATH="$PWD:$PATH"`,
file executable |
| PROTEL + Swift | `make interop-swift` | `examples/interop/build_interop.sh` |
| PROTEL + Rust | `make interop-rust` | `examples/interop/build_interop_rust.sh` |
| PROTEL + Python | `make interop-python` | `examples/interop/build_interop_python.sh` |
| PROTEL + Java | `make interop-java` | `examples/interop/build_interop_java.sh` |
| Edit `.protel` / `.P` / PLS `.AA01` in Emacs | Add `emacs/` to `load-path`, `(require 'protel-
mode)` | See §7.2 |
| Beautify source | `Pb file.protel` | `Pb` |

---
```

7.3.7 Troubleshooting

```
| Symptom | Likely cause |
|---------|----------------|
| Link error: undefined symbol | EXTERNAL name mismatch; foreign function not `extern "C"` /
`#[no_mangle]` / `@_cdecl` |
| Garbled string output | Missing `, 0` NUL suffix on char tuple |
| Duplicate `main` at link | Swift/Rust object compiled with its own `main`; use `-parse-as-
library` for helper objects |
| `ctypes` cannot load library | Build EXPORT module as `.dylib`/`.so` first; pass full path to
`CDLL` |
| Python embed link error | Install Python dev package (`python3-dev`); use `python3-config --
embed --ldflags` (or `python3-config --ldflags`) |
| Python greet silent | Embedded `print` may need `flush=True` when stdout is not a TTY |
```

```
| `UnsatisfiedLinkError` (Java) | Set `-Djava.library.path` to the directory containing
`libjavacallsprotel`; build JNI bridge first |
| JVM embed link error | Set `JAVA_HOME` to a JDK; link `-L$JAVA_HOME/lib/server -ljvm` with
rpath |
| No `main` at link | Program section missing **IS ENTRY** |
| `multiple ENTRY` error | More than one `IS ENTRY` in the module |
```

For the formal semantics of **EXTERNAL** and **EXPORT**, see Reference Manual §13.0.

Appendix A. Keywords

AREA	**NONTRANSPARENT**
AS	**NOTINCL**
BIND	**OF**
BLOCK	**OPERAND**
BOOL	**OUT**
BY	**OVER**
CASE	**OVLY**
CAST	**PACK**
CLASS	**PERPROCESS**
DCL	**PRIVATE**
DEFINITIONS	**PROC**
DESC	**PROTECTED**
DO	**PTR**
DOWN	**QUICK**
ELSE	**REF**
ENDAREA	**REFDESC**
ENDBLOCK	**REFINES**
ENDCASE	**RETURN**
ENDCLASS	**RETURNS**
ENDDO	**SECTION**
ENDIF	**SELECT**
ENDOVLY	**SET**
ENDSELECT	**SELF**
ENDSTRUCT	**SHARED**
ENTRY	**STRUCT**
EXIT	**TABLE**
FALSE	**TDSIZE**
FAST	**THEN**
FOR	**TO**
FORWARD	**TRUE**
FROM	**TYPE**
IF	**TYPEDESC**
IN	**UNRESTRICTED**
INCL	**UP**
INIT	**UPB**
INTERFACE	**UPDATES**
INTRINSIC	**USES**
IS	**VAL**
LITERAL	**VARDESC**
METHOD	**VARIABLE**
MOD	**WHILE**
NIL	**WITH**

Figure 24. Reserved Words: These keywords are reserved and may not be used as identifiers.

Appendix B. Special Symbols

=	.
(	^
)	<
;	>
,	&
:	|
@	!
{	+
}	-
[	*
]	/
$	#
_	%
(*	<<
*)	>>
(|	->
|)	

Figure 25. Special Symbols