PLANC
Reference Manual

ND--60.117.04 Revision A

Norsk Data

PLANC
Reference Manual

ND--60.117.04 Revision A

NOTICE

The information in this document is subject to change without notice. Norsk Data A.S assumes no responsibility for any errors that may appear in this document. Norsk Data A.S assumes no responsibility for the use or reliability of its software on equipment that is not furnished or supported by Norsk Data A.S.

The information described in this document is protected by copyright. It may not be photocopied, reproduced or translated without the prior consent of Norsk Data A.S.

PRINTING RECORD

Printing	Notes
10/79	Original Printing
06/80	Second Edition
01/82	Third Edition
06/83	Fourth Edition
12/83	Revision A
	The following pages have been revised or added:
	ix, 2, 4, 8 - 9, 11, 20 - 21, 35 - 36, 38 - 39, 43, 45, 48, 58, 61, 72, 79, 87,
	102, 104, 107, 110, 119, 126 - 127, 129, 138 - 140, 149, 151, 154 - 156,
	156a - 156b, 158 - 162, 166, 173, 175, 182, 184, 186 - 188, 190,
	205 - 206, 216, 219, 225 - 226, 226a, 229, 231, 237 - 239, 241 - 243, 245,
	249, 251 - 253, 279 - 288, 291, 295 - 304
	Page 305 has been removed.

Publ. No. ND-60.117.04, Rev. A
PLANC Reference Manual
December 1983

NORSK DATA A.S
P.O. Box 4, Lindeberg gård
Oslo 10, Norway

Manuals can be updated in two ways, new versions and revisions. New versions consist of a complete new manual which replaces the old manual. New versions incorporate all revisions since the previous version. Revisions consist of one or more single pages to be merged into the manual by the user, each revised page being listed on the new printing record sent out with the revision. The old printing record should be replaced by the new one.

New versions and revisions are announced in the ND Bulletin and can be ordered as described below.

The reader's comments form at the back of this manual can be used both to report errors in the manual and to give an evaluation of the manual. Both detailed and general comments are welcome.

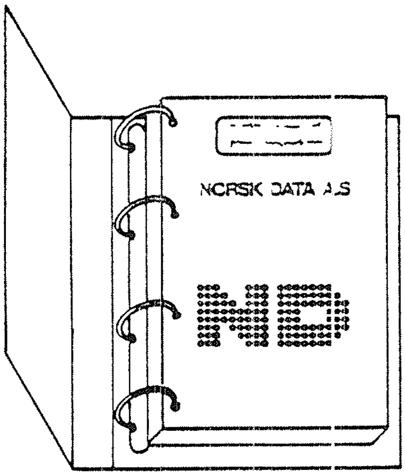
These forms, together with all types of inquiry and requests for documentation should be sent to the local ND office or (in Norway) to:

Documentation Department
Norsk Data A.S
P.O. Box 4, Lindeberg gård
Oslo 10

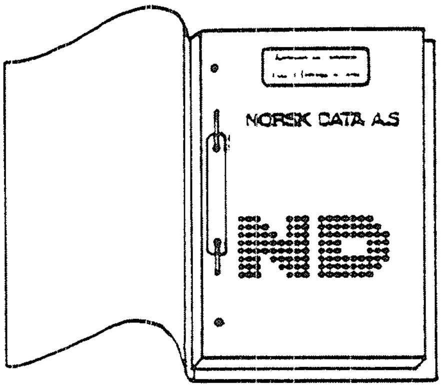
This manual is in loose leaf form for ease of updating. Old pages may be removed and new pages easily inserted if the manual is revised.

The loose leaf form also allows you to place the manual in a ring binder (A) for greater protection and convenience of use. Ring binders with 4 rings corresponding to the holes in the manual may be ordered in two widths, 30 mm and 40 mm. Use the order form below.

The manual may also be placed in a plastic cover (B). This cover is more suitable for manuals of less than 100 pages than for large manuals. Plastic covers may also be ordered below.



A Ring Binder



B Plastic Cover

Please send your order to the local ND office or (in Norway) to:

Documentation Department
Norsk Data A.S
P.O. Box 4, Lindeberg gård
Oslo 10

ORDER FORM

I would like to order

.... Ring Binders, 30 mm, at nkr 20,- per binder

.... Ring Binders, 40 mm, at nkr 25,- per binder

.... Plastic Covers at nkr 10,- per cover

Name ....
Company ....
Address ....
....
City ....

Preface:

THE PRODUCTS

This manual describes products which run under the SINTRAN III operating system :

Compilers

PLANC Compiler - ND-100	ND-10309 (release D)
PLANC Compiler - ND-500	ND-10310 (release C)
PLANC Compiler - MC68000	ND-10491 (release B)

Run-time Systems

PLANC-1BANK	- ND-100	ND-10309
PLANC-2BANK	- ND-100	ND-10309
PLANC	- ND-500	ND-10310
PLANC-MC68	- MC68000	ND-10491

THE READER

This manual will be of interest to the users wishing to write or read PLANC programs.

PREREQUISITE KNOWLEDGE

The reader should have had some programming experience prior to using a systems programming language like PLANC. A general knowledge of compilation and execution of programs under the SINTRAN III operating system would also be useful.

RELATED MANUALS

Related manuals for basic SINTRAN knowledge :

SINTRAN III Introduction	ND-60.125
SINTRAN III Time-sharing Batch Guide	ND-60.132

THE MANUAL

This manual is primarily intended for reference purposes and is organised in a progressive sequence of topics from chapter 2 onwards. Chapter 1 however, is intended to give an overview of the whole language for the less experienced programmer, or for a user only requiring a reading knowledge of PLANC programs.

This version of the manual corresponds to releases, noted above, of the various PLANC compiler products.

This version of the manual contains details of the new product, the compiler for the MC68000. Two new Appendices have been added. Otherwise the changes and additions are mainly corrections and clarification of details which have been reported by users.

TABLE OF CONTENTS

Section Page

Section Page

Section Page

APPENDIX

Notation In This Manual

The notation used throughout the manual to describe PLANC statements and constructs is listed below :

1) Square brackets, [and], indicate optional items.

2) An ellipsis, ..., following square brackets specifies that the preceding optional items may appear one or more times in succession.

3) Parentheses, (and), sometimes referred to as round brackets, are part of the PLANC language and must be coded where shown.

4) Blanks are used to improve readability, but unless otherwise noted have no significance.

1 INTRODUCTION_AND_OVERVIEW_OF_THE_PLANC_LANGUAGE

The PLANC (Programming Language ND Computers) is designed as a high-level systems programming language. It is a member of the ALGOL/PASCAL family of block structured languages. PLANC is used mainly for writing systems software such as operating systems and compilers. It has been defined in a machine-independent manner and machine-dependent features (eg. data allocation strategies,interfaces to programs in other languages) for particular machines will be specifically noted in this manual.

In the late 60's and early 70's many computer scientists and software developers identified the 'software crisis'. One trend from this recognition of problems and difficulties in software development was that using assembly languages for large software projects was inadequate. The first move was more extensive use of macro processors to create single language constructs which gave more powerful facilities to an assembly language, in a reliable and consistent way. The next step was to develop 'middle-level' languages, primarily for systems programming, but with features similar to the popular high-level languages, eg. Fortran, Cobol and Algol. A notable middle-level language was PL360, developed by N. Wirth for the IBM S/360, and was the forerunner of PASCAL which is very widely used now.

The early 70's saw the emergence of PASCAL, BCPL, BLISS, C and other languages designed for writing systems software such as compilers and operating systems. Some of these developments had as a side benefit, fairly straightforward techniques for implementation on various hardware. System software development began to escape from the exclusive province of the hardware manufacturers. Further, these languages extended some areas in which the previous high-level languages were limited or simply did not have, eg. data structures and the so-called structured programming control mechanisms IF-THEN-ELSE, CASE and DO-WHILE etc. This has also affected the recent development of general-purpose languages, namely some of the particular features specified for Fortran 77 and Ada.

1.1 PLANC LANGUAGE OVERVIEW

This chapter is a detailed overview of the PLANC language and should enable programmers to read and understand PLANC programs. A detailed presentation of PLANC will appear in later chapters, for those who wish to write large complex programs and systems or to interface to programs and systems written in PLANC.

1.2 A SIMPLE PLANC PROGRAM

PLANC programs are structured into modules and routines; the routine concept, as will be seen, is a broad one compared with other programming languages.

But first a simple example. The program below consists of a module mudpie which contains a routine mprog, of the special routine type, main program, for specifying the entry point at execution time. The program also contains some examples of simple declarations, a standard routine, and the use of the assignment operator.

EXAMPLE 1.1 A VERY SIMPLE PROGRAM

```
MODULE mudpie
INTEGER ARRAY : stack (0:100)
PROGRAM : mprog
INTEGER : i,j,k,m
INISTACK stack
1 =: i
2 =: j
i+j =: k =: m
ENDROUTINE
ENDMODULE
```

The first line declares a module which is the smallest section of a PLANC program that can be compiled separately.

On line 2, a single dimension array with bounds of 0 and 100 is declared as a data-element in the basic module mudpie. Note that the lower index bound must be 0 to be used by the INISTACK standard routine.

Variables local to mprog appear in a declaration statement in line 4. i and j are set to 1 and 2 and their sum is assigned to both k and m, within one expression.

However simple a program may be, the INISTACK standard routine, shown on line 5, must appear in the main program (here mprog) before any other routines are called. It creates a stack to provide storage for dynamic allocation of the data-elements within each routine while it is being executed. In the above example this stack will be called "stack", declared in the main module.

1.3 DATA TYPES

Having looked at a basic PLANC program we will now look in greater detail at the way in which data is described.

PLANC supports a variety of data types which are divided into the categories of simple and composite. A data-element of composite type may be subdivided into simple or further composite types. They are the following:

SIMPLE TYPES	COMPOSITE TYPES
INTEGER	ARRAY
REAL	RECORD
BOOLEAN	SET
LABEL	ROUTINE
VOID	
ENUMERATION	
POINTER	

The PLANC data types ENUMERATION and VOID are unusual; since the type VOID only appears in the declaration of routines it is described along with them. Data type ENUMERATION enables a data-element to take any value from an explicitly specified ordered group. Examples of declarations would be:

```
ENUMERATION (hot,warm,mild,cool,cold) : weather,temperature
ENUMERATION (lousy,firstclass,luxury,deluxe) : hotel :=lousy
```

Note that hotel has been set to an initial value of "lousy" (hopefully our program will be able to improve it!).

POINTERS are data types which are "addresses" of variables of some other type. For instance, we could declare:

```
REAL : r
REAL POINTER : rp := ADDR(r)
```

where the pointer data-element rp is initialised with the address of the REAL data-element r.

Some of the simple data types may have certain characteristics modified. Thus type INTEGER may have its RANGE modified, type REAL its PRECISION modified, and any simple type may be ACCESS modified.

Access modified types are either READ or WRITE modified. If the modification is READ then write operations on the data-element are illegal, ie. the data-element may only take the initial value. Conversely, WRITE modification usually precludes read access. This facility can be useful when a data-element is used as a routine parameter.

There are also some predefined types of data (ie. they can be defined in terms of already existing simple types) for holding sequences of characters (sometimes called character strings) or sequences of binary bits. They are:

1) BYTE : For containing a single character

2) BYTES : For containing character strings

3) BITS : For containing bit strings

1.4 TYPE SPECIFICATION

Just as predefined and modified data types are based on the simple data types, it is also possible in PLANC for the user to define his own data types in terms of any of these three. However, a user type specification differs in that it does not cause a data-element to be constructed. This will only occur on a subsequent declaration statement.

Examples of the use of the TYPE specification are:

```
TYPE personnel_number = INTEGER RANGE (0 : 999999)
TYPE calc = REAL READ
TYPE section = REAL ARRAY POINTER
```

Note that the data type section represents a pointer to an array of reals. Contrast this with:

```
TYPE sparse = REAL POINTER ARRAY
```

where sparse is an array of pointers, each pointing to a single real.

1.5 RECORDS

Using a TYPE specification for the declaration of RECORD data types provides a "structure template" for the components of a record as seen in the example.

EXAMPLE 1.2 A RECORD TYPE SPECIFICATION

```
         TYPE monthnames = ENUMERATION(jan,feb,mar,apr,...,nov,dec)

         TYPE date = RECORD
                       INTEGER RANGE (1:31) : day
                       monthnames : month
                       INTEGER RANGE (0:2000) : year
                     ENDRECORD
     % declare some data-elements of the newly specified data type
         date : startdate,end_date
```

The above record has three components but it could have had any number of them.

Note that as this is an example of a TYPE specification no data-element is constructed unless a declaration statement is encountered such as the last line of the example.

It is possible to define a record which has components in addition to those of an existing one. This variant record will then have the components defined in the base record together with the new components from the variant part.

EXAMPLE 1.3 A VARIANT RECORD

```
          TYPE part = RECORD
                         REAL : partno, buyprice, sellprice
                      ENDRECORD
          TYPE tax_rating = part RECORD
                                   INTEGER : taxcode
                                 ENDRECORD
          TYPE stock = part RECORD
                              BYTES : wharehouse(1:4)
                              REAL : quantity
                            ENDRECORD
      % declare some record data-elements
          part : frame
          tax_rating : boughtin
          stock : screw
```

Thus records of type tax_rating (eg. boughtin) will have components partno, buyprice, sellprice, and taxcode, and records of type stock (eg. screw) will have components partno, buyprice, sellprice, wharehouse and quantity.

To access components of a record a dot notation is used. Thus to access components in the records of example 1.3 we would use references like:

```
          frame.buyprice
          boughtin.taxcode
```

It can be useful to have an empty base record which can serve as a common entry point to the variant ones by using a pointer which references the base record.

1.6 LIST_PROCESSING

List structures can be defined as record structures as illustrated below.

EXAMPLE 1.4 RECORD TYPES IN LIST PROCESSING

```
          TYPE element = RECORD
                            element POINTER : NEXT
     %
     %                      other components
     %
                          ENDRECORD
     % pointer for the start of a linked list of records
          element POINTER : HEAD
```

The pointer HEAD would point to the first element in the list and the pointer NEXT in each record would point to each successive element in a list.

There are 3 standard routines available in PLANC for list processing:

INSERT	will insert a new element at the head of a list
APPEND	will append a new element at the end of a list
REMOVE	removes any element from the list

1.7 SEQUENCE CONTROL STATEMENTS

Control statements enable the normal sequence of statement execution to be altered. PLANC has a number of facilities to form repetitive loops or select a course of action from a number of possibilities.

The FOR and ENDFOR statements create a very simple loop. The code bounded by them must include a DO statement as shown in the example.

EXAMPLE 1.5 A SIMPLE FOR-ENDFOR LOOP

```
        FOR count IN 1:n DO
          count+sum =:sum
        ENDFOR
```

Another simple loop is formed by the DO-ENDDO statements. The structure is:

```
        DO
         statements for execution
        ENDDO
```

Either of the two loops above may contain a WHILE statement. For example:

EXAMPLE 1.6 ANOTHER FOR-ENDFOR LOOP

```
        INTEGER : lower,upper
        INTEGER ARRAY : a(0:10)
        FOR i IN lower:upper DO
          a(i-1)+a(i) =:a(i)
    % continue the loop only for negative array elements
          WHILE a(i)<0
    %
        ENDFOR
```

A simple conditional statement is the IF statement. It must always be followed by a corresponding ENDIF as in:

EXAMPLE 1.7 IF-THEN-ENDIF

```
    % make the value positive
        IF x < 0 THEN
          -x =:x
        ENDIF
```

IF statements may be nested, and there are no restrictions on the executable statements which may be contained in a nested IF statement.

Further, PLANC has a CASE statement. It selects one of a number of a group of statements to be executed, the remaining groups are ignored.

EXAMPLE 1.8 THE CASE STATEMENT

```
         ENUMERATION (stop_signal,go_signal) : action
         TYPE colour_list = ENUMERATION (red,blue,green,amber)
     %
         colour_list : colour
     %
         CASE colour
           INCASE red
             stop_signal =:action
           INCASE green
             go_signal =:action
           ELSE
      %
      % control only comes here for other colours
      %
         ENDCASE
```

Note the percent character (%) indicating a comment line. It may appear in any column of a statement. Everything following the percent character, on the same line, is ignored by the compiler.

1.8 ROUTINES

From a language point of view, routines can be regarded as composite data-elements. When a routine is declared, a data-element is constructed which is sufficiently large to contain all of the storage the routine will require. (Storage required at run time is provided by the INISTACK standard routine, as illustrated in the very first example.)

PLANC routines are similar to the subprograms of other languages but they have an extra feature in that a specific single value can be supplied to the routine by the caller, and vice versa, such that the value input is available anywhere within the routine. These values are in addition to the usual parameters. For example:

EXAMPLE 1.9 A SIMPLE ROUTINE

```
          ROUTINE VOID,VOID (INTEGER WRITE) : simple(intpara)
      %
      % no values supplied into or out of the routine SIMPLE,
      % it has only one integer parameter intpara
      %
          INTEGER : local,int
          FOR local IN 1,2,3,8:10 DO
      %
      % executable statements within the loop
      %
          ENDFOR
      % intpara will be returned to caller
          int=:intpara
          RETURN
          ENDROUTINE
```

The use of the data type VOID is shown, so-named since it indicates the absence of the in-value data-element or the out-value data-element respectively. The routine body contains control statements for a simple repetitive loop.

Only one parameter (within the parentheses following the routine name) will be passed to the routine and it is declared to have WRITE access only. Parameters have by default READ access only. The keyword WRITE allows this parameter to have values stored into it and the actual parameter will not receive this new value before the routine has returned to its caller.

A more sophisticated example of sorting by successive maxima follows. The mechanism used is to find the maximum element of an array which is "swapped" with the first element. The subarray of all elements, except the first, is now scanned and the maximum element will be interchanged with the second of the original array, and so on. Within the routine the standard routine MAXINDEX yields the maximum index (upper bound) of "vector", and the invocation of "highest" obtains the index of the maximum element of each subarray. (The routine "highest" is in fact given as example 1.12.)

EXAMPLE 1.10 SORTING BY SUCCESSIVE MAXIMA

```
ROUTINE VOID,VOID (REAL ARRAY READ WRITE) : sort(vector)
REAL : temp
INTEGER : k,highval
  FOR k IN vector DO
    highest(vector(k : MAXINDEX(vector,1))) =:highval
    vector(highval) =:temp; vector(k) =:vector(highval)
    temp =:vector(k)
  ENDFOR
ENDROUTINE
```

The next example returns an out-value, ie. it is like a Fortran function reference, which indicates whether an array contains all the same values or not. The out-value is declared as BOOLEAN in the routine declaration so that a value of TRUE or FALSE can be returned. In this case it depends on whether or not all the values of an integer array are unequal.

EXAMPLE 1.11 ROUTINE WITH AN OUT-VALUE

```
    ROUTINE VOID,BOOLEAN (INTEGER ARRAY) : func(arrx)
%
% no in-value, out-value BOOLEAN, in the routine func
% having 1 parameter, arrx, an INTEGER array
%
    INTEGER : i,j
% loop through all the elements of the array
    FOR i IN arrx DO
% loop through each element prior to this element of the array
      FOR j IN 1:i-1 DO
% is there a different value ?
        IF arrx(i) >< arrx(j) THEN
% all array elements not the same value
          FALSE RETURN
        ENDIF
      ENDFOR
    ENDFOR
% all elements are the same value
    TRUE RETURN
    ENDROUTINE
```

The routine "highest", invoked in example 1.10, is a further example we can give at this point. It returns the index of the maximum value in an array (MININDEX obtains the value of the lower bound).

EXAMPLE 1.12 ANOTHER ROUTINE WITH AN OUT-VALUE

```
    ROUTINE VOID,INTEGER(REAL ARRAY) : highest(v)
    REAL : max
    INTEGER : answer,i
% set an initial index of the highest value
      v( MININDEX(v,1)=:answer ) =:max
% scan the array for the highest value
      FOR i IN v DO
        IF v(i) > max THEN
% note the use of the resulting value as the subscript
          v( i=:answer ) =:max
        ENDIF
      ENDFOR
% give back the index of the highest value as an out-value
      answer RETURN
    ENDROUTINE
```

In the case where there is an in-value, this can be referenced within the routine by use of the @ (commercial at) character. If the routine has an in-value but no out-value it will simply store the in-value it receives; the in-value will be the data-element associated with the identifier referred to <u>immediately</u> preceding the routine invocation. Example 1.13 shows some of the principles involved.

EXAMPLE 1.13 ROUTINE WITH AN IN-VALUE BUT NO OUT-VALUE

```
    INTEGER : param2,prog_data_el  ; REAL : param1
% set up the in-value
    INTEGER : inval
    2=:inval
% invoke the routine rtn with inval as the in-value
    inval rtn(param1,param2)
% after the routine call, parm2 will have the value 1
% it could be assigned to a program data-element
    param2=:prog_data_el
```

The routine declaration might be

```
    ROUTINE INTEGER,VOID (REAL,INTEGER WRITE) : rtn (p1,p2)
%
% and the routine body might contain
%
    IF @>0 THEN        % reference in-value
      1=:p2
    ENDIF
%
    RETURN
    ENDROUTINE
```

A routine of this type might be used in situations such as reading or writing to/from files or similar service functions, thus saving the programmer some coding.

Finally, the routine with both an in-value and an out-value. As an example, the routine below adds two complex numbers represented as records.

EXAMPLE 1.14 ROUTINE WITH BOTH IN AND OUT-VALUES

```
% a record type specification of a complex number
   TYPE complex = RECORD
     REAL : r,i
   ENDRECORD
%
% a routine to perform addition of two complex numbers
%
   ROUTINE complex,complex (complex) : plus(cnum)
% the out-value is declared as complex
   complex : result
% the in-value, referenced by @, is one complex number,
% the parameter is the other
   @.r + cnum.r=::result.r
   @.i + cnum.i=::result.i
% put the sum of the two complex numbers into the out-value
   result RETURN
   ENDROUTINE
```

The routine plus could be invoked by

```
c1 plus c2
```

where c1 and c2 have been declared as:

```
complex :c1,c2
```

Since routine identifiers can be a string of letters or special characters, the routine name might equally well have been + or *+, and the invocation:

```
c1 + c2   or
c1 *+ c2
```

thus the routine defines a user-written operator.

1.9 MODULES

A module, which is the smallest unit of a PLANC program which can be compiled separately, can be contained within other modules. Thus we can have basic modules and any number of compound ones. All program and data must be inside a basic module and in addition, if it is to be independently executable, it must contain a main program, as shown in example 1.1. However, only one main program routine can exist per executable program since it is this which defines the execution-time entry point.

Large programs are usually subdivided into logical groups, ie. modules, to simplify their administration. Access from one module to the data and routines of another is controlled by the two PLANC statements: EXPORT and IMPORT.

An IMPORT statement defines items of other modules to be accessible in the present module. An EXPORT statement defines items in the present module to be accessible to other modules. In the example below we show the structure of a compound module which contains two basic modules, together with a simple usage of the IMPORT and EXPORT statements.

EXAMPLE 1.15 MODULE STRUCTURE

```
        MODULE comp                   % Compound module
          MODULE basic1               % Basic module
          EXPORT x
          IMPORT REAL : y
          INTEGER : x
    %
    %
    %
          ENDMODULE                   % End of module basic1
    %
          MODULE basic2               % Another basic module
          EXPORT y
          IMPORT INTEGER : x
          REAL : y
    %
    %
    %
          ENDMODULE                   % End of module basic2
        ENDMODULE                     % End of compound module comp
```

1.10 SCOPE OF IDENTIFIERS

An identifier has a scope which is the routine, or module, which contains its declaration and all the routines, or modules, within it. For example:

EXAMPLE 1.16 SCOPE OF IDENTIFIERS

```
    MODULE update
% global variables
    BOOLEAN ARRAY : busy(0:100)
%
    ROUTINE VOID,INTEGER : reserve
    INTEGER : i,j
      FOR i IN busy DO
%
      ENDFOR
      j RETURN
    ENDROUTINE
%
    ROUTINE INTEGER,VOID : release
    INTEGER : i,j
%
    ENDROUTINE
    ENDMODULE
```

The array busy has the scope of module update, and is also known by the routines reserve and release. The variables i, j in reserve are different from the i, j in release.

1.11 SIMPLE INPUT/OUTPUT TO THE TERMINAL

PLANC has no extensive facilities for handling input and output. However, there are some system-supplied routines to handle the simple case. As an example, the statement :

```
         INPUT (1,'I5',number)
```

will read an integer from the terminal and place it in number.

For output, the statement :

```
         OUTPUT (1,'I5',number)
```

will write number as an integer using 5 places on the output line.

We can now write a PLANC program to read 2 numbers from the terminal, and write out their sum.

EXAMPLE 1.17 SIMPLE I/O USING THE TERMINAL

```
         MODULE summer
         INTEGER ARRAY : stack(0:100)
     % a main 'PROGRAM' routine follows
           PROGRAM : sum
           INTEGER : a,b,c
           INISTACK : stack
     % get two numbers from the terminal
           INPUT (1,'I5',a)
           INPUT (1,'I5',b)
     % output the sum of the two numbers on the terminal
           a+b=:c
           OUTPUT (1,'I5',c)
           ENDROUTINE        % end of routine 'sum'
         ENDMODULE
```

1.12 A MORE COMPLEX EXAMPLE

So that we can see how some of the previously mentioned features might be combined, we give a final example. Suppose it is required to find the area of a farm where each field is represented by a record in a linked list of records. In the given code these records are chained together through the record component data-element next.

EXAMPLE 1.18 DINKUM PLANC

```
% specify a RECORD data type for each field of the farm
    TYPE field = RECORD
      REAL : area
      field POINTER : next
    ENDRECORD
%
% a pointer data-element to begin a linked list
% - see later chapters for details of building the list
%
    field POINTER : pepfarm
% a data-element for the area of the farm
    REAL : farmsize
% invoke the routine to compute the total farm area
    acreage(pepfarm)=:farmsize

    ROUTINE VOID,REAL (field POINTER) : acreage(first)
    field POINTER : work
    REAL : answer
      0.0 =:answer
% scan the list of field records to compute the total area
      FOR work IN first:next DO
        answer + work.area =:answer
      ENDFOR
      answer RETURN
    ENDROUTINE
```

The FOR-ENDFOR loop contains an "pointer implied range" first:next which describes a linked list of pointers. The data-element before the colon is a record pointer indicating the start of the chain. Following the colon is the data-element within the record which contains the linking pointers through the chain. In this way we can access a linked list of records using a simple FOR-ENDFOR loop, a useful facility when processing lists.

2 BASIC_LANGUAGE_ELEMENTS

2.1 INTRODUCTION

Following the overview of the PLANC language as a whole we will now begin to look at the language features in complete detail.

This chapter will present the lowest level language elements; such as the character set, identifiers and literals; with which PLANC source language statements can be formed. A number of source statements can then be put together to construct a complete PLANC program. This program can be submitted to the PLANC compiler to produce an executable program if the compilation process is successful.

2.2 THE_CHARACTER_SET

The full ASCII character set may be used in PLANC programs. However particular elements of the language may be made up of a restricted subset of characters as indicated in the following sections. Lower case alphabetic characters are converted to upper case except when used in string literals.

2.3 STANDARD SYMBOLS

The Standard Symbols have predefined meanings in the PLANC language. They are special characters or are formed from special characters and letters. Standard Symbols comprising alphabetic characters only are often referred to as keywords. A list of all the Standard Symbols follows :

Special characters

% - treat the rest of this line as comment text.
& - the statement on this line is continued on the next line.
; - terminate the preceding language statement on this line.

Note : this is used to put more than 1 statement on a line.

' - single apostrophe, is used to delimit a string literal.
- precedes a single character literal.
(- opening parenthesis.
) - closing parenthesis.
: - delimiter in declaration statement or range expression.
, - delimiter in a list of identifiers.
@ - routine in-value qualifier.
. - dot notation for accessing record components.
? - predeclaration indicator.
$ - as first character indicates line is a compiler command.
" - enclose macro parameters within the macro definition.

Keywords

ALIAS
ARRAY
ASSERT
ASSERTFALSE
BITS
BOOLEAN
BYTE
BYTES
CASE
COMMON
CONSTANT
DO
ELSE
ELSIF
ENDCASE
ENDDO
ENDFOR
ENDIF
ENDRECORD
ENDROUTINE
ENDMODULE
ENDON
ENUMERATION
ERRCODE
ERRETURN
EXITFOR
EXITWHILE
EXPORT
FALSE
FOR
GO
IF
IMPORT
INCASE
INLINE
INTEGER
INTEGER1
INTEGER2
INTEGER4
LABEL
MODULE
NIL
ON
OVERFLOW
PACKED
POINTER
POINTERERROR
PRECISION
PROGRAM
RANGE
RANGEERROR
READ
REAL
REAL8
RECORD
REFERENCE
RETURN
REVERSE
ROUTINE
ROUTINEERROR
SET
SPECIAL
STANDARD
STACKERROR
SYSTEM
THEN
TRUE
TYPE
VOID
WHILE
WRITE

Operators

=:	- assignment
:=:	- change
+	- addition
-	- subtraction (binary operator), negation (unary operator)
*	- multiplication
/	- division
**	- exponentation
ABS	- absolute value or maximum number of SET members
MOD	- modulo
SHIFT	- shift bits

>	- greater than
<	- less than
=	- equivalent value
>=	- greater than or equal
<=	- less than or equal
><	- not equal
IN	- membership

AND	- logical and
OR	- inclusive or
XOR	- exclusive or
NOT	- logical negation
=	- assignment in CONSTANT statement, storage equivalence and identifier data type in TYPE specifications
:=	- initial value in declaration statements

Standard Routines

ADDR
APPEND
BIT
BLOCKSIZE
CLOSE
CONVERT

DISPOSE
FILESIZE
FORCE
IND
INISTACK
INSERT

INPUT
MAXINDEX
MININDEX
NEW
OPEN
OUTPUT

PRED
REMOVE
SIZE
SUCC
TYPEOF

2.4 STATEMENTS

PLANC statements are usually written one per line. A statement may be terminated by a semicolon character (;) but this is not required. However more than one statement may be included on one line by using the semicolon character (;) to terminate each statement within the line. All alphabetic characters in PLANC statements may be typed in lower or upper case but the compiler will convert all the alphabetic characters to upper case with the exception of single character literals and string literals, including format descriptors in INPUT/OUTPUT statements ie. anything between single apostrophes. For clarity it is suggested that all keywords are typed in upper case. A single blank must be present immediately before and after most keywords, but more blanks are not treated as significant by the compiler. Some keywords may be preceded or followed by operators or delimiters. While PLANC has a free format, it is recommended that blanks be used generously to indent and space source code elements for clarity and readability.

For example :

```
         INTEGER : int1,int2 ; REAL : rl1 ; BOOLEAN : bool1
```

2.5 CONTINUATION OF STATEMENTS

Sometimes it may be necessary to write a statement which is longer than one line. If a statement is to be continued on the next line, an ampersand character (&) must be placed after the statement text on the first line, and the compiler will append the next line to the first line and treat both lines together as a single language statement.

For example :

```
         INTEGER : int1,int2,int3,  &   % this line will be continued
                   int4,int5
```

2.6 COMMENTS

Comments within program source, are important for documentation purposes and they may be included on any lines of PLANC source by inserting a percent character (%). All text following the percent character (%) on the same line will be regarded as comment text by the compiler and have no effect on the program.

For example :

```
         INTEGER : integ1,integ2
    %
    % The line above, this line and the following 2 lines
    %    are comment lines. They have no effect on the program.
    %
         INTEGER : integ3
         INTEGER : integ4        % This is also comment text ! aha !
```

Note that there is a special use of two consecutive percent characters (%%), see section 2.10.

For example :

```
    %%%%  this is not a comment line
    % %%  but this is
```

2.7 LITERALS

A literal is an integer, real, boolean, character or string constant. Literals do not change their value during the execution of the program. A literal value is held in a storage entity known as a data-element.

2.7.1 INTEGER_LITERALS

The form of an integer literal is an optional minus sign followed by a string of digits.

Examples of integer literals :

```
0
123
-1
123456
```

The maximum and minimum possible values and the actual size of the data-element used to store the integer literal is machine-dependent. In general the smallest data-element possible to contain the actual value will be allocated by the compiler.

For example on the ND-100 the values must lie between :

```
-2147483648  and  2147483647 inclusive,
351          will be stored in an INTEGER2 data-element.
```

For full details of limits on possible range of values and actual storage allocated, see Appendix C.

An integer literal in PLANC may be written as an octal value rather than as a decimal value. An octal literal is an optional minus sign followed by a string of digits, each in the range 0 to 7 inclusive, and followed by the letter B.

Examples of octal integer literals :

```
0B
777B
-765B
```

The range of values possible and the storage allocated by the compiler will be the same as for decimal literals. For example :

on the ND-100

```
537B      (351 decimal) will be stored in an INTEGER2
          data-element.
```

2.7.2 REAL_LITERALS

The form of a basic real literal is an optional minus sign, a whole number part, a decimal point and a fractional part. Both the whole number part and the fractional part are strings of digits; the whole number part must be present.

A real exponent consists of the letter E followed by an unsigned whole number for a positive exponent or a minus sign and a whole number for a negative exponent. The value of a real literal containing an exponent is the product of the basic real literal preceding the E and the power of 10 indicated by the number following the E. The exponent must not be preceded by a space.

Examples of some valid real literals :

0.0	
11.	
3.1415927	
-728.998	
-98765.0	
1.23E2	exponent form of a real literal
1.32E-4	real literal with a negative exponent

Examples of some invalid real literals :

12	a valid integer but no decimal point
.0	no digit preceding the decimal point
+1.2	must not be preceded by a + sign
1.5E2.5	exponent must be a whole number
1.6E+2	exponent must not have a + sign

The real value is an approximation to the actual value of a mathematical expression. The actual internal representation of real values may not be the same in all implementations of PLANC. The maximum and minimum real values possible may vary on different model machines or according to the type of floating-point hardware on a particular machine. Further, the number of significant digits which may be represented accurately also depends on the machine model and the floating-point hardware present. Full details of storage allocation, maximum and minimum possible values, and the number of significant digits which can be represented accurately are available in Appendix C.

2.7.3 BOOLEAN_LITERALS

The possible values of a boolean literal are TRUE or FALSE.

Examples of boolean literals :

TRUE
FALSE

2.8 LITERAL EXPRESSIONS

A literal expression is an expression made up of either literals of the same data type or identifiers which have already been declared in a CONSTANT statement, thus having a literal value. For a detailed description of the way expressions are evaluated, see Chapter 5 , EXPRESSIONS - FORMATION AND EVALUATION. In addition to the operators listed below for each data type, parentheses may be used for clarity or to force an expression to be evaluated in a particular order of operations.

2.8.1 INTEGER LITERAL EXPRESSIONS

Integer literal expressions may be formed by using integer data-elements and the following operators and standard routines :

+	arithmetic plus
-	arithmetic minus
-	unary minus
*	arithmetic multiplication
/	arithmetic division
**	exponentiation
MOD	modulo
ABS	absolute value
SHIFT	shift bits
NOT	logical complement
AND	logical 'and'
OR	logical 'inclusive or'
XOR	logical 'exclusive or'
MININDEX	array index lower bound
MAXINDEX	array index upper bound
SIZE	data-element size

For example :

```
         INTEGER : int1:=2*2            % integer literals only
     % the indentifier int1 will be initialised to 4 .

         CONSTANT four=4
         INTEGER : int2:=(1+four)*2     % literals, constants mixed
     % the identifier int2 will be initialised to 10 .

         INTEGER : int3:=777B AND 17B  % use of logical operator
     % the identifier int3 will be initialised to 17B
     % ie. 15 decimal
```

2.8.2 REAL_LITERAL_EXPRESSIONS

Real literal expressions may be formed by using real data-elements and the following operators :

+	arithmetic plus
-	arithmetic minus
-	unary minus
*	arithmetic multiplication
/	arithmetic division
ABS	absolute value

For example :

```
        REAL : rl1:=2.5*4.0            % real literals only
    % the identifier rl1 will be initialised to 10.0 .

        CONSTANT rlconst=2.0
        REAL : rl2:=(5.7-rlconst)/2.0 % literals, constants mixed
    % the identifier rl2 will be initialised to 1.85 .
```

2.8.3 BOOLEAN_LITERAL_EXPRESSIONS

Boolean literal expressions may be formed by using boolean data-elements and the following operators :

NOT	logical negation
AND	logical 'and'
OR	logical 'inclusive or'
XOR	logical 'exclusive or'

Further, boolean literal expressions may contain any of the relational operators (see section 5.4) with integer operands only.

For example :

```
        BOOLEAN : bool1:=TRUE AND FALSE  % literals only
    % the identifier bool1 will be initialised to FALSE .

        CONSTANT bc1=TRUE
        BOOLEAN : bool2:=bc1 OR TRUE  % literals, constants mixed
    % the identifier bool2 will be initialised to TRUE .

        BOOLEAN : bool3:=TRUE AND (2=3)
    % the boolean expression in parenthesis results in FALSE
    % and the identifier bool3 will be initialised to FALSE .
```

2.9 SINGLE_CHARACTER_LITERALS

The form of a single character literal is the number sign character (#) followed by one ASCII character. For example :

#a	value is lower case 'a'
#Z	value is upper case 'Z'
#(	value is left parenthesis

PLANC has no 'character' data type. A single character literal will be held in a data-element of the predefined data type BYTE (see section 3.12.1). With certain choices of data storage allocation, this enables much faster handling of a single character than a character string of length greater than one character.

Note that to specify the special characters percent (%),ampersand (&) and apostrophe (') in a single character literal, only one occurrence of such a character should follow the number sign character(#).

2.10 STRING_LITERALS

The form of a string literal is the apostrophe character ('), followed by one or more ASCII characters, terminated by another apostrophe character (').

For example :

```
'this is a STRING of characters'
```

PLANC has no string data type. String literals will be held in a data-element of the predefined data type BYTES (see section 4.1.7.1).

Upper case alphabetic characters within string literals will not be converted to lower case.

Note : if % (percent), & (ampersand), or ' (apostrophe) characters are to appear within a string literal then these characters must be duplicated for each occurrence required, in order to prevent their usual 'special' interpretation in PLANC. For example :

String Literal	value
'his && hers'	his & hers
'two %%%% characters'	two %% characters
'Tom''s 5 %% share'	Tom's 5 % share
''''	' (one apostrophe)

Note that 'a' is not equivalent to #a and has a different internal representation.

2.11 IDENTIFIER_NAMES

An identifier in PLANC is the name associated with a data-element. An identifier is a sequence of letters, digits and underscore characters, but the first character of which must be a letter. An underscore must not be last character of an identifier and only single underscore characters may be used, ie. two consecutive underscore characters are invalid. While an identifier may be of any length, only the first ten characters are used as for unique identification. For example :

integ1	
counter_VARIABLE	
a_b_c	
5abc	invalid, does not begin with a letter
in-valid	invalid, contains an illegal character, a hyphen (-)
abc_	invalid, ends with an underscore
a__b	invalid, two consecutive underscores

Since uppercase and lowercase letters are treated as equivalent by the compiler, the identifiers :

ident1 and
IDENT1

will be associated with the same data-element.

As only the first ten characters of identifier names are significant, the identifiers :

a_very_long_name and
a_very_long_identifier

will be associated with the same data-element.

2.12 ENUMERATION_LITERAL_LISTS

The form of an enumeration literal list is a list of enumeration identifiers separated by commas. The general form is :

```
        enum-ident[,enum-ident ...]
```

where

enum-ident   is formed under the same rules as identifiers

The order of appearance in the list specifies the sequence of the enumeration identifier values for use as operands with the relational operators (see section 5.4) or with the PRED and SUCC standard routines (see section 7.9) which will return previous or successive values respectively.

For example :

```
        red,dark_blue,green,purple
```

is a valid enumeration literal list with four enumeration identifiers.

2.13 IMPLIED RANGE

The implied range is an abbreviated form for describing all or part of a list of Integer values, Enumeration identifiers or Pointer data-elements. The precise meaning of such a list depends on which PLANC statement it is used in. It has the following general form :

```
         value1 : value2              or
         expn1 : expn2                or
         ptr1 : ptr2
```

where

value1, value2 are both, either integer literals, enumeration identifiers or the resulting value of literal expressions of these data types.

expn1, expn2 are expressions which will be evaluated at run-time to give an integer or enumeration resulting value.

Note : in both the above cases the second value must be greater than or equal to the first value or a list with no values will be generated.

ptr1, ptr2 are pointer identifiers within a linked list of record data-elements, or a linked list of pointer data-elements.

Examples of implied ranges :

```
         12 : 36
      % specifies the list of integer values
      %      12, 13, 14, ... , 35, 36

         2*(3+1) : 10**2
      % specifies the list of integer values
      %      8, 9, 10, . . . , 99, 100

         ENUMERATION (white,black,red,blue,grey,green,mauve)
      % followed by a statement containing
         red : green
      % specifies the enumeration literal list, ie. enumeration
      % identifiers
      %      red, blue, grey, green
```

The implied pointer range is discussed in more detail in section 3.8, together with the description of the Pointer data type. For examples of the use of an implied pointer range, see FOR - ENDFOR loops, section 6.5, and Processing of Records in List Structures section 4.6.

3 DATA DECLARATION AND SIMPLE DATA TYPES

This chapter will describe some of the basic terms and concepts associated with the storage and accessing of data values in PLANC programs. Only the simple data types will be discussed here. More complex data structures are available in PLANC, eg. arrays and records, but they will be discussed later.

3.1 DEEINITION OE ELANC TERMINOLOGY

Amongst the basic language elements of PLANC, literals and identifiers have already been discussed (Chapter 2, BASIC LANGUAGE ELEMENTS). A data-element is any area of storage that can be referred to as an entity and may contain a definite value. Most data-elements are referred to by an identifier name but some, such as literals do not have any associated name. Each data-element is of a defined data type which specifies two characteristics :

1) the format and range of possible values of information stored in the data-element.

2) the operations which may be applied to the data-element.

Data-elements may be of either a simple or a composite data type. A data-element of a simple data type is an entity which may not be split into any components. A data-element of a composite data type consists of components, each of which is a data-element of simple or composite type.

The PLANC language has a variety of data types available.

Simple data types are :

1) INTEGER

2) REAL

3) BOOLEAN

4) LABEL

5) VOID

6) ENUMERATION

7) POINTER

Composite data types are :

1) ARRAY

2) RECORD

3) SET

4) ROUTINE

Some simple data types may have particular characteristics modified. The modifications which are available are :

1) RANGE - for INTEGER type only

2) PRECISION - for REAL type only

3) ACCESS MODIFIED - for some simple and composite data types

In a PLANC program a new data type may be created by defining the new type in terms of existing data types. The existing simple data type used in such a definition is called the base type of the new data type.

A declaration specifies an identifier name to be associated with a data-element, the data type of the data-element and allocates appropriate storage to contain the values of the data-element. A declaration may also optionally specify an initial value to be present in the data-element when the program begins execution. The general form of a declaration statement for a simple data type is :

```
data-type : ident[:=lit-exp] [,ident[:=lit-exp]  ] ...
```

where

data-type   is a valid simple data type

ident   is a valid identifier

lit-exp   is a literal expression of appropriate type

Note : initial value is valid only for INTEGER, REAL, BOOLEAN types.

An initial value should normally be used in the outer level of a module. If an identifier is to have an initial value inside a routine, then its access must be declared as READ, see section 3.11.3.

3.2 INTEGER DATA-ELEMENTS

The data type 'integer' specifies data-elements which can contain whole number values. The general form of a declaration of an integer data-elements is :

```
INTEGER : ident[:=lit-exp] [,ident[:=lit-exp]  ] ...
```

where

ident is a valid identifier

lit-exp is a integer literal expression

The range of possible values which can be held in an integer data-element has been discussed briefly under Integer Literals, see section 2.7.1 . For full details of the range of possible values and storage allocated, see Appendix C.

Some variants of the INTEGER type are available and these have particular range limits. These are :

1) INTEGER1 - to be stored in an 8 bit field. The range of possible values is :
   -128 <= value <= 127 .

2) INTEGER2 - to be stored in a 16 bit field. The range of possible values is :
   -32768 <= value <= 32767 .

3) INTEGER4 - to be stored in a 32 bit field. The range of possible values is :
   -2147483648 <= value <= 2147483647 .

The type INTEGER will default to one of the variants depending on the machine implementation, see Appendix C.

During compilation, the initial value of an integer literal data-element, will not cause a compiler error if it is too large for the storage available for the data type declared; some form of truncation will occur. During program execution no checks will be carried out other than those provided by the hardware being used, eg. overflow, see Exception and Error Handling, section 6.8 and Appendix C.

Examples of integer declarations :

```
INTEGER : int1,int2,int3,init1:=45,int4
INTEGER1 : int8b:= -22
INTEGER2 : int16b
INTEGER4 : int32b
```

3.3 REAL DATA-ELEMENTS

The data type 'real' specifies data-elements which can contain floating-point values. The general form of a declaration of real data-elements is :

```
REAL : ident[:=lit-exp] [,ident[:=lit-exp]  ] ...
```

where

ident   is a valid identifier

lit-exp   is a real literal expression

The range of possible values which can be held in a real data-element has been discussed briefly under Real Literals, see section 2.7.2 . For full details of the range of possible values, the number of significant digits and storage allocated, see Appendix C.

A variant of the REAL type is available and it has particular range limits. These are :

1) REAL8 - to be stored in a 64 bit field. The range of possible values is :

   10**-76 <= value <= 10**76
   with accuracy of 15 significant digits.

The type REAL will default to a 32, 48 or 64 bit format depending on the machine implementation and the floating-point hardware being used, see Appendix C.

During compilation, the initial value of a real literal data-element, will not cause a compiler error if the value is too large for the storage available for the data type declared; some form of truncation will occur. During program execution no checks will be carried out other than those provided by the hardware being used, eg. overflow, see Exception and Error Handling, section 6.8 and Appendix C.

Examples of real declarations :

```
REAL  : rl1,rl2,rinit1:=45.0,rinit2:=2.65E-8,rl3
REAL8 : rl64bit
REAL8 : rl64b:= -22.765E24
```

3.4 BOOLEAN DATA-ELEMENTS

The data type 'boolean' specifies data-elements which can contain logical values. The general form of a declaration of boolean data-elements is :

```
          BOOLEAN : ident[:=lit-exp] [,ident[:=lit-exp]  ] ...
```

where

ident   is a valid identifier

lit-exp   is a boolean literal expression

The possible values which can be held in a boolean data-element are TRUE or FALSE. They have been discussed briefly under Boolean Literals, see section 2.7.3 .

Examples of boolean declarations :

```
          BOOLEAN : bool1,bool2,bool3
          BOOLEAN : blinit1:=TRUE,blinit2:=FALSE AND TRUE
```

3.5 CONSTANT DECLARATIONS

The 'constant' declaration specifies identifiers which will be associated with data-elements whose value will be fixed at compile time and not allowed to change during program execution. The general form of a constant declaration is :

```
CONSTANT ident[=lit-exp] [,ident=lit-exp] ...
```

where

ident is a valid identifier

lit-exp is a literal expression of integer, real, boolean type

The following rules apply to CONSTANT declarations :

1) The data type of an identifier is determined by the data type of the corresponding literal expression following the equals character (=).

2) If the equals character (=) and the literal expression are omitted, then the identifier type will be of type integer by default. In this case the integer value stored in the data-element will be the next integer value higher than the previous integer value in this CONSTANT statement. If there is no previous integer value specified in this CONSTANT statement, either explicitly or by default, then 0 will be the first value provided.

Examples of constant declarations :

```
   CONSTANT int1=23,rl1=3.14,bl1=TRUE
% explicit value data types

   CONSTANT zero,rl2=1.1,one,bl2=FALSE,two
% identifiers without values take values 0, 1, 2

   CONSTANT four=4,five,nine=four+five
% 'five' takes the next higher value after 4
% and 'nine' is the sum of 4 and 5

   CONSTANT rl3=rl1*rl2,bl3=bl1 AND bl2
% expressions result in rl3 taking the value 3.454
% and bl3 taking the value FALSE.
```

3.6 ENUMERATION DATA-ELEMENTS

The data type 'enumeration' specifies data-elements which can take any one of a finite number of values declared in an enumeration literal list. The general form of a declaration of enumeration data-elements is :

```
ENUMERATION ( en-lit-list ) : ident[:=en-id-val]
                               [,ident[:=en-id-val] ] ...
```

where

en-lit-list   is an enumeration literal list

ident   is a valid identifier

en-id-val   is one of the values in the enumeration literal list

The possible values which can be held in an enumeration data-element are strictly limited to those values in the enumeration literal list of this declaration statement. An enumeration data-element will usually be held in an integer size storage location which will determine the maximum number of distinct values in the enumeration literal list, for details see Appendix C.

Examples of enumeration declarations :

```
ENUMERATION (saturday,sunday) : weekend_days,days
ENUMERATION (ringnes,becks,fosters) : goodbeer:=ringnes
ENUMERATION (ringnes,mack,fosters) : bestbeer:=fosters
```

The enumeration data type is of particular interest when used in conjunction with the CASE statement, see section 6.3 .

The SUCC standard routine and the PRED standard routine may be used to obtain the following or previous enumeration values respectively. For detailed description of these standard routines see section 7.9 .

3.7 POINTERS

The data type 'pointer' specifies data-elements which can contain references (addresses) to any data-element of a given data type. The given data type for which a pointer identifier can hold references is called the 'qualification' of the pointer. The general form of a declaration of pointer data-elements is :

```
         d-type POINTER : ident[:=p-ident] [,ident:=p-ident ...]
```

where

d-type   is any valid data type

ident   is any valid identifier

p-ident   is any identifier of 'd-type' data type whose reference is to be stored in the pointer data-element initially.

The value 'NIL' may be used to specify that a pointer identifier should reference no data-element. This may be used as an initial value or anywhere within the executable statements to reset the value of a pointer data-element.

Examples of pointer declarations :

```
         INTEGER : int1,int2
         INTEGER POINTER : intptr1,intptr2:=int2
         REAL POINTER : rlptr1,rlptr2:=NIL
```

The possible values of a pointer data-element will vary according to the data type which is to be referenced. Details of storage requirements of pointer data-elements for various data types may be found in Appendix C.

Pointer data-elements may be initialised at compile time by using the ADDR standard routine, providing the parameter of the standard routine invocation can be evaluated by the compiler.

For example :

```
         INTEGER POINTER : ip1:=int    % has the same effect as
         INTEGER POINTER : ip1:=ADDR(int)
     %
         INTEGER POINTER : ipt10:=ADDR(10)
```

will initialise the data-element with the address of the integer constant 10.

Pointer identifiers may be used in expressions with all of the relational operators, eg. to compare addresses for equality in a conditional statement. However it should be noted that evaluation of such expressions and the resulting value depend critically on the internal representation of addresses in each machine implementation of PLANC, see Appendix C.

Pointer data-elements used as operands for the relational operators are treated as unsigned integers for the purposes of comparison. For the size of these integers on each particular machine implementation see Appendix C.

The data-elements described so far are all static in that the necessary memory is allocated for a data-element at the time that the module containing the declaration is about to begin execution. It is also possible to use dynamic data-elements which are created and destroyed dynamically during the execution of the module. The standard routines NEW and DISPOSE may be used for dynamically creating and destroying data-elements respectively, see section 4.5. The POINTER data type may be used to refer to either static or dynamically created data-elements. Dynamically created data-elements do not have explicit identifiers with which to access their values as do static data-elements, so the standard routine IND (see section 7.9) may be used to access the value of dynamically created data-elements.

3.8 POINTER_IMPLIED_RANGE

The pointer implied range is an abbreviated form which describes a linked list of pointer data-elements which may form a chain of records. The syntax of the pointer implied range has been described in section 2.13 . A linked list of records may be set up statically or created dynamically using the NEW standard routine.

The list of data-elements which such a pointer implied range implies, may be created at compile time or dynamically at run-time when the appropriate addresses must be set up by the program. A list being processed by the use of a implied pointer range will terminate when a NIL pointer value is encountered. See Records and List Processing, section 4.6, the IN operator, section 5.4, and FOR - ENDFOR loops, section 6.5, for examples of the use of pointer implied ranges.

3.9 LABELS

The data type 'label' defines an identifier which has no associated data-element. A label identifier may only be placed at the start of an executable statement. The general form is :

```
label-ident : executable-statement
```

Labels must be declared if they are to be referred to by GO statements, see section 6.1. Labels will be further discussed in Scope of Identifiers, see section 7.8.

Examples of label declarations :

```
LABEL : lab1,loop,next
```

3.10 VOID

The data type 'void' denotes the absence of a data-element where a data-element could be present in a statement. The general form of a void declaration is :

```
VOID
```

It has particular use in routine declarations and will be discussed in more detail in Chapter 7, ROUTINES.

3.11 MODIFIED DATA TYPES

A 'modified data type' is one of the simple or complex data types with certain of its characteristics restricted. The following modifications of simple types are available :

1) Range Modification - for INTEGER data types only.

2) Precision Modification - for REAL data types only.

3) Access Modification - read/write access to data-elements of all simple data types.

3.11.1 RANGE MODIFICATION

A 'range modified' integer data-element has its value range restricted to an explicit upper and lower bound. The general form of a range modified integer declaration is :

```
INTEGER RANGE (int-lit-exp : int-lit-exp) :
      ident[:=int-lit-exp] [,ident[:=int-lit-exp] ]...
```

where

int-lit-exp   is a valid integer literal expression

ident   is a valid identifier

The data-elements of a range modified integer data type will have storage allocated as the smallest number of storage units able to hold all values of the range explicitly declared.

Examples of range modified integer declarations :

```
     INTEGER RANGE (-10:990000) : dblint1,dblint2:=99999
  % will require 32 bit integer data-elements

     INTEGER RANGE (0:200) : int1,int2:=148
  % will require data-elements of at least 8 bits
```

During compilation of a program, the size of an integer literal, used for an initial value of a range modified integer data-element, will not cause a compiler error if the value is too large for the storage available for the data type declared; some form of truncation will occur. During program execution no checks will be carried out other than those provided by the hardware being used, eg. overflow, see Exception and Error Handling, section 6.8 and Appendix C.

3.11.2 PRECISION MODIFICATION

A 'precision modified' real data-element has its maximum number of significant digits explicitly specified. The general form of a precision modified real declaration is :

```
         REAL PRECISION (int-lit) : ident[:=real-lit-exp]
                                       [,ident[:=real-lit-exp] ] ...
```

where

int-lit — is an integer literal less than or equal to a number determined by the machine and the floating-point hardware being used.

ident — is a valid identifier

real-lit-exp — is a real literal expression

The data-elements of a precision modified real data type will have storage allocated as the smallest number of storage units able to give the required number of significant digits.

Examples of precision modified real declarations :

```
         REAL PRECISION (4) : rl1,rl2:=99.99
     % will require 32 bit real data-elements

         REAL PRECISION (8) : rl3,rl4:=919.99129
     % will require 48 bit real data-elements
```

During compilation of a program, the size of a real literal, used for an initial value of a precision modified real data-element, will not cause a compiler error if the value is too large for the storage available for the data type declared; some form of truncation will occur. During program execution no checks will be carried out other than those provided by the hardware being used, eg. overflow, see Exception and Error Handling section 6.8, and Appendix C.

3.11.3 ACCESS MODIFICATION

An 'access modified' data-element may have its access restricted to either READ or WRITE operations respectively. The general form of an access modified declaration is :

```
         data-type READ : ident:=lit-exp[,ident:=lit-exp] ...
          or
         data-type WRITE : ident[,ident] ...
```

where

data-type	is a simple data type
ident	is a valid identifier
lit-exp	is a literal expression resulting in a value of 'data-type'

READ access will not allow the value of a data-element to be changed during program execution so it is necessary to initialise such identifiers in a declaration statement.

WRITE access will only allow values to be stored into a data-element. This is of particular interest in the declaration of arrays and records, to control access to their component data-elements, see sections 4.1.3 and 4.2.3 . WRITE access is discussed also in relation to parameter transfer in routines, see section 7.4 .

The default access for all declarations is both READ and WRITE, except for formal parameters of ROUTINES, see Chapter 7 .

3.12 PREDEFINED DATA TYPES

Some predefined data types are provided in the PLANC compiler. The predefined data types are defined in terms of the already described simple data types. The simple data types have operators and operations defined for them, however, the predefined have the same operators and operations as those defined for the base data type from which the predefined type has been derived. The following predifned data types are available :

1) BYTE - data-elements can contain single characters only.

2) BYTES - data-elements can contain character strings.

3) BITS - data-elements can contain sequences of bits.

3.12.1 BYTE DATA-ELEMENTS

The data-element of the BYTE predefined data type can contain a single character only. It is equivalent to the declaration :

```
INTEGER RANGE (0:255) : declaration-list
```

Thus BYTE data-elements may represent all characters in the ASCII character set. However BYTE identifiers may be used as integer identifiers with the operators defined for the integer data types.

Examples of BYTE declarations :

```
    BYTE : ch1,ch2,ch3
    BYTE : chinit:=#z       % an initialised byte data-element
%
%
    #x=:ch1                 % store ch. in a byte data-element
    ch1+chinit=:ch3         % add two byte data-elements
```

3.12.2 BYTES DATA-ELEMENTS

The BYTES predefined data type used for character strings will be discussed in section 4.1.7.1.

3.12.3 BITS DATA-ELEMENTS

The BITS predefined data type used for bit strings will be discussed in section 4.1.7.2.

3.13 TYPE SPECIFICATION AND USER DEFINED TYPES

The predefined data types and the modified data types are examples of variations of the simple data types described earlier. In a similar sense, the programmer may define his own data types in terms of the available data types, including the predefined and modified data types. The general form of a type specification is :

```
        TYPE new-type-ident = data-type
```

where

new-type-ident is an identifier to be used as the name of the newly defined data type

data-type is a simple, predefined, modified data type or a previously defined 'new' data type

It is important to note that a type specification statement will not cause any data-elements to be constructed. A type specification statement describes the precise characteristics to be associated with a data-elements defined by a declaration statement. Data-elements will only be constructed, and storage allocated for program execution, as a result of declaration statements for static data-elements or by using the NEW standard routine for dynamically created data-elements.

Examples of new type specifications and their use :

```
        TYPE mychar = INTEGER RANGE (0:127)  % ie. 7 bit characters
    % this new type can now be used in a declaration
        mychar : ch1,ch2,ch3

        TYPE colour = ENUMERATION (red,green,blue,black,white)
        colour : cl1,cl2,cl3
    % a new data type  colour' is now available

    % however, a similar effect could be achieved without creating
    % the new data type 'colour'
        ENUMERATION (red,green,blue,black,white) : cl1,cl2,cl3
```

3.14 TYPEOF STANDARD ROUTINE

The TYPEOF standard routine specifies identifiers to be of the same data type as a previously declared identifier. The general form of use of the TYPEOF invocation is :

```
TYPEOF p-ident : ident-list
```

where

p-ident is a previously declared identifier

ident-list is a list of identifier declarations

Example of use of the TYPEOF standard routine :

```
     INTEGER : int1,int2,int3
     TYPEOF int2 : id1,id2
% id1 and id2 are dependent on the data type of int2,
% ie. id1, id2 are currently of type integer
```

3.15 EQUIVALENT DATA STORAGE FOR DATA-ELEMENTS

The equivalence declaration will force two data-elements to begin at the same storage location, regardless of their data types. The general form of an equivalence declaration is:

```
data-type : identifier = previous-identifier
```

where

data-type is any valid data type

identifier is an identifier of type 'data-type'

previous-identifier is a previously declared identifier

Data-elements of different types require different amounts of storage, so it will be necessary to know precise implementation details of storage allocation in order to understand the consequences of overlapping data-elements with the equivalence declaration, see Appendix C.

Example of an equivalence declaration :

```
INTEGER : int1,int2
REAL : rl1,rl2=int1
```

The data-element for rl2 will begin at the same storage location as int1 but will not be of the same length.

3.16 PREDECLARATION

The predeclaration facility may be used if it is necessary to refer to a data-element in a statement which precedes the actual declaration of that data-element. A predeclaration must precede the statement which refers to the data-element. This predeclaration informs the PLANC compiler that an actual declaration will occur somewhere further on in the module.

A predeclaration is of the same form as the actual declaration, but a question mark character (?) follows the data-element name.

For example :

```
          INTEGER : int1?
```

is a predeclaration of int1 and further in the module there must be a following declaration :

```
          INTEGER : int1
```

The predeclaration is of particular use if two routines have mutual references, eg. if two routines invoke each other.

For example :

```
     % predeclaration of routine data-element rt2
          ROUTINE VOID,VOID : rt2?
     %
          ROUTINE VOID,VOID : rt1
     % invoke rt2
          rt2
          ENDROUTINE
     %
          ROUTINE VOID,VOID : rt2
     % Note the following invocation of rt1 prevents simply
     % exchanging the order of the routines
          rt1
     %
          ENDROUTINE
```

A further possible use of predeclarations is to initialize a static linked list of records.

For example :

```
% define a data type for records in the linked list
    TYPE myrecord = RECORD
                     myrecord POINTER : linkptr
                     INTEGER : recnumber
                    ENDRECORD
% initialise a static linked list of records
    myrecord : r1?,r2?,r3?  % predeclaration of data-elements
    myrecord POINTER : listhead:=ADDR(r1)
    myrecord : r1:=( ADDR(r2),1 )
    myrecord : r2:=( ADDR(r3),2 )
    myrecord : r3:=( NIL,3 )
% Note that predeclaration may be avoided by reversing the
% order of the last four lines
```

3.17 SIZE STANDARD ROUTINE

The SIZE standard routine returns the number of bytes used for the storage of the data-element associated with the identifier specified in the call to the SIZE routine. As the storage requirements vary with the different implementations of PLANC, see Appendix C, this standard routine gives access to the quantity of storage used for a particular data-element. This routine may also be used for composite data-elements which could be of particular use for dynamically created arrays or records, see section 4.5.

For example:

```
         REAL : rl1
         INTEGER2 : int2,int2size, realsize
     %
         SIZE rl1 =: realsize
     % store the number of bytes used for a floating-point value
         SIZE int2 =: int2size
     % store the number of bytes used for an INTEGER2 value
```

Note that the SIZE standard routine may be used to give the size of a data-element of a user defined data type which appears in a TYPE specification. Further, any data type keyword may also be used as the parameter of the SIZE invocation.

4 DATA DECLARATION AND COMPOSITE DATA TYPES

This chapter will describe the composite data types available in PLANC. Composite data types have components which are either further composite data types, or simple data types which have been discussed in Chapter 3 . In array and record composite data-elements, the component data-elements are uniquely identified and may be accessed individually. The following composite data types are available in PLANC :

1) ARRAY - has components, all of the same type.

2) RECORD - has components of different types.

3) SET - is a collection of elements, treated as an entity.

4) ROUTINE - is a subprogram to carry out some specific function.

4.1 ARRAYS

An array data-element is made up of a group of components, all of the same type. The array elements may be either of a simple data type or themselves be of a composite data type, eg. an array or record. An array whose components are arrays is called a multidimensional array. All elements of an array data-element are uniquely identified by an index value from a continuous integer range or from a range of values of an enumeration set.

Array data-elements are the basis for the predefined data types, BYTES for character strings and BITS for sequences of bits. Arrays may also be used to represent other data structures by defining new data types.

4.1.1 ARRAY DECLARATIONS

A declaration of an array data-element specifies the following information :

1) Array Name - an identifier which can be used to refer to the array data-element as a single entity or to refer to individual elements of the array by the use of unique index values.

2) Number of Dimensions - specifies the number of index values needed to uniquely identify an element of the array data-element.

3) Range of Values for each Dimension - specifies the valid range of values that each index may take in order to uniquely identify an element of the array cata-element.

4) Initial Element Values - optionally some or all array elements may contain initial values at the beginning of program execution.

The general form of a declaration of array data-elements is :

```
data-type ARRAY [ARRAY] ... : array-decl[,array-decl] ...
```

where

data-type — is a simple, composite or predefined data type.

ARRAY — is repeated as many times as the number of dimensions required for the array data-elements specified here.

array-decl — is declaration of one specific array data-element. It has the following general form :

```
ident(low-bnd:up-bnd[,low-bnd:up-bnd]      ...)[:=(value-
list)]
```

where

ident — is a valid identifier.

low-bnd — is a literal expression which results in an integer or enumeration value when evaluated. This value is the lowest value that an index for this dimension may take.

up-bnd — as for low-bnd and must be of the same data type as the low-bnd. This value is the highest value that the index for this dimension may take.

Note : low-bnd:up-bnd is called the <u>index set</u> and there must be one index set for each dimension specified for the array data-element.

value-list — is a list of literal values which will be the initial values of the array elements. For array elements of composite or predefined data types, the data-elements of the initial value list must be of the correct base type.

Note : 1. that literal expressions may be used, provided that the resulting value is of the correct type.

2. For array elements of the predefined type BYTES, string literal values will have apostrophes instead of parentheses.

The data type of all the elements of the array data-element is the data-type specified in the declaration statement. The number of array data-elements may be computed by taking the product of the number of distinct values that each index set contains, ie. the number of values for each dimension specified for a multidimensional array. The actual storage required for such an array depends on the storage required for a single array element, then multiplied by the number of elements specified in the array. For the storage requirements of the simple data types see Appendix C.

The array data-element may contain initial values when program execution begins. These values are specified in the list of literal expressions, which have evaluated results of the data type 'data-type'. The list of values is placed in the array in the following order; set each index to its lowest value, then vary the indices through their index sets to their highest value, with the last index changing most rapidly. For multidimensional arrays, if an initial value list is specified, then it must contain one level of parentheses for each dimension, to uniquely define the correspondence of the values to their array elements. An exception to this rule is available for BYTES arrays of more than one dimension, see section 4.1.7.1.

Note : this default sequence of elements of an array is the same as that used in the PASCAL language, but different to that used in FORTRAN. This is significant if modules of mixed languages are to communicate satisfactorily, see Mixed Language Programming, Appendix D .

If an array declaration contains a list of initial values which has fewer values than specified by the index set, the specified number of array elements will be initialized and the rest will be set to a null value, in fact binary zeroes. For multidimensional arrays, the first few elements of a group, corresponding to a particular index set, may be initialized by the use of parentheses.

If a list of literal expressions, to be used as initial values in an array data-element, is present in the declaration statement then the index set may be omitted and the PLANC compiler will supply implicit bounds so that the array will have sufficient elements to contain the list of initial values. In this case the list of initial values will implicitly determine the number of elements of the array data-element. The implicit bounds are zero (0) and the number of elements minus one.

Examples of array declarations :

```
% two one-dimensional arrays, same number of elements, but
% the values of each index range are different
    REAL ARRAY : vector1(1:11),vector2(-5:5)

% the second array has a list of initial values
    CONSTANT two=2
    INTEGER ARRAY : ar1(1:5),ar2(1:4):=(-2,4+two,21,-108)

% the array ar3 has the same size characteristics as ar2 above,
% with an index set, with values 0:3, implicitly specified by
% the list of initial values
    INTEGER ARRAY : ar3:=(-2,4+two,21,-108)

% an array whose elements are range modified to be 6 bit
% integers
    INTEGER  RANGE (1:63) ARRAY : modint(1:3):=(2,4,6)

% a real and an integer array with enumeration index sets
    ENUMERATION (red,yellow,blue,white,black) : colour
    REAL ARRAY : aren1(yellow:white):=(1.0,2.0,3.0)
    INTEGER ARRAY : aren2(red:blue):=(2,3,5)

% a 2 dimensional boolean array and a 3 dimensional real array
    BOOLEAN ARRAY ARRAY : bool2(1:5,1:10)
    REAL ARRAY ARRAY ARRAY : rl3(1:2,1:3,1:4)

% cube is a 3 dimensional array with implicit index sets
% equivalent to a declaration :
%     cube(0:2,0:1,0:1)
%
    INTEGER ARRAY ARRAY ARRAY :         &
      cube:=( ((1,3),(2,4)), ((0,0),(0,2)), ((-1,1),(1,-1)) )
```

4.1.2 ARRAY TYPE SPECIFICATION AND USER DEFINED TYPES

A type specification may be used to create a new data type based on the array data type. This newly defined data type may then be used for declaring data-elements with the characteristics of the newly defined data type. The general form of an array type specification is :

```
        TYPE type-ident = data-type ARRAY[ ARRAY] ...
```

where

type-ident   is an identifier which is the name of the new array data type.

data-type   is a simple data type as in an array declaration.

ARRAY;is repeated for the number of dimensions required for each array data-element to be declared of this new data type.

Note : For each 'ARRAY' keyword there must be an index set, specified explicitly or implicitly, in each data-element declaration of this new data type.

A type specification will not result in any data-elements being constructed, it only specifies certain characteristics that data-elements will have if they are declared to be of a newly specified type. Array data-elements will only be constructed in association with a declaration statement.

Examples of array type specifications :

```
        TYPE ivector = INTEGER RANGE (0:127) ARRAY
    % an array type of one dimensional arrays
    % with 7 bit unsigned integer array elements
        ivector : ivc1(1:10),ivc2(1:100)
    % 2 data-elements of the 'ivector' array data type

        ENUMERATION (red,blue,green,blue,pink) : colour
        TYPE artype = INTEGER ARRAY ARRAY
    % type specification
    %
        artype : ar1(red:blue,red:pink)      &
                  :=( (1,2,3,4,5),(6,7,8,9,10) )
    % this is a 2 dimensional 2*5 array with 10 integer elements
    % which may be accessed with enumeraticn identifier values
```

4.1.3 REFERENCE TO ARRAY ELEMENTS AND ACCESS MODE

In the executable part of a program it is necessary to refer to individual elements of an array data-element, either to store a value or to access a stored value. The general form of a reference to an array element is :

```
          array-ident(index-expr[,index-expr] ...)
```

where

array-ident   is the identifier in the array declaration.

index-expr   is an expression of integer or enumeration data type to match the type of the index set in the array declaration.

Note : there must be the same number of index-expr's in an array element reference, as index sets in the array declaration.

Examples of array references :

```
          BOOLEAN ARRAY : bool1(1:20)
     %
          TRUE=:bool1(2)
          TRUE=:bool1(1+1)  % is the same as the previous statement

          ENUMERATION (red,blue,green,pink) : colour
          INTEGER ARRAY ARRAY :iar1(red:green,blue:pink)
     %
          2=:iar1(blue,blue)  % store 2 in the array element
```

An exception to the above is available for BYTES arrays with more than one dimension. The last subscript may be omitted and the reference will be to the entire string, ie. the entire range of values of the last index set.

For example :

```
          BYTES ARRAY : b1(1:2,1:3):=('abc','xyz')
          BYTES : b2(1:3)
          b1(1)=:b2
     % the string 'abc' will be stored in array b2
     % Note, one extra ARRAY keyword is implicitly included in a
     % BYTES declaration
```

In the above example the reference to the array b1, b1(1), is equivalent to the subarray :

```
          b1(1:1,1:3)
```

In an array declaration the data type of the elements of the array may be a modified simple data type. In particular, the READ 'access' modified type may be used in the following manner :

```
REAL READ ARRAY : rlar1(1:2):=(8.0,9.0)
```

This declaration specifies that the array elements are for read access only. Consequently no values can be stored into the individual array elements during program execution.

4.1.4 OPERATIONS ON ENTIRE ARRAYS AND ARRAY ACCESS

The contents of an array data-element may be copied into another data-element by using the store operator. Such a copy operation treats an array as a single entity. An array copy is only allowable if both source and destination arrays have identical declaration characteristics, ie. elements of the same data type, same number of dimensions and the same index sets.

Example of an array copy :

```
        INTEGER ARRAY ARRAY : iarray1(1:2,1:2):=( (1,2),(3,4) ),  &
                              iarray2(1:2,1:2)
    %
        iarray1=:iarray2      % copy iarray1 into iarray2
```

An entire array, ie. all of its elements, may be assigned to a single value by using the store operator in the following way :

```
        expr=:array-ident
```

where

expr has a value of the same data type as the declared data type of the elements of the array.

array-ident is an array identifier.

Example of assigning a single value into an entire array :

```
        INTEGER ARRAY ARRAY : iarray(1:10,1:10)
    %
    %
        5+3**2=:iarray        % stores 14 in each array element
```

Arrays have an access mode, identical to that for simple data types, for operations which treat an array as a single entity. The entire access mode may be declared as READ or WRITE, following the ARRAY keywords.

Example of use of array access mode :

```
        INTEGER ARRAY READ : iar1(1:10)
```

is an array into which entire array operations cannot store values. However it is still valid to store into individual elements of the array.

If the declaration is :

```
        INTEGER READ ARRAY READ : iar2(1:10)
```

then it is not permitted to store into individual elements or into the entire array as an entity.

Note that the access mode keywords, READ/WRITE, may not be placed between the ARRAY keywords. READ/WRITE must precede or follow all the ARRAY keywords of any ARRAY declaration.

4.1.5 INDEX_SET_INFORMATION

All array data-elements have a descriptor which contains information specifying the number of dimensions, number of index sets, the range of values for each index set and the data type of the array elements. All array operations and operations on individual elements use this descriptor information.

The lower and upper bound values for each index set are available during program execution through the use of the following standard routines :

1) MININDEX (array-ident,dimension number) - returns the lower bound of the corresponding index set.

2) MAXINDEX (array-ident,dimension number) - returns the upper bound of the corresponding index set.

These routines are described in Standard Routines, section 7.9.

4.1.6 SUBARRAYS

A subarray is a part of an array which may be referred to as a single entity. A subarray is specified by using a subarray index set for each dimension of the original array. Each subarray index set must be a subset of the corresponding index set in the original array.

Examples of subarrays :

```
          REAL ARRAY : rvector1(1:10),rvector2(5:40)
     % copy one subarray to another
          rvector1(4:8)=:rvector2(24:28)

          INTEGER ARRAY ARRAY : intar1(0:10,1:5),intar2(1:11,-2:2)
     % copy subarrays of 2 dimensional arrays
          intar1(0:10,i:k-2)=:intar2(1:11,i-3:k-5)
          intar1(0:1,1:j)=:intar2(2:3,0:j-1)
```

If the ADDR standard routine (see section 7.9) is called with a subarray as a parameter then an array descriptor for the subarray will be constructed. This descriptor may be stored in a pointer data-element which is qualified to reference an array of these characteristics. The subarray may then be treated as if it were an array, just like a dynamically created array, and the IND standard routine could be used to obtain the values of elements of this subarray.

If an array is declared with two or more dimensions, then a subarray may be implied by omitting the last one or more dimensions. If the array is declared with n dimensions, and the subarray has the last k dimensions ommitted (k<n), then the subarray will have n-k dimensions.

For example :

```
          INTEGER ARRAY ARRAY : twod(1:100,1:100)
          INTEGER ARRAY : oned(1:100)
          INTEGER : sub1,sub2
     % a one dimensional subarray may be referred as follows
          twod(10)=:oned    % the explicit subarray twod(10:10,1:100)
```

Note that an element in the implied subarray twod(10), may be referred to by the form twod(10) (2). An alternative to using this form would be to refer to the original array twod, using twod(10,2) which gives much faster access at run-time.

4.1.7 PREDEFINED_DATA_TYPES_USING_ARRAYS

The array data type is used as a base type for the following data types :

1) BYTES - for character strings.

2) BITS - for bit strings.

4.1.7.1 BYTES_-_ARRAYS_USED_TO_REPRESENT_CHARACTER_STRINGS

A BYTES data-element can contain any number of characters. Each character is held as an unsigned 8 bit integer and is equivalent to the declaration :

```
          TYPE bytes = BYTE ARRAY PACKED
```

Note : the keyword PACKED will be discussed in section 4.2.5.

The declaration of a BYTES data-element includes one ARRAY keyword implicitly, as this predefined data type is defined as an array of BYTE data-elements.

The elements of a BYTES data-element, ie. a BYTE array, may be used as operands for integer operators or the entire array may be treated as an integer array, but the only specific character string operations provided by the PLANC compiler are assignment and the relational operators, see section 5.4. The user may of course create more string functions, eg. string concatenation.

Examples of BYTES data-elements :

```
          BYTES : magic(1:100)
      %
      % a data-element which can hold 100 separate characters
      %
          'abracadabra'=:magic(10:20)  % store 11 characters

          BYTES : string:='i am the greatest'
      % a data-element which can hold 17 characters
      % the first character can be referenced by
      %   string(0)
      % the second by
      %   string(1)        and so on .
```

If a BYTES array of more than one dimension is to be initialized, then an exception to the normal predefined data type facilities is available. This represents an array of strings, where the last dimension may be initialized by a whole string.

For example :

```
         BYTES ARRAY : bytes2by4(0:1,2:5):=('abcd','wxyz')
     % two strings, each containing 4 characters, in an array
```

It is of interest to note in the type specification, that the BYTES type is effectively specified in terms of another predefined type.

As a consequence of the data type BYTES being defined as a BYTE ARRAY, there may be a difficulty if an access mode, READ/WRITE is to be used for each array element, ie. each BYTE data-element which makes up the BYTES array. In order to declare an access mode for each array element, the access mode keyword, READ/WRITE, must precede all of the ARRAY keywords. Since the BYTES declaration includes an implicit ARRAY keyword, it is not possible to declare an explicit access mode keyword prior to the first ARRAY keyword. If such an explicit access mode for each element of a BYTES array is required, the user will have to construct his own declaration as a BYTE array, with the access mode keyword placed prior to all ARRAY keywords.

For example :

```
         BYTE ARRAY ARRAY PACKED : safe_els (0:9,0:9)
```

is exactly equivalent to the declaration

```
         BYTES ARRAY : safe_els (0:9,0:9)
```

However, if the array elements are to have a READ access mode only, then the following declaration is the only way to achieve this :

```
         BYTE READ ARRAY ARRAY PACKED : safe_els (0:9,0:9)
```

If a number of BYTES arrays were required with READ access mode for each element, a newly defined data type could be created for convenience.

For example :

```
         TYPE mybytes = BYTE READ ARRAY ARRAY PACKED
```

4.1.7.2 BITS - ARRAYS USED TO REPRESENT SEQUENCES OF BITS

A BITS data-element can contain a sequence of bits of any length. Each bit is represented by a BOOLEAN data-element compressed into succesive bits of storage. It is equivalent to the declaration :

```
TYPE bits = BOOLEAN ARRAY PACKED
```

Note : the keyword PACKED will be discussed in section 4.2.5 .

The elements of a BITS array may be used as operands for boolean operators or the entire array may be treated as a boolean array, but there are no specific bit operations provided by the PLANC compiler. The user may of course create bit functions, eg. concatenate two bit strings. An element of a BITS array may take the values TRUE and FALSE.

Examples of BITS data-elements :

```
    BITS : flags1(1:10)
% set individual flags
    TRUE=:flags1(1)
    FALSE=:flags1(3)

    BITS ARRAY : flags2(1:2,1:2):=( (TRUE,FALSE),(TRUE,TRUE) )
    BOOLEAN : bl1
% access a single bit value
    flags2(2,2)=:bl1
```

4.2 RECORDS

A record data-element is made up of components each of which may be of any data type, simple, composite or newly defined. Each component of a record data-element is uniquely identified by an identifier within the record declaration. The RECORD data type must be declared in a TYPE specification statement; declaration statements for RECORD data-elements must use a record data type specified previously in the program in which the declaration statement occurs.

4.2.1 RECORD DECLARATIONS AND TYPE SPECIFICATION

A record type specification specifies the following information :

1) Record Type Name - an identifier to be used in declaration statements to refer to the record data type.

2) Component Data Type - the data type of each component of the record data-element.

3) Component Identifier - the name used to refer uniquely to each component of a record.

The general form of a record type specification is :

```
TYPE rec-type-ident = RECORD
                        .
                        comp-data-type : comp-ident-list-1
                        comp-data-type : comp-ident-list-2
                        .
                        comp-...-list-n [MOD literal-expr]
                        .
                      ENDRECORD
```

where

rec-type-ident is an identifier to name the record data type.

comp-data-type is the data type of the component data-element.

comp-ident-list is one or more component identifiers.

Note : use a list if a number of components of the same type, grouped together, are required.

literal-expr is any literal expression.

A record type specification will not result in any data-elements being constructed, it is only a description of which component data-elements are constructed for declaration statements which use this newly specified record data type. Records which are specified independently of each other, ie. not variants, may use the same identifier name for a component.

Examples of record type specification and declaration :

```
     % specification of a 'parts' record type
         TYPE partrec = RECORD
                         INTEGER : partnumber
                         BYTES   : partname (1:20)
                         REAL    : partprice
                        ENDRECORD
     % each record has 3 components - a number, name and
     % price for a part
     %
     % declare 2 data-elements of the 'parts' data type
         partrec : mypart,yourpart

     % a record may have arrays or records as components
         TYPE person = RECORD
                        BYTES : personname(1:20)
                        INTEGER : age
                       ENDRECORD
         TYPE team = RECORD
                      BYTES : teamname(1:15)
                      INTEGER ARRAY : teamnumbers (1:30)
                      person ARRAY : teammembers (1:30)
                     ENDRECORD
     % the record 'team' has an array 'teamnumbers' and
     % an array of records 'teammembers'
         team : myteam      % a 'team' data-element declaration
```

The components of a record data-element may be initialized by the compiler so that the values will be present when the program begins execution. The initial values must be specified in the record data-element declaration. If any components of a record data-element are to be initialized, then all components of that record must be given an initial value.

Example of initializing record components :

```
          TYPE partrec = RECORD
                          INTEGER : partnumber
                          BYTES : partname (1:20)
                          REAL : partcost
                        ENDRECORD
     % declare a record data-element with components initialized
          partrec : psupply:=(123,'power supply',100.2)
```

Note that if equivalence is used within record components and initial values are to be placed in the data-element, only the first declaration of the data-element may have an initial value.

The storage alignment of record component data-elements will be carried out according to the descriptions in Appendix C . Alignment of record component data-elements may be explicitly controlled by the MOD alignment clause. A MOD alignment clause forces the data-element to be allocated at an address, whose displacement from the start of the record, is a multiple of the resulting value from evaluation of the expression in the MOD clause.

4.2.2 *VARIANT RECORD TYPE SPECIFICATION*

Record data-elements declared for a given data type have so far all had the same structure of components. It is possible to specify two or more records which have some common components and some components which vary from one record to the next. Such related records are called variant records. Variant records may be specified by specifying a record type with all the common components, called the base record and then specifying each variant record as comprising the base record plus those components particular to the variant record. The general form of a type specification of a variant record is :

```
TYPE var-rec-ident = base-rec-ident                &
       RECORD
          var-comp1-data-type : var-comp1-ident-list
          var-comp2-data-type : var-comp2-ident-list
             .
             .
       ENDRECORD
```

where

var-rec-ident — is an identifier to name the variant record type.

base-rec-ident — is the identifier naming the base record type.

var-comp1-data-type — are the data types of the additional components of the variant record.

var-comp1-ident-list — are identifiers to uniquely name the additional components of the variant record.

Following type specifications of two or more variant record data types, declarations of record data-elements of the variant data type may be made as for normal record data-element declarations.

Example of variant record specification and declaration :

```
    % specify a 'vehicle' record data type
        TYPE vehicle = RECORD
                            REAL : weight,length,width,height
                          ENDRECORD
    % specify first variant record data type using 'vehicle' as
    % the base record
        TYPE bus = vehicle RECORD
                                INTEGER : seats,numbercrew
                              ENDRECORD
    % specify second variant record data type
        TYPE truck = vehicle RECORD
                                  REAL : loadcapacity
                                  BOOLEAN : automatic
                                ENDRECORD
    % declare 'bus' and 'truck' data-elements with initial values
        bus : localbus:=(100.0,10.1,3.4,2.1,44,1)
        bus : toursbus:=(150.0,11.3,3.4,2.1,35,3)
        truck : tiptruck:=( 50.5,8.6,3.2,1.9,45.0,TRUE)
```

Note that a record pointer identifier, declared for the base record, may be used to contain addresses of base record data-elements or any of its variant record data-elements.

If a routine declaration contains a base record data type for a parameter, then an invocation of this routine may have any variant of this record data type as an actual parameter. However, if the routine declaration contains a variant record data type as a formal parameter, only this variant record data type (or further variants of this data type), may be used as an actual parameter in a routine invocation.

4.2.3 REFERENCE TO RECORD COMPONENTS AND ACCESS MODE

In the executable part of a program it is necessary to refer to components of a record data-element, either to store a value or to access an already stored value. The general form of a reference to a record data-element component is :

```
        data-el-ident.comp-ident
```

where

data-el-ident is the identifier in a record declaration. Note that it may be a record pointer, but the following references will all access the same data-element :

```
     rec.element          % rec is a record
     recp.element         % recp is pointer to rec
     ADDR rec.element
```

comp-ident is the component identifier in the record type specification.

Note : if the component is itself a record, then use a further dot followed by a component identifier from that record.

Examples of record component references :

```
         TYPE person = RECORD
                        BYTES : givenname (1:15)
                        BYTES : familyname (1:30)
                        INTEGER : age,heightcm
                       ENDRECORD
     % declare a 'friend' data-element of data type 'person'
         person : friend:=('Fred','Bloggs',49,179)
     % access a component of a 'friend' data-element
         friend.age=: ...    % store the age of 'friend'
     % would access the value 49

     % specify a 'team' record type using 'person' from above
         TYPE team = RECORD
                      person : captain
                      INTEGER ARRAY : teamnumbers (1:5)
                     ENDRECORD
     % declare a 'team' data-element
         team : usteam:=( ('Ronald','Raygun',79,141) ,1,3,5,7,9)
     % access a component of a record within a record
     % ie. the 'family name' of the 'captain' of the 'usteam'
         usteam.captain.familyname=: ...
     % would access the value 'Raygun'
```

4.2.4 OPERATIONS ON ENTIRE RECORDS AND RECORD ACCESS

The contents of a record data-element may be copied into another record data-element by using the store operator. For such a copy the record data-elements must be of the same record data type.

Example of a record copy :

```
     % type specification of an 'address' record
         TYPE address = RECORD
                          BYTES : name(1:30)
                          INTEGER : streetnumber
                          BYTES : streetname(1:20)
                          BYTES : city(1:15)
                        ENDRECORD
     % declare two address data-elements
         address : NDaddress:=('NDOSLO',20,'jerikoveien','oslo 10')
         address : myaddress
               .
     % copy the initialized address to the other data-element
         NDaddress=:myaddress
```

Records have an access mode, identical to that for simple data types, for operations which treat a record as a single entity. The entire access mode may be declared as READ or WRITE, following the RECORD keyword.

Example of use of record access mode :

```
         TYPE address = RECORD READ      % same as previous record
                              .
                              .
                        ENDRECORD
```

is a record into which entire record operations cannot store values. However it is still valid to store into individual components of such a record.

If the declaration is :

```
         TYPE address = RECORD READ
     %
                          INTEGER READ : streetnumber
     %
                        ENDRECORD
```

then it is not allowable to store into the name component of the address record or into the entire record data-element as an entity.

4.2.5 PACKED OPTION FOR ARRAYS AND RECORDS

For data-elements of simple data types storage may be wasted in particular machine implementations. For the composite data types, arrays and records, space required for data-elements can be minimized by using the option PACKED in a TYPE definition or a declaration, in the case of an array.

For example :

```
          INTEGER1 ARRAY PACKED : minints(1:500)
      % will require 250 words on the ND-100 whereas
          INTEGER1 ARRAY : ints(1:500)
      % will require 500 words and use only half of each word
```

Use of the PACKED option will minimize storage requirements but it should be noted that this may cause a program to execute more slowly because of time taken to extract component data-elements from the more compact storage allocation being used.

Further examples of the effect of the PACKED option :

```
      % on the ND-100
          TYPE letters = ENUMERATION (a,b,c,d)
          letters ARRAY : waste(1:10)
      % will require a 16-bit word per array element, ie. 10 words
          letters ARRAY PACKED : nowaste(1:10)
      % will require an 8-bit field per array element, ie. 5 words

      % on the ND-100
          TYPE myrec = RECORD PACKED
            letters : alphabet  % 2-bit  instead of 16-bit field
            BYTE : bytvar       % 8-bit  instead of 16-bit field
            BOOLEAN : b1        % 1-bit  instead of 16-bit field
                      ENDRECORD
```

The specific rules of how PACKED affects the storage requirements of a data-element, on both the ND-100 and the ND-500, are described in Appendix C.

4.3 SETS

A set data-element is of a composite data type that, like the array and record, is made up of a collection of components. However, unlike the array or record, we neither index nor access the individual components of a set. Instead a set is used only as a single entity.

The components that comprise a particular set are chosen from the possible values of a simple data type called the base type of the set. The valid base types for sets in PLANC are :

1) INTEGER RANGE

2) ENUMERATION

A set data-element may represent all subsets of the value of the base data type of the set, including the 'empty' set. There is no mutual ordering between the components of a set.

Thus the set data type in PLANC corresponds to the mathematical notion of a set, with some restriction as to what may form the members of the set. The usual mathematical set operations, eg. union, intersection, difference and complement are available as operators for use with set operands.

4.3.1 SET DECLARATIONS

A set data-element declaration specifies the following information :

1) Set Name - an identifier which can be used to refer to the set data-element as a single entity.

2) Base Type - a data type which will specify all the possible members of a set data-element.

3) Initial Members - optionally specify a subset of the base type values to be members of a set at the beginning of program execution.

The general form of a declaration of set data-elements is :

```
base-type SET : ident[:=memb-list] [,ident[:=memb-list] ]
```

where

base-type is one of the data types ENUMERATION, INTEGER RANGE or a data type newly defined with one of these as a base type.

Note : integer range base type is restricted to a maximum of 256 values and the lower bound must be zero. Ie. an INTEGER RANGE must be 0:x, where x <= 255.

ident is a valid identifier.

memb-list is a list of values, selected once only, from the possible values of the base data type.

Note : that literal expressions may be used, provided that the resulting value is of the correct type.

The 'memb-list' may be partly or entirely specified by an implied range providing that the list of values is of the correct data type, see section 2.13 .

If the 'memb-list' is omitted, then the set will be empty when program execution begins.

A set data-element will require enough storage to hold an indicator of the presence or absence of every possible member of the set, ie. every valid value of the base type of the set. For details of the actual storage used see Appendix C.

Examples of set declarations :

```
% specify an enumeration data type
   TYPE day = ENUMERATION (monday,tuesday,wednesday,
                           thursday,friday,saturday,sunday)
% declare a set data-element with the weekend days as members
   day SET : weekend:=(saturday,sunday)
% declare a set data-element for the week days using an
% implied enumeration range
   day SET : workdays:=(monday:friday)

% declare a set of base type integer using an implied integer
% range to specify a list of integer values
   INTEGER RANGE(0:255) SET : twenties:=(20:27,28,29)

% declare a set which will be empty initially
   INTEGER range(0:255) SET : emptyint
```

4.3.2 SET_TYPE_SPECIFICATION_AND_USER_DEFINED_TYPES

A type specification may be used to describe a new data type based on the set data type. This newly defined data type may then be used for declaring data-elements with the characteristics of the newly defined data type. The general form of a set type specification is :

```
          TYPE set-type-ident = set-base-type SET
```

where

set-type-ident is an identifier which is the name of the new set data type

set-base-type is the base data type for this set data type.

A type specification will not result in any data-elements being constructed, it only specifies certain characteristics that data-elements will have if they are declared to be of a newly specified type. Set data-elements will only be constructed in association with a declaration statement.

Examples of set type specifications :

```
          TYPE numbers = INTEGER RANGE(0:127) SET
     % declare data-elements of the 'numbers' data type
          numbers : tensset:=(10,20,30,40,50,60,70,80,90)
          numbers : digitsset:=(0:9)

          TYPE colours = ENUMERATION (black,red,blue,green,white)
          TYPE houses = colours SET
     % declare a data-element of the 'houses' data type
          houses : myhouse:=(red,white,blue)
```

4.3.3 OPERATIONS ON SETS

The relational operators (see, section 5.4) may be used with set data-elements. As for other data types, evaluation of a relational operator with two set data-elements as operands will give a boolean resulting value, ie. TRUE or FALSE. The relational operators and their meanings when used with set data-elements as operands are as follows :

```
=         true if  both sets contain the same members.
><        true if at least one member of one set is not a
          member of the other set.
>=        true if the left-side set has as a subset the
          right-side set.
<=        true if the left-side set is a subset of the
          right-side set.
>         true if the left-side set has as a true subset
          the right-side set.
<         true if the left-side set is a true subset of
          the right-side set.
IN        true if the left-side identifier is a member
          of the right-side set.
```

Note : the IN operator is the only relational operator without both operands as sets. The first operand data-element of the IN operator must have a base type of INTEGER RANGE, ENUMERATION or POINTER and the second operand data-element is a set of the corresponding base type.

Examples of sets and relational operators :

```
% declare some sets
    TYPE day = ENUMERATION (monday,tuesday,wednesday,    &
                 thursday,friday,saturday,sunday)
    day SET : week:=(monday,tuesday,wednesday,thursday, &
                      friday,saturday,sunday)
    day SET : weekend:=(saturday,sunday)
    day SET : workdays:=(monday,tuesday,wednesday,       &
                           thursday,friday)
%
      expression                           result

    week = workdays                          false
    weekend >< workdays                      true
    week >= workdays                         true
    week >  workdays                         true
    weekend <= week                          true
    weekend <  week                          true
    monday IN weekend                        false
    monday IN week                           true
```

The store operator =:, see section 5.1, may be used with set data-elements as operands. It will have the effect of setting the members of one set data-element exactly equal to the members of another set data-element.

Example of sets and the store operator :

```
        INTEGER RANGE (0:10) SET : odds:=(1,3,5,7,9),numbers
    % store the members of set 'odds' in set 'numbers'
        odds=:numbers
```

Beware that the way the set "odds" is initialised above, cannot be used in an executable statement in exactly the same way, eg.

```
        (1,3,5,7,9)=:numbers
```

will give a compile error. The correct way to specify an unnamed set with a constant group of members requires the set base data type. This is described following the description of the logical operators below.

The logical operators, see section 5.3, may be used with set data-elements. Evaluation of logical operators with set data-elements as operands gives a resulting value of the set data type with the exception of the ABS operator which gives an integer result. The set operators and their meanings when used with set data-elements as operands are as follows :

AND set intersection, ie. result is a set with members which are members of both operand sets.

OR set union,ie. result is a set with members which are members of either operand set or both.

XOR set difference, ie. result is a set with members which are members of one of the two operand sets and not members of the other.

NOT set negation, ie. result is a set which has as members all the members which are not members of the operand set.

ABS cardinal number, ie. result is an integer value of the maximum possible number of members of the operand set.

Examples of sets and logical operators :

```
    % declare some sets
        TYPE colour = ENUMERATION (red,green,blue,pink,ash,  &
                                   yellow,white,black)
        colour SET : bright:=(red,green,yellow,pink),anycolour
        colour SET : pastel:=(blue,yellow,pink)
        INTEGER : int1
    % union - result will have red, green, yellow, pink, blue
        bright OR pastel =: anycolour
    % intersection - result will have yellow, pink
        pastel AND bright =: anycolour
    % difference - result will have red, green, blue
        bright XOR pastel =: anycolour
    % negation - result will have ash, yellow, white, black
        NOT bright =: anycolour
    % set cardinal number - result is 8
        ABS bright =: int1
```

The following standard routines are provided to carry out operations on set data-elements :

1) Specify a set data-element with a constant group of members.

2) INSERT

3) REMOVE

To specify an unnamed set data-element with a constant group of members use the general form

```
set-data-type (memb-list)
```

where

set-data-type is data type with a set base data type.

memb-list is a list of literals, selected once only, from the possible values of the base data type.

Note : 1. this list may include literal expressions which are to be evaluated at compile-time.

2. omission of the 'memb-list' from the parenthesis denotes the 'empty' set for that base data type.

Example :

```
     TYPE tnumbers = INTEGER RANGE (0:100) SET
     tnumbers : numbers
     TYPE colour = ENUMERATION (red,blue,grey,pink,black)
     TYPE tcolour = colour SET
     tcolour : luckyset
     INTEGER : int1
 % store an unnamed constant set data-element
     tnumbers (1,3,5,7) =:numbers
     tnumbers (1,3,5:10) =: numbers
     tcolour (blue:black) =:luckyset   % lots of luck !
 % use an expression evaluated at compile-time
     CONSTANT int2=15
     tnumbers (int2*3+4,int2:int2+5)=:numbers
 % an empty 'colour' set data-element
     tcolour () =:luckyset          % no luck at all !
```

Restriction : such an unnamed set data-element with a constant group of members, must not be the first statement of a routine, unless the entire statement is contained within parentheses.

Add a member to a set data-element

```
          set-member-ident INSERT set-ident
```

where

set-member-ident is a data-element of the set base data type.

Note : this may be an expression to be evaluated at run-time.

set-ident is a set identifier.

Example :

```
          INTEGER : int
          INTEGER RANGE(0:100) SET : numbers
     % add a member to the 'numbers' set data-element
          3 INSERT numbers
          int*2 INSERT numbers
```

Remove a member from a set data-element

```
          set-member-ident REMOVE set-ident
```

where

set-member-ident is a data-element of the set base data type.

Note : this may be an expression to be evaluated at run-time.

set-ident is a set identifier.

Example :

```
          INTEGER : int
          INTEGER RANGE(0:10) SET : evens:=(0,2,4,6,8,10)
     % remove a member from the 'evens' set data-element
          6 REMOVE evens
          int+5 REMOVE evens
```

4.4 ROUTINES

The 'routine' is defined in the PLANC language as a composite data type. While this may seem a little unusual, it is of benefit in declaring a routine name to be used as a generic function with in fact a family of similar routines which differ only in that their parameters are of different data types and perhaps their return values too, eg. a 'plus' operator may be thus created for integer,real and complex parameters.

A full description of the syntax of routine type specification, declaration, invocation and the use of parameters to communicate information to and from routines may be found in Chapter 7, ROUTINES.

4.5 DYNAMIC ALLOCATION OF DATA-ELEMENTS

During execution of a PLANC program, data-elements may be dynamically created and destroyed in storage. The actual storage used for dynamically created data-elements may be the program stack or an INTEGER array. If the program stack is used, it must be declared with enough space to hold all the dynamically created data-elements as well as all the other usual run-time requirements. One or more INTEGER arrays may be used as storage for dynamically created data-elements.

The NEW standard routine will dynamically create unnamed simple or composite data-elements. Invocations to the NEW standard routine return a pointer data-element of the type of the parameter used in the call. Invocations of the NEW standard routine are as follows :

For simple or composite data-elements use

```
NEW data-type [IN int-array-ident]
```

where

data-type — is any simple, composite, predefined or user defined data type.

int-array-ident — is an integer array identifier.

For arrays or subarrays it is possible to use

```
NEW ( ar-type-ident(index-set[,index-set] ...) )[as above]
```

where

ar-type-ident — is array data type identifier.

index-set — is an index set specifier for each corresponding index set for this array data type.

Example of dynamic creation of a simple data-element :

```
        INTEGER ARRAY : store(1:1000)
        REAL POINTER : rlptr
    %
        NEW REAL=:rlptr
    % dynamic creation of a real data-element on the program stack
    %
        NEW REAL IN store=:rlptr
    % dynamic creation of a real data-element in an integer array
```

Dynamically allocated data-elements will be created in the local data area of a routine unless an INTEGER ARRAY from an outer level routine is used in the NEW routine call. Note that all data-elements, including those dynamically created, in the routine's local data area will be lost when an exit from a routine occurs.

The DISPOSE standard routine is used to deallocate dynamically created data-elements, ie. a data-element which has been created by use of the NEW data type IN array standard routine. Invocations of the DISPOSE standard routine are to be used as follows :

```
DISPOSE pointer-ident
```

where

pointer-ident is a pointer data-element with a value pointing to the data-element to be deallocated.

During execution, an INTEGER ARRAY POINTER called FREE_P is available. It is initialized to point to the memory location immediately following the PLANC library routines loaded from the appropriate PLANC library files. In order to safely use this pointer to utilise the free space, the library routines must be loaded last.

In order to use the free space available, the declaration

```
IMPORT INTEGER ARRAY POINTER : FREE_P
```

must appear in the appropriate module. MININDEX(IND(FREE_P),1) and MAXINDEX(IND(FREE_P),1) give the low and high addresses of the free memory area, represented as unsigned integers. This pointer may be used with the NEW standard routine as follows :

```
NEW ... IN IND(FREE_P)=:ptr
```

Examples of dynamic creation of array and record data-elements :

```
% specify an array data type
   TYPE doublereal = REAL ARRAY ARRAY
% declare a pointer data-element for the array data-element
   doublereal POINTER : arraypointer
   REAL : rl1
%    .
% dynamically create an array and store its pointer value
   NEW ( doublereal(1:5,0:10) ) =: arraypointer
%    .
% access an element of the array data-element as follows
   IND (arraypointer) (1,10) =: rl1  % store value in rl1



% specify a record data type
   TYPE complex = RECORD
                   REAL : realpart,imagpart
                  ENDRECORD
% declare a constant value record data-element
   complex : constcomplex:=(1.0,1.0)
% declare pointer data-element for the 'complex' data type
   complex POINTER : complexpointer
%    .
% dynamically create another 'complex' record
   NEW (complex) =: complexpointer
% store the constant record into the dynamically created
% 'complex' record data-element
   constcomplex =: IND( complexpointer )
```

4.6 PROCESSING OF RECORDS IN LIST STRUCTURES

The following standard routines are available for processing linked lists of record data-elements :

1) The INSERT standard routine will add a record data-element to the front of a linked list.

2) The APPEND standard routine will add a record data-element to the end of a linked list.

3) The REMOVE standard routine will remove a record data-element from anywhere in a linked list.

The general form of the invocations of all of these standard routines is :

```
rec-pntr INSERT list-pntr-range
```

where

rec-pntr is a pointer to the record to be processed.

list-pntr-range
is a pointer implied range, describing the linked list.

The use of these list processing routines is illustrated in the following code examples.

Set up a static linked list.

```
% define a record data type for the linked list
   TYPE myrecord = RECORD
                    myrecord POINTER : linkptr
                    INTEGER : recordnumber
                   ENDRECORD
% initialise a static linked list of records
   myrecord : r1?,r2?,r3?  % predeclaration of data-elements
   myrecord POINTER : listhead:=ADDR(r1),anyrecptr
   myrecord : r1:=( ADDR(r2),1 )
   myrecord : r2:=( ADDR(r3),2 )
   myrecord : r3:=( NIL,3 )
% declare some records to illustrate list processing
   myrecord : front:=( NIL,-1 ),back:=( NIL,99 )
   myrecord POINTER : frontptr:=ADDR(front),backptr:=ADDR(back)
```

The record <u>front</u> may be added to the start of the linked list by the statement,

```
frontptr INSERT listhead:linkptr
```

Following the execution of this statement, the linked list will contain four records whose record numbers are -1, 1, 2, 3.

The record back may be added to the end of the linked list by the statement,

```
backptr APPEND listhead:linkptr
```

Following the execution of this statement, the linked list will contain five records whose record numbers are -1, 1, 2, 3, 99.

The record r1 may be removed from the linked list by the following statements,

```
ADDR(r1)=:anyrecptr
anyrecptr REMOVE listhead:linkptr
```

Now the linked list will have only four records, with the record numbers -1, 2, 3, 99.

The standard routines will do all the necessary changes to the linkptr component data-elements of records affected by the changes in the linked list, eg. when record r1 is removed, record number -1 is changed to point to record r2 (number 2).

Record data-elements may be created dynamically by the use of the NEW standard routine. Such record data-elements may be manipulated in linked lists in the same way as the explicitly declared record data-elements above. In fact an entire list may be constructed from such unnamed dynamically allocated record data-elements.

If a new record is to be placed in the middle of the linked list, then the program will have to change the linkptr component data-elements explicitly.

Note that the standard routines INSERT, APPEND and REMOVE will not give any error indication if the record pointer in the routine invocation is empty, ie. the pointer to the record to be processed has a value NIL. This also applies to the REMOVE standard routine if the linked list is empty. Beware that if INSERT or APPEND is used on a record that is already in a linked list, there is no error indication, but the address link field will be overwritten.

5 EXPRESSIONS - FORMATION AND EVALUATION

An expression comprises operators and data-elements as operands, formed according to a set of rules. During program execution, an expression may be evaluated to give a resulting value which may be stored in a data-element. PLANC, unlike most high level languages, does not have an assignment statement. It has assignment operators which may be used within expressions to store any temporary resulting value during the evaluation of an expression. At any point during evaluation of an expression, a temporary resulting value is available. Evaluation of one expression may store a number of values into data-elements, or if the expression is simply to invoke a routine with no out-value, see section 7.2, then there is no resulting value and no value is stored. The PLANC compiler will, if possible, try to evaluate an expression at compile-time, eg. if it contains literals only.

The operands used to form an expression may be literals, identifiers or routine invocations. An expression must contain operands whose corresponding data-elements are of one data type only, or parts of the expression must give a resulting value data-element of the correct data type required for further evaluation. This means that in general, there is no automatic conversion of the operand data-elements to the data types required by a specific operator. A routine invocation, within an expression evaluation, may have a side-effect of modifying a data-element value which is to be used later in the evaluation.

The operators in PLANC are defined for one or more data types. The following sections will describe all the available operators for each specific data type. Further, some operators are binary, ie. they may be used with two operands. For example, the sum of the values held in two integer data-elements may be obtained by the following part of an expression,

```
            integ1+integ2
```

by using the binary + operator for the integer data type. Other operators are unary, ie. they may be used with only one operand. For example, the complement of a boolean data-element may be obtained by the following part of an expression,

```
            NOT bool1
```

by using the unary NOT operator for the boolean data type. The evaluation of any operator and its operands will give a resulting value, except for routines with no out-value. This resulting value, which the run-time system may store in a temporary data-element, may be explicitly stored by the use of the assignment operators.

The operators available in PLANC each have a priority which determines the order of evaluation within the expression. An expression is evaluated by first forming the resulting values of the highest priority operators. These resulting values replace the operator and its operands and then the next highest priority operators are evaluated. For operators of the same priority, evaluation is from left to right.

Parentheses may be used to enclose part of an expression, causing that part to be evaluated separately from anything outside the parentheses.

User defined routines may be used within expresions and will be evaluated accordingly. Such routines have a higher priority than all the PLANC defined operators.

There are four classes of operators :

- assignment
- arithmetic
- logical
- relational

5.1 ASSIGNMENT OPERATORS

PLANC has two assignment operators which may be included within expressions. The assignment operators are used to store values, into data-elements, during evaluation of an expression. More than one assignment operator may be used in an expression, causing a number of values to be stored during evaluation of this expression. PLANC has no distinct assignment statement as many other high-level languages have.

The assignment operators have a priority associated with each side of the operator. The left-side priority is the lowest possible priority, to ensure that the entire expression to the left of the operator has been evaluated before evaluation of the assignment operator.

Both operands for an assignment operator may be of any simple, composite or predefinec data types. Both operands must be of the same data type. If however the operands are modified integer or real data types, they may be of different modified data types, ie. integer range or real precision, and appropriate conversion will take place prior to evaluation of the assignment operator, provided the receiving data-element is large enough to contain the value to be stored. If not, truncation will occur and no run-time error indication will be given.

If the operands are data-elements of composite data types, then the value of the entire data-element will be moved by the store operator, eg. a store operator with array operands will move the entire array as an entity, see section 4.1.4

The two assignment operators are :

Operator	Priority	Operation	Data types
=:	1, left-side 12, right-side	Store	all simple, composite and predefined
:=:	1, left-side 12, right-side	Change	all simple

When evaluation of an expression reaches a <u>store operator</u>, the resulting value of the part of the expression, immediately to the left of the store operator, will be stored into the data-element associated with the operand immediately to the right of the store operator.

The resulting value after evaluation of a store operator has the same value as the resulting value immediately prior to the evaluation of the store operator, ie. evaluation of a store operator does not change the resulting value of the expression during evaluation.

For example :

1. `53=:int`

will store the integer literal value in the integer data-element associated with the identifier int.

2. `3+5=:int`

will evaluate the sum of the two integer literals first because the integer + operator has a priority of 8. The left-side priority of the store operator is 1, ie. lower than that for the + operator, and thus it will be evaluated after the +. The resulting value of evaluation of the integer + operator is 8, and will then be stored in the integer data-element associated with the identifier int.

3. `intval=:int`

will store the value stored in the data-element associated with the identifier intval, into the data-element associated with the identifier int.

4. `2+2=:int1=:int2`

will store the value of the sum, 4, into the data-element associated with the identifier int1. The resulting value at this point of the expression evaluation is 4. Then evaluation of the second assignment operator stores the resulting value 4 into the data-element associated with the identifier int2.

5. `1+2=:int1+4=:int2`

will have a resulting value 3 from the first sum. Evaluation of the first store operator will store the resulting value 3 in the data-element associated with the identifier int1. Then second + operator will have a resulting value of the sum, 3+4. This resulting value, 7, will be stored by the second store operator into the data-element associated with the identifier int2.

6.

```
5*4+1=:int
```

will store the value of the entire expression, ie. 21, into the data-element associated with the identifier int. If however, parentheses were used,

```
5*(4+1)=:int
```

the order of evaluation of the operators is different. In the expression without parentheses, the product 5*4 is evaluated to give the resulting value 20. Then the sum 20+1 is evaluated to give the resulting value 21, which is then stored. In the expression with parentheses, first the sum 4+1 is evaluated to give the resulting value 5. Then the product 5*5 is evaluated to give the resulting value 25, which is then stored. Note that the parentheses not only change the order of evaluation within the expression, but cause a different final result, depending on the mixture of operators used in the expression.

When evaluation of an expression reaches a change operator, the resulting value of the part of the expression, immediately to the left of the store operator, will be stored into the data-element associated with the operand immediately to the right of the store operator. This is identical to the store operator.

The resulting value, from evaluation of a change operator, is different to that of a store operator. The value of the data-element to receive the value to be stored by a change operator, immediately prior to evaluation of the change operator, will be the resulting value following evaluation of the change operator.

For example :

```
1.       3=:int    % store 3 into data-element associated with int
         4:=:int
```

will store the integer literal value 4 into the data-element associated with the identifier int, but the resulting value of the expression following evaluation of the change operator is 3, ie. the value that was in the data-element associated with int before evaluation of the change operator.

```
2.       3=:i       % store 3 into data-element associated with i
         4=:j       % store 4 into data-element associated with j
         i:=:j=:i   % exchange the values of i and j
```

will store the value, 3, from the data-element associated with the identifier i into the data-element associated with j. However the resulting value of the change operator is the value in j prior to evaluation of the change operator, ie. 4. Then the resulting value, 4, is stored by the second operator in the expression, ie. 4 is stored into the data-element associated with i.

5.2 ARITHMETIC_OPERATORS

PLANC has a number of arithmetic operators which are available for operands whose data-elements are integer or real data types. There are both unary and binary arithmetic operators. The operands for a binary operator must both be either real or integer, but the operands may vary in the declared modifications, ie. range for integer and precision for real.

The following table lists all the available arithmetic operators :

Operator		Priority	Operation	Data types
+	binary	8	addition	integer, real
-	binary	8	subtraction	integer, real
-	unary	10	negation	integer, real
*	binary	9	multiplication	integer, real
/	binary	9	division	integer, real
**	binary	11	exponentiation	integer, real
ABS	unary	11	absolute value	integer, real
MOD	binary	11	modulo	integer
SHIFT	binary	8	shift bits	integer

The binary operators +, -, * and /, and the unary operators - and ABS can have operand data-elements of either integer or real data types. Further, the operands may be modified, ie. integer range or real precision. Various modified integer data type operand data-elements may be mixed when used with the binary operators. Likewise, modified real operands may be mixed when used with the binary operators.

The resulting value data-element will be of the same data type as the operands. If the operands are different modifications of one data type, then the resulting value will be a data-element of the data type appropriate to hold the larger of the two operand modified data types, ie. for integer data-elements, a data-element of the larger range, and for real data-elements, a data-element of the larger precision.

For example :

```
        REAL PRECISION (15) : rl1
        REAL PRECISION (7)  : rl2
    %
        ... rl1+rl2 ...
```

evaluation of the real addition operator within an expression would give a resulting value at that point in the expression, in a REAL PRECISION (15) data-elenent, for further expression evaluation.

Note that the integer division will not return any remainder, the MOD operator must be used.

The ** operator, for exponentiation, may have a first operand data-element of integer or real data type. The second operand data-element can only be an integer data type.

The binary operators, MOD and SHIFT, must have integer, or integer modified, operands only. They both give a resulting value in an integer data-element.

The SHIFT operator will shift bits in the first operand data-element. The second operand specifies the number of bit positions to be shifted and if this operand is positive, then the shift is to the left, negative means shift to the right. If the first operand data-element is a signed integer data type, then the sign bit is not affected by left shifts and it is extended for right shifts. If the first operand data-element is an unsigned data type, ie. a non-negative integer range, then zeroes are shifted in from the left in right shifts, and they are shifted in from the right for left shifts.

For example :

```
        77B SHIFT 3
```

gives a resulting value 770B.

The MOD operator gives a resulting value of the first operand modulo the second operand, ie. the remainder after dividing the first operand value by the second operand value.

For example :

```
        27 MOD 5
```

gives a resulting value of 2, ie. remainder of 27/5,

```
        -27 MOD 5
```

gives a resulting value of -2,

```
        27 MOD -5
```

gives a resulting value of 2,

```
        -27 MOD -5
```

gives a resulting value of -2.

Examples of the use of the arithmetic operators :

1. x+y will form the sum of x and y.

2. x-y will subtract y from x.

3. x+y+z will sum together x, y and z.

4. x+y-z will add x and y and then subtract z from the result, see note below.

5. x*y/z will multiply x and y before dividing the result by z, see note below.

6. x/y*z will divide x by y first, and then multiply the result by z.

7. x*y+z will multiply x and y and add z to the result.

8. x+y*z will multiply y and z and add x to the result. The order is determined by the different priorities, * is 9 and + is 8.

9. -x**2 Since the operator ** has a higher priority, 11, its operands will be combined first. Thus the expression will be interpreted as :
-(x**2).

If the operator priorities do not give the desired order of evaluation, then parts of an expression may be enclosed in parentheses. Parts thus enclosed are evaluated as a whole expression before being used as an operand.

For example :

1. x+y/z will cause division of y by z before adding x to form the result, because of operator priority.

2. (x+y)/z will ensure that x and y are added, and then that result will be divided by z.

3. (x+y)/(x+z) here x+y and x+z will be computed separately and subsequently, the former result will be divided by the latter. Note that either x+y or x+z may be evaluated first.

While the operators +, -, *, / and ** represent the usual mathematical operations, one must be aware that the underlying computing hardware has fixed limits to the precision and accuracy of representation of values and the results of operations. These limits are described in Appendix C .

Note : The order of operations on computer hardware is such that the result would be mathematically exact if the hardware were mathematically precise. If a particular order of operations is vital for numerical accuracy, it is best to use parentheses to force the order.

For example :

1. x+y+z represents the sum of x, y and z. The computation may add x to y and then add z, or it may add y to z and then add x.

   But,

2. (x+y)+z will ensure that x and y are added together, before z is added to the result.

5.3 LOGICAL_OPERATORS

PLANC has logical operators which are available for operands whose data-elements are of the integer, boolean or set data types. There are both unary and binary logical operators. The operands for a binary operator must both be either integer, boolean or set, but the operands may vary in the declared modifications, ie. range for integer.

The following table lists all the available logical operators :

Operator		Priority	Operation	Data types
AND	binary	3	logical and	integer,boolean,set
OR	binary	2	inclusive or	integer,boolean,set
XOR	binary	2	exclusive or	integer,boolean,set
NOT	unary	4	logical negation	integer,boolean,set
ABS	unary	11	cardinal number	set

The binary operators, the AND operator, the OR operator and the XOR operator, and the unary NOT operator can have operand data-elements of either integer, boolean or set data types. Further, modified integers may be used as operands. Integer range and modified integer operands may be mixed when used with the binary operators.

The resulting value will be of the same data type as the operands. If the operands are different modifications of integer data type, then the resulting value will be an integer data-element appropriate to hold the larger range of the two modified integer operand data-elements.

The ABS operator will give as a resulting value, the maximum number of members declared for the operand set data-element. The resulting value will be an integer data-element.

It should be noted that the evaluation rules described, are for explanatory purposes so that an expression can be correctly interpreted. However, the actual order of interpretation is not fixed so long as the result is mathematically and logically equivalent. Indeed it can happen that part of an expression is not evaluated at all. For example :

```
IF ( i=1 OR 1.5+i=:r>10.1 ) THEN ...
```

in which, if i has the value 1, then the expression in parentheses is known to have the value TRUE after testing i for 1. Further, no value will be stored into r during evaluation of the expression in parentheses.

The resulting value of expressions involving the above operators, with boolean operand data-elements :

b1	NOT b1
TRUE	FALSE

b1	b2	b1 AND b2
TRUE	TRUE	TRUE
TRUE	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	FALSE	FALSE

b1	b2	b1 OR b2
TRUE	TRUE	TRUE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

b1	b2	b1 XOR b2
TRUE	TRUE	FALSE
TRUE	FALSE	TRUE
FALSE	TRUE	TRUE
FALSE	FALSE	FALSE

If these operators are used with integer operand data-elements, then the operator will be applied to all bits in the entire integer data-element, where a bit value 1 is interpreted as TRUE and 0 as FALSE.

If these operators are used with set operand data-elements, the operators will carry out the usual mathematical operations on the sets.

Examples of the use of logical operators :

```
         INTEGER : int1:=12B,int2:=14B
    %
         .. NOT int1 ..          % resulting binary value is ..10101
    %
         .. int1 AND int2 ..     % resulting binary value is ..01000
    %
         .. int1 OR  int2 ..     % resulting binary value is ..01110
    %
         .. int1 XOR int2 ..     % resulting binary value is ..10110
```

Examples of sets and logical operators :

```
    % declare some sets
       TYPE colour = ENUMERATION (red,green,blue,pink,ash,  &
                                  yellow,white,black)
       colour SET : bright:=(red,green,yellow,pink),anycolour,fool
       colour SET : pastel:=(blue,yellow,pink)
       INTEGER : int1
    % inclusive or - result is red, green, yellow, pink, blue
       bright OR pastel =: anycolour
    % logical and - result is yellow, pink
       pastel AND bright =: anycolour
    % exclusive or - result is red, green, blue
       bright XOR pastel =: anycolour
    % logical negation - result is blue, ash, white, black
       NOT bright =: fool
    % set maximum number of members - result is 8
       ABS bright =: int1
```

5.4 RELATIONAL OPERATORS

PLANC has relational operators which are available for operands whose data-elements are of the integer, real, enumeration, pointer and set data types. There are only binary relational operators.

The following table lists all the available relational operators :

Operator		Priority	Operation	Data types
=	binary	6	equal	integer,real,set, enumeration,pointer
><	binary	6	not equal	integer,real,set, enumeration,pointer
>=	binary	6	greater than or equal	integer,real,set, enumeration,pointer
<=	binary	6	less than or equal	integer,real,set, enumeration,pointer
>	binary	6	greater than	integer,real,set, enumeration,pointer
<	binary	6	less than	integer,real,set, enumeration,pointer
IN	binary	5	membership	integer,set, enumeration,pointer

All relational operators, except IN, must have both operand data-elements of the same data type. Operand data-elements of integer or real data types may be modified, ie. integer range or real precision, and modified integer or real data type operand data-elements may be mixed when used with the binary relational operators.

If the IN operator has a first operand data-element of integer, enumeration or pointer data types, then the second operand is a list of data-elements of the same data type as the first operand. This list may contain explicit literals, constant identifiers, identifiers, expressions to be evaluated at run-time or implied ranges of the correct data type. If the IN operator has a second operand data-element of the set data type, then the first operand must be a possible member value of the set, which may be evaluated from an expression at run-time.

The resulting value from evaluation of any relational operator will be stored in a boolean data-element.

Examples of the use of relational operators :

```
    INTEGER : int1
    INTEGER RANGE (0:200) : int2
%
    54=:int1
    .. int1 >= 0 ..      % resulting value TRUE
%
    20000=:int1; 5=:int2
    .. int1 < int2 ..    % resulting value FALSE
%
    21=:int1; -3=:int2
    .. int1*int2 = 0..   % resulting value FALSE
%
    5=:int1; 10=:int2
    .. int1-1 IN 1,3,5,int2 ..    % resulting value FALSE
    .. int1-2 IN 1:100,2*int2 .. % resulting value TRUE
%
    REAL : rl1
    REAL PRECISION (9) : rl2
    1.5=:rl1; 3.7=:rl2
    .. rl1 >< rl2 ..     % resulting value TRUE
%
    ENUMERATION (pink,blue,bottle,red) : mycolor,yourcolor
    red=:mycolor; blue=:yourcolor
    .. mycolor > yourcolor .. % resulting value TRUE
    .. bottle IN mycolor,yourcolor.. % resulting value FALSE
    .. mycolor IN blue:red   .. % resulting value TRUE
%
    INTEGER ARRAY : vectorlist (1:100)
    INTEGER ARRAY POINTER :                                  &
      listhead:=ADDR(vectorlist( MININDEX(vectorlist,1) )),  &
      listtail:=ADDR(vectorlist( MAXINDEX(vectorlist,1) ))
    .. listhead = listtail .. % resulting value FALSE
%
    INTEGER RANGE (1:100) SET : odds:=(1,3,5,7,9)
    .. 1+3 IN odds ..    % resulting value FALSE
```

5.5 CONVERSION BETWEEN DATA TYPES

The rules for forming expressions in PLANC restrict the way data types may be used, especially for moving and storing data-element values of a particular data type. Sometimes it may be necessary to move a value into a data-element of a different data type or simply convert between different data types, eg. integer to real. While good programming practices generally try to avoid this sort of operation, care should be taken if it is necessary to use this sort of operation. The following Standard Routines are provided in the PLANC language :

1) CONVERT - convert between the various integer and real data types.

2) FORCE - take the value from one data-element, and store it into another data-element of a different data type to the first, but of exactly the same size.

These standard routines give a value in a temporary resulting value data-element, ie. the routine out-value, which should be stored with one of the assignment operators.

The general form of the routine invocations are :

```
     identifier CONVERT data-type
  or identifier FORCE   data-type
```

where

identifier — is an identifier whose data-element value is to be converted.

data-type — is the data type of the data-element into which the value is to be stored.

The CONVERT routine may be used for a data type conversion with an assignment operator to simply store the value.

For example :

```
     INTEGER : int
     REAL : rl
     12=:int
  % convert an integer value to real value
     int CONVERT REAL =:rl
  % use conversion within expression
     3.0+2.0*(int CONVERT REAL)=:rl
  % note, parentheses not required, but they help visually
```

The FORCE standard routine may be used with any mixture of simple, composite, predefined or user specified data types.

For example :

```
         TYPE colour = ENUMERATION (red,pink,blue)
         INTEGER : intl
         INTEGER : int
     % put an integer value into a real pointer data-element
         int FORCE REAL POINTER ...
     % for some bizarre reason the following might be done!
         12=:int1
         int1 FORCE colour ...
```

Note that the data-element data type to receive the value from the FORCE standard routine should be exactly the same size as the originating data-element. For example :

```
         INTEGER1 : int
         int FORCE REAL ...
```

will give unpredictable results. A compile-time message will occur.

The FORCE standard routine must be used with great care. The internal representation of the data types involved must be known, see Appendix C, otherwise results may be unpredictable after use of the FORCE routine.

6 SEQUENCE CONTROL STATEMENTS

The executable statements discussed so far will be executed strictly in the sequence that they appear in the source program. PLANC has a number of statements which will unconditionally or conditionally change the sequence of statements to be executed or cause a group of statements to be executed repeatedly under some form of iteration control. The sequence control statements available are :

1) GO - unconditional change of sequence.

2) IF - conditional change of sequence.

3) CASE - multi-choice conditional change of sequence.

4) DO - repetitive execution, of a group of statements.

5) FOR - repetitive execution, of a group of statements, a specified number of times.

6) WHILE - repetitive execution, of a group of statements, until a condition is satisfied.

7) ASSERT - run-time error occurs if a specified condition is not true.

6.1 GO STATEMENT

The 'GO' statement unconditionally transfers control to another statement within a routine. The general form of a GO statement is :

```
GO label-identifier
```

where

label-identifier   is a label, declared within the scope of this GO statement.

Note, for a full description of 'scope of identifier' rules, see section 7.8 .

Beware that control transfers into structures such as FOR - ENDFOR or DO - WHILE - ENDDO loops may have unpredictable results.

Example of the use of a GO statement :

```
% declarations
    INTEGER : int1
    LABEL : lab1,lab2,lab3
%
% executable program
%
 lab1 : 1=:int1        % any executable statement
%
          GO lab1         % transfer to statement 'lab1'
```

6.2 IF STATEMENT

The 'IF' statement will conditionally execute one or more groups of executable statements. The groups of statements executed in this manner may contain further 'nested' IF statements. The general form of an IF statement is :

```
IF expr THEN
    ex-stmts
[ELSIF expr THEN
    ex-stmts]...
[ELSE
    ex-stmts]
ENDIF
```

where

expr is an expression with a boolean resulting value.

ex-stmts is a group of executable statements.

If expression immediately following the IF gives a value TRUE, the group of statements immediately following the THEN will be executed, and then control will be transferred to the statement immediately following the ENDIF.

If this expression gives a value FALSE then :

- if there is neither an ELSIF nor ELSE present, control will pass to the statement following the ENDIF,

- if the IF - ENDIF contains any ELSIF's, the expression immediately following each ELSIF will determine whether its THEN group of statements is to be executed or not. This process will continue for each next ELSIF, for each expression which gives a value FALSE. If a THEN path is taken, the control will pass to the statement following the ENDIF after that group of statements has been executed,

- if the IF - ENDIF contains an ELSE, control passes to the group of statements following the ELSE only if the expressions of the IF and those of any ELSIF's present, all give the value FALSE.

Examples of IF statements :

1. A simple IF - THEN.

```
% test for a full page
    IF currentline+lines > linesperpage THEN
% yes, start a new page
        newpage
        0=:currentline
        printheading
    ENDIF
```

2. An IF - THEN - ELSE.

```
% adjust wages for tax
    IF taxed THEN
% yes, reduce payment by tax amount
        gross - tax(gross)=:nett
    ELSE
% no, pay full amount
        gross=:nett
    ENDIF
```

3. An IF - THEN - ELSIF - ELSE

```
% compute area of a many-sided figure
    IF sides = 3 THEN
% area of a triangle
        (a+b+c)/2.0=:s
        sqrt( s*(s-a)*(s-b)*(s-c) )=:area
    ELSIF sides = 4 THEN
% area of a rectangle
        a*b=:area
    ELSE
% approximate other figures by the area of a circle
        pi*(radius**2)=:area
    ENDIF
```

4. Nested IF's.

```
% check document signatures
   IF amount > 10000 THEN
% large amount, check number of signatures
       IF signatures < 2 THEN
% reject
           setnogood
       ELSE
% large amount check
           bigcheck
       ENDIF
   ELSIF amount > 100 THEN
% medium amount check
       midcheck
   ENDIF
% if passed, pay it
   IF chequeok THEN
       payit
   ELSE
       chequeerror
   ENDIF
```

6.3 CASE_STATEMENT

The 'CASE' statement will select one of a number of groups of executable statements to be executed. During one execution of a CASE statement, only one of the groups will be executed and the remaining groups will be skipped. The selection of a particular group of statements is by the CASE expression whose value must correspond to the integer or enumeration data type values used in the INCASE parts of the CASE statement. The general form of the CASE statement is :

```
CASE expr
  INCASE value-list
    ex-stmts
  [INCASE value-list
    ex-stmts]...
  [ELSE
    ex-stmts]
ENDCASE
```

where

expr — is an expression with a resulting value data type, corresponding to the data type of the INCASE value-lists.

value-list — is a list of integer or enumeration literal values. Note : it may be expressed as an implied range.

ex-stmts — is a group of executable statements.

The values in each INCASE part must all be of the same data type as expr. Each value which occurs in an INCASE part, must not occur more than once in all of the value-list's of the entire CASE statement.

The group of statements following the ELSE will be executed if the value of the expression is valid but does not appear in any INCASE value-list. If the value-lists do not contain all possible values, an ELSE must be present.

If the value of the expression is invalid, eg. outside a defined integer range, control will be transferred to the statement immediately following the ENDCASE, ie. the CASE statement will be skipped, unless an ELSE part is present. If an ELSE part is present, the group of statements following the ELSE will be executed.

> Note : if the values belong to an INTEGER RANGE, the lower bound of the INTEGER RANGE must be 0. The values actually checked currently are 0 and the nearest higher power of 2 to the upper bound.

Examples of CASE statements :

```
          TYPE days = ENUMERATION (monday,tuesday,wednesday,     &
                         thursday,friday,saturday,sunday)
          days : thisday
    %
          CASE thisday
            INCASE saturday
               shopping
            INCASE sunday
               dayofrest
            INCASE monday : thursday
               workdays
          ELSE
     % control comes here only for the value friday
             leftovers
          ENDCASE
```

6.4 DO_STATEMENT

The 'DO' statement may be used to repetitively execute a group of statements with no control of the number of the repetitions or of the termination condition to exit from such a loop. The general form of a DO - ENDDO loop is :

```
          DO
            ex-stmts
          ENDDO
```

where

ex-stmts is a group of executable statements.

The group of statements will be executed repeatedly. At least one GO statement must be in the group of statements to leave the loop under some condition. If not the program will contain an infinite loop.

Example of a DO - ENDDO loop :

```
          REAL start:=1.0,increment:=0.1,limit:=2.0,value
          LABEL : next
    % loop through a series of fractional values
          start=:value
          DO
    % use 'value' for computation
    %
    % test for end of loop
            increment+value=:value
            IF value > limit THEN
                GO next
            ENDIF
          ENDDO
    next : ....
```

6.5 FOR STATEMENT

The 'FOR' statement will cause repeated execution of a group of statements bounded by the FOR and ENDFOR. The number of repetitions is specified during execution just prior to entering a FOR - ENDFOR loop for the first time. The group of statements may be executed the specified number of times or perhaps fewer times if some exceptional condition arises during the repetitive execution. The general form of the FOR - ENDFOR loop is :

```
FOR control-ident IN [REVERSE] list DO
  ex-stmts
[EXITFOR
  ex-stmts]
ENDFOR
```

where

control-ident is an identifier whose data type must correspond with that of the 'list' values.

list is a list of data-elements of INTEGER, ENUMERATION, ARRAY or POINTER data type.

ex-stmts is a group of executable statements.

The control identifier will take the values of the 'list' in the sequence that they have been specified. The control identifier is available within the loop but care must be taken if its value is changed, as this may interfere with orderly control of the loop. Upon exit from a FOR - ENDFOR loop, the control identifier will have an unpredictable value. This applies as soon as the loop exit action begins, namely if an EXITFOR is present, the control identifier value will not have a predictable value on entering the EXITFOR group of statements.

The list of the FOR - ENDFOR loop is an implicit or explicit list of values which will determine the number of repetitions of the loop. The list may comprise :

- Integer, Enumeration or Pointer data-elements which may be literal expressions or expressions evaluated at run-time. The control identifier must be of the same data type. Expressions are evaluated at run-time within the loop initialisation so that modifying identifiers used in such an expression during execution of the loop will have no effect on the control of the loop.

- An implied range, of type Integer or Enumeration, may be used for any elements of such a list or for the whole list. The upper and lower bounds of an implied range, which must be evaluated at run-time, will be computed during loop initialisation - as is the case for explicit data-elements. However, when using an implied range, altering the value of the control identifier during execution of the loop may affect the loop control, see paragraph on loop testing below.

- The list may contain one or more single-dimensioned array data-elements. In this case the control identifier must be an integer data type, which will take the successive values of the index sets of the specified arrays in the list.

  The control identifier may also be a pointer data-element of the same base data type as the elements of the arrays specified in the list. However, a pointer must not be used for the control identifier if the array has been declared with the PACKED option, and the elements of the array require less storage than the smallest addressable unit on a particular machine, eg. on the ND-100 an array whose elements were declared as INTEGER1 PACKED would produce unpredictable results. Further, if the control identifier is an pointer data-element, only one array is permitted in the list.

- The list may contain one or more Pointer Implied Ranges. This is used to step through some records in a linked list, see section 4.6 .

The keyword REVERSE, if present, applies to each implied range in the list, with the exception of Pointer implied ranges. It will cause the loop control to begin with the second value (the last value as declared) in each implied range and step downwards to the first value of the range. Note that implied ranges must be specified in ascending order. The REVERSE option may not be used with a Pointer implied range.

The keyword REVERSE also applies to any arrays in the list. If the control identifier is either an integer or a pointer data-element, it will begin with the value corresponding to the upper bound of the index set and take successive values until the lower bound of the index set is reached.

A FOR - ENDFOR loop contains a test to check if the required number of repetitions has been completed. This test is done at the end of the loop. Further, if one or more implied ranges is in the list of the FOR statement, then incrementing through the implied range values will also take place at the end of the loop. Note that while stepping through the values of an implied range, if the value of the control identifier is explicitly set greater than or equal to the final value of the range, then that will terminate looping through the values of that particular implied range.

If the list of a FOR - ENDFOR loop contains one or more implied ranges, a further test is placed within the loop initialisation. If the values of the implied range can be computed at compile time, then if the terminal value of the implied range is smaller than the initial value, the entire FOR - ENDFOR loop will be skipped, ie. it will not be executed at all. If the values of the implied range can only be computed at execution time, then a run-time check within the loop initialisation will result in zero repetitions of the loop if the terminal value of the range is smaller than the initial value.

The group of executable statements may include any executable statements but statements such as DO - ENDDO and IF - ENDIF must be entirely contained within the FOR - ENDFOR loop. Loops may be nested to any number of levels provided each loop is entirely contained within an outer level loop. While the number of levels of nesting is theoretically unlimited, the actual number is limited by the memory available to the PLANC compiler.

If an EXITFOR is present, then when all the list values are exhausted, control will be passed to the statement immediately following the EXITFOR.' Following the execution of this group of statements, control will be passed to the statement immediately following the ENDFOR. If an exit from the loop is made by any other means than exhausting the value list, the EXITFOR group will not be entered.

Examples of FOR - ENDFOR loops :

1. A simple loop with explicit integer values.

```
     INTEGER : intcontrol
     FOR intcontrol IN 1,5,15,3,17 DO
 % group of statements - to be executed 5 times
     ENDFOR
```

2. A simple loop with explicit enumeration values.

```
     ENUMERATION (red,pink,blue,grey,brown) : colour
     FOR colour IN pink,grey,red,brown DO
 % group of statements - to be executed 4 times
     ENDFOR
```

3. A simple loop with explicit pointers in the FOR list.

```
     INTEGER POINTER : ptrcontrol,ptr1,ptr2,ptr3
 % put some addresses into ptr1,ptr2 and ptr3
     FOR ptrcontrol IN ptr1,ptr2,ptr3 DO
 % group of statements - to be executed 3 times
     ENDFOR
```

4. A simple loop with implied ranges in the FOR list.

```
     INTEGER : intcontrol
     FOR intcontrol IN 1:10,21,24,51:60,101 DO
 % group of statements - to be executed 23 times
     ENDFOR
```

5. A simple loop with implied ranges, using REVERSE.

```
     INTEGER : intcontrol
     FOR intcontrol IN REVERSE 1:10,21,24,51:60,101 DO
 % group of statements - to be executed 23 times
 % Note : the sequence of values of the control identifier is
 %        10,9,...,1,21,24,60,59,...,51,101
     ENDFOR
```

6. A simple loop, values in FOR list to be evaluated at run-time.

```
     INTEGER : intcontrol,int1,int2,int3
     FOR intcontrol IN int1,int2:int3*2 DO
 % group of statements - to be executed n times,
 % ie. 1+(int3*2-int2+1), evaluated at run-time.
 % intcontrol takes the values int1,int2,int2+1,...,int3*2.
     ENDFOR
```

7. A simple loop with arrays in the FOR list.

```
     INTEGER : intcontrol
     REAL ARRAY : arreal1(1:3),arreal2(1:7)
     FOR intcontrol IN arreal1,arreal2 DO
 % group of statements - to be executed 10 (ie. 3+7) times
 % control identifier takes the values 1,2,3,1,2,3,4,5,6,7
     ENDFOR
```

8. A simple loop, arrays in FOR list, a pointer control identifier.

```
    REAL POINTER : ptrcontrol
    REAL ARRAY : arreal1(1:3)
    FOR ptrcontrol IN arreal1 DO
  % group of statemerts - to be executed 3 times
  % control identifier takes the addresses of the array elements
  % arreal1(1),(2),(3)
    ENDFOR
```

9. A simple loop, pointer implied range in FOR list.

```
  % define a record cata type for the linked list
    TYPE myrecord : RECORD
                     myrecord POINTER : linkptr
                     INTEGER : recnumber
                    ENDRECORD
  % initialise a static linked list of records
    myrecord : r1?,r2?,r3?  % predeclaration of data-elements
    myrecord POINTER : listhead:=ADDR(r1)
    myrecord : r1:=( ADDR(r2),1 )
    myrecord : r2:=( ADDR(r3),2 )
    myrecord : r3:=( NIL,3 )
  % declare a record pointer for scanning the list
    myrecord POINTER : ptrcontrol
  % loop through all records in the linked list
    FOR ptrcontrol IN listhead:linkptr DO
  % group of statements to process one record data-element
    ENDFOR
```

10. A nested loop.

```
    INTEGER : rowelement,colelement
    REAL ARRAY ARRAY : square(1:5,1:5)
    REAL : sum
  % sum elements to the left of the diagonal element
    FOR rowelement IN 1:MAXINDEX(square,1) DO
     0.0=:sum
     FOR colelement IN 1:rowelement-1 DO
     sum+square(colelement,rowelement)=:sum
     ENDFOR
  % store the sum in the diagonal array element
    sum=:square(rowelement,rowelement)
    ENDFOR
```

11. A simple loop with an EXITFOR part.

```
        INTEGER : intcontrol,sum,limit
        BOOLEAN : sumflag
        LABEL : next
        INTEGER ARRAY : vector(1:100)
        0=:sum ; FALSE=:sumflag ; 500=:limit
        FOR intcontrol IN vector DO
          sum+vector(intcontrol)=:sum
          IF sum < limit THEN
            GO next
          ENDIF
        EXITFOR
          IF sum < 0 THEN
            FALSE=:sumflag
          ENDIF
        ENDFOR
    next : ....
```

6.6 WHILE STATEMENT

The 'WHILE' statement may be used within DO - ENDDO or FOR - ENDFOR loops to exit when a condition becomes false. While the condition remains true, the loop control will not be affected. The general form of a WHILE statement used within a loop is :

In a DO - ENDDO loop

```
DO
  ex-stmts
  WHILE expr
  ex-stmts
[EXITWHILE
    ex-stmts]
ENDDO
```

In a FOR - ENDFOR loop

```
FOR control-ident IN [REVERSE] list DO
  ex-stmts
  WHILE expr
  ex-stmts
[EXITWHILE
    ex-stmts
[EXITFOR
  ex-stmts]
ENDFOR
```

where

expr is an expression with a boolean resulting value.

ex-stmts is a group of executable statements.

The effect of the WHILE statement each time it is executed within the loop, is to test if the resulting value of the expression is TRUE. If it is, pass control to the executable statement immediately following the WHILE. If the resulting value of the expression is FALSE, then control will exit from the loop and pass to the statement immediately following the ENDFOR or ENDDO.

If an EXITWHILE is present within the loop, the group of statements following the EXITWHILE will be executed as soon as the loop exit action begins, as a consequence of the relevant WHILE statement. Note however, that if an EXITWHILE and an EXITFOR are both present in a FOR - ENDFOR loop, then an exit from the loop effected by the WHILE condition will execute the EXITWHILE group of statements but not the EXITFOR group of statements, prior to the exit from the loop.

A WHILE statement may be placed anywhere within the group of executable statements of a loop, depending on where a loop exit is desired under the control of a logical condition. Further, any number of WHILE statements may be used within a FOR - ENDFOR or a DO - ENDDO loop.

Examples of use of the WHILE statement :

1. Within a DO - ENDDO loop.

```
         INTEGER : records
         BOOLEAN : endoffile
     % read first record of a file
         0=:records
         openfile
         nextrecord
     % loop through all records in the file
         DO
     % if end of file, exit from loop
           WHILE NOT endoffile
     % process a record
           1+records=:records
     %     .
     % end of the loop statements
     %
     % loop exit condition
         EXITWHILE
     % close file
           closefile
         ENDDO
```

2. A WHILE used to leave a FOR - ENDFOR loop without using a label.

```
         INTEGER : intcontrol
     %
         FOR intcontrol IN 1:100 DO
     %
     % exit from loop under certain conditions
     %
           IF NOT checkvalid THEN
             WHILE FALSE
           ENDIF
     %
     % things are ok, continue looping
     %
         EXITWHILE
         ENDFOR
```

3. Multiple WHILE's within a FOR - ENDFOR loop.

```
        CONSTANT rows:=10,cols:=10
        INTEGER rowelement,colelement
        REAL ARRAY ARRAY : matrix(1:rows,1:cols)
        REAL ARRAY : rowsum(1:rows)
        REAL : limitsum
    % loop through all rows of the matrix
        100.0=:limitsum
        FOR rowelement IN 1:rows DO
          0.0=:rowsum(rowelement)
    % sum the row elements, provided it is within limits
          FOR colelement IN 1:cols DO
            matrix(rowelement,colelement)+rowsum(rowelement)    &
               =:rowsum(rowelement)
    % check sum limits
            WHILE rowsum(rowelement) < limitsum
    % too many elements for sum ?
            WHILE colelement CONVERT REAL < limitsum/4.0
    % in case of abnormal exit, set sum negative
          EXITWHILE
            -1.0=:rowsum(rowelement)
    % end of inner loop
          ENDFOR
        ENDFOR
```

6.7 THE ASSERT STATEMENT

An 'ASSERT' statement requires an associated condition to be true whenever the statement is encountered. The general form of the ASSERT statement is :

```
        ASSERT expr
```

where

expr   is an expression with a boolean resulting value.

During program execution, if the resulting value of the expression is TRUE then control will simply pass to the next executable statement. If however the resulting value of the expression is FALSE, an error condition arises and control will be transferred elsewhere depending on what has been specified for handling 'ASSERT' errors. For further details of how 'ASSERT' errors may be handled see Exceptions and Error Handling, section 6.8. This provides an explicit means for supplementing the normal run-time checks provided by the system.

Examples of ASSERT statements :

```
        ASSERT int1 < number*2
        ASSERT int2 < 1 AND red IN mycolours
```

6.8 EXCEPTION AND ERROR HANDLING

PLANC provides a mechanism for handling specific sorts of error conditions which may arise during program execution. A part of the program, called an 'exception handler', may have control passed to it when the corresponding error condition occurs, rather than continue executing statements in the normal way. The general form of such an exception handler is :

```
ON exception[,exception]... DO
 ex-stmts
ENDON
```

where

exception is any defined exception condition.

ex-stmts is a group of executable statements.

An exception handler may handle errors due to one or more exception conditions. An exception condition will be sensed only, in the source code following the ON statement - ENDON statement group of source statements, within a routine. If more than one ON - ENDON exception handler appears in a routine, then the one immediately preceding the occurrence of an exception, in the source code, will be activated to handle the exception.

The particular exception conditions defined in PLANC are :

ASSERTFALSE - for the expression in an ASSERT statement giving a value FALSE.

OVERFLOW - arithmetic overflow.
Note : hardware checks only activate this exception.

POINTERERROR - attempt to use a data-element, referenced by a pointer whose value is NIL. (not implemented)
The NEW..IN standard routine will trap such errors if the space to be used is not adequate.

RANGEERROR - array index or integer range error.
(not implemented)

ROUTINEERROR - a called routine has taken an ERRETURN exit.

STACKERROR - stack overflow or underflow has occurred, eg. when using the NEW standard routine.

Executing an exception handler is similar to execution of a routine invocation. The ENDON is in this sense equivalent to the RETURN statement, passing control back to the place that exception condition occured.

Note that a ROUTINEERROR exception handler cannot set-up or repair the out-value, or output parameters which would have been passed back by successful execution, after invocation of the routine which generated the exception condition.

If a ROUTINEERROR exception occurs and no exception handler has been provided, an ERRETURN exit from the routine will be simulated. Control will pass back up the invocation hierarchy as described in section 7.5.

Examples of exception handlers :

```
ON ASSERTFALSE DO
  0=:int1
  GO out
ENDON

ON STACKERROR,OVERFLOW DO
  int2 ERRETURN
ENDON
```

Note that the PLANC run-time system has a routine which will be invoked if an ASSERT condition is FALSE and the user has no ON ASSERTFALSE exception handler. The form of the declaration of this routine is :

```
% on the ND-100
    ROUTINE SPECIAL VOID,VOID : assert_handler ALIAS '5FATA'
% on the ND-500
    ROUTINE SPECIAL VOID,VOID : assert_handler ALIAS '#FATA'
```

If a user wishes to replace this routine with another, the user written routine must be loaded before PLANC library routines.

7 ROUTINES

A PLANC routine is group of statements which can be referred to as an entity to carry out a particular function. A routine comprises executable statements and declarations of any identifiers used within the routine. The routine concept in PLANC is defined as a composite data type, whose declaration includes data types of the data-elements to be used in communication between the routine and its caller. A routine has an explicit in-value and out-value which affect the way a routine invocation appears in a calling routine.

A routine may be invoked to carry out a specific function or operation. The PLANC routine is similar to the 'subprogram' concept of other programming languages. However, a PLANC routine has one explicit in-value and one explicit out-value. A PLANC routine may also have a list of formal parameters declared, for transmitting data-elements into or out of the routine. A routine may be invoked from another routine in the same module or a routine in a separate module.

7.1 ROUTINE_DECLARATION

A routine is a composite data type. Consequently, a routine declaration causes the construction of a data-element which includes all the memory area used for the routine, excepting dynamically created data-elements.

A routine declaration will include the following :

1) Options which determine the specific structure of the routine for particular types of routine invocation.

2) The data types of the explicit in-value and out-value of the routine.

3) The data types of any formal parameters used within the routine, which will consequently be required in any call to the routine.

4) The identifier to be used as a routine name for invoking the routine.

5) The identifiers of any formal parameters declared, to be used within the routine.

6) The declarations of local data-elements which will only be available inside the routine.

7) Executable statements which carry out the desired operations required of the routine.

The first five of the above items are called the <u>routine header</u>. The last two items are called the <u>routine body</u>.

The general form of a ROUTINE declaration is :

```
ROUTINE - rest of routine header
  routine body
ENDROUTINE
```

The general form of a ROUTINE header is :

```
ROUTINE [option[ option]...] in-data-type,out-data-type
   [(p-data-types)] : rout-ident [(p-ident-list)]
                          [ALIAS 'a-rout-ident']
```

where

option is one of the ROUTINE modifiers STANDARD, REFERENCE, SPECIAL or INLINE.

in-data-type is the data type of the in-value.

out-data-type is the data type of the out-value.

p-data-types is a list of the data types of the formal parameters of this routine.

rout-ident is the identifier for referring to this routine.

Note : special characters allowed, see below.

a-rout-ident is a text string. It qualifies the routine name to distinguish routines with the same structure, eg. same parameters, but of different data types.

Note : the string may contain any characters. p-ident-list is a list of identifiers of the formal parameters of this routine.

Note, that there is a special form of a routine header, namely for a main PROGRAM routine, see 8.2 .

As an alternative to the normal identifier name formation rules, a routine identifier name may be made up of the following special characters only :

! " $ * + - . / : < = > ? ↑ \ [ ]

Further, such a routine identifier name may be a mixture of these special characters, but the rules concerning number of characters in an identifier still apply, see section 2.11 .

Note : 1. A dollar character ($) cannot begin a routine identifier.

2. A full stop character (.) can only begin a routine identifier.

3. A space character must precede the routine identifier if it begins with one of the above special characters.

Several routines may be declared with the same routine identifier. The PLANC compiler will only accept these routines if they can be distinguished by the data types of the in-value and the parameters. For examples and details of such families of routines, see section 8.4 .

A routine has one distinct in-value data-element and one distinct out-value data-element. The in-value and out-value data-elements may be of any valid data type, ie. simple, composite, predefined or user-defined. Further, if either in-value or out-value data-element is not required for a particular routine, then the keyword VOID may be used to denote the absence of a data-element in the formal routine declaration,ie. in the routine header.

A routine may be declared with any number of formal parameters for communication between the invoker and the routine itself. A parameter may be used to transfer a value into a routine or to transfer a value out of a routine or both. It is generally regarded as an unwise practice to use one parameter for transferring values in both directions. The routine header contains a declaration of the data type of each formal parameter. It also contains the identifier names of each formal parameter which must be used to refer to each parameter within the routine. The data type of each formal parameter may be access modified, see section 3.11.3, with READ or WRITE. The default access for each declared formal parameter is READ. Parameter transfer is discussed in more detail in section 7.4 .

Examples of simple routines :

1. A routine to return the larger of two integer values.

```
        ROUTINE VOID,VOID (INTEGER,INTEGER,INTEGER WRITE) :      &
          simple(in1,in2,outval)
    % in-value and out-value data-elements are absent
    %
    % declarations local to this routine
    %
        INTEGER : local
    % select the larger parameter value
        in2=:local
        IF in1 > in2 THEN
          in1=:local
        ENDIF
    % transfer the larger value back to the invoking routine
        local=:outval
        RETURN
        ENDROUTINE
```

2. A similar routine, using the out-value to return the value.

```
        ROUTINE VOID,INTEGER (INTEGER,INTEGER) : simple(in1,in2)
    %
    % declarations local to this routine
    %
        INTEGER : local
    % select the larger parameter value
        in2=:local
        IF in1 > in2 THEN
          in1=:local
        ENDIF
    % send the larger value back to caller
    % Note that the out-value is part of the RETURN statement
        local RETURN
        ENDROUTINE
```

A routine is normally invoked by use of the routine name identifier in the declaration. However, if a number of routines have the same name and the same number of parameters, eg. an operator myplus may be required to handle various data types, then each routine may be uniquely identified by use of an ALIAS name for access from another module (see Chapter 8, PROGRAM STRUCTURE). Further, any module wishing to use such a family of routines, must IMPORT each one of the family it wishes to use. The IMPORT statements may use either the originally declared routine name identifier or the ALIAS name (see section 8.4), as the routine identifier and whichever is chosen must be used for all routine invocations in that module. This use of ALIAS is necessary to generate adequate information for the Loader to resolve all references correctly. For examples of use of the ALIAS option, see section 8.4. If a module containing a family of routines is to be accessible within a library file, the $LIBRARY-MODE command must be used, see Appendix A.

The name in the ALIAS text string may contain characters which form an identifier which is illegal as a routine name identifier in PLANC or other languages. This facility may be used to create a protection mechanism, for preventing a user program from inadvertently naming and invoking a system routine, which would normally only be invoked by other systems software, eg. the Fortran I/O routines.

A system routine with a (SYSTEM) EXPORT qualifier, will enable other modules to access it, provided that the (SYSTEM) IMPORT qualifier is used, see 8.3 . Then this identifier will be handled by the Loader in the same way as an ALIAS name. For example, most of the Fortran run-time library routines are protected from unintentional invocation by names declared with this protection mechanism. This may be set-up by the use of the EXPORT/IMPORT qualifier, (SYSTEM), or an ALIAS name. Beware, this protection mechanism must be used with the greatest care possible, as it may lead to conflicts with system routines.

Routine declarations may be nested to any number of levels within another routine. However, there are some restrictions on the recursive invocation of routines, see section 7.7 .

The optional routine modifiers specify how the compiler should construct routines with regard to parameter transfer and calling sequence. The following modifiers are available :

1) INLINE - the data-element of such a routine will have no object code generated by the compiler. Each invocation of this routine will have the entire routine data-element instead of the usual call sequence. This will result in a larger program with several copies of the routine. But the program will execute faster as the invocation overheads are not incurred for each use of the routine. INLINE should only be used for small routines, eg. 1 - 5 lines. An INLINE routine cannot be declared or invoked within another INLINE routine.

2) SPECIAL - no routine entry/exit sequence at all is provided. The invocation of such a routine can be made faster than for a normal routine, as the usual register storage and stack initialisation will not be done. Consequently the extra speed might be gainec with a corresponding decrease in security of the environment during the execution of such a routine. This should only be used by the most experienced and knowledgeable users, who may be using assembly code!

3) STANDARD - a calling sequence, including parameter transfer, is generated which is the standard used by Fortran and Cobol to call subprograms. In-values are not allowed. Note that the standard routines, MININDEX, MAXINDEX for array parameters and ERRETURN, are not available in STANDARD routines, either PLANC calling other language routines or vica versa. For examples of the use of such mixed language combinations of routines see Appendix D.

4) REFERENCE - normally parameters whose data-elements are of the simple data types are transferred by value. In a REFERENCE routine all parameter data-elements are transferred by reference, ie. the routine is given the address of each data-element concerned. The calling sequence is not the same as for STANDARD.

While routines are defined as a composite data type in PLANC, the invocation of any routine is treated as an occurrence of an operator. When treated in this manner as an operator, a routine has the priority 11 for the purposes of evaluation of any expressions containing routine invocations. However, if a routine name is the same as any operator defined by the PLANC compiler, eg. +, * or ABS, then this routine will have the same priority as the predefined operator, for the purposes of expression evaluation.

Predeclaration of a routine may be used in the same way as for data-elements of any other data type. An illustration of this facility is in section 3.16 .

A pointer data-element may be declared to reference a routine data-element. If this is done, then the pointer data-element and the IND standard routine may be used to invoke the routine. Note that the IND standard routine may only invoke routines in the outer level of a module, see section 7.9.

For example :

```
% define a data type for a sort of routine
   TYPE myroutine = ROUTINE VOID,VOID
% declare a routine data-element
   myroutine : myfirst
%     .
   ENDROUTINE
% declare a pointer for the defined routine data type
   myroutine POINTER : mypointer
%
% executable statements
%
% set-up the address of the routine data-element
%
   ADDR myfirst =:mypointer
% invoke the routine
   IND mypointer
```

7.2 IN-VALUE AND OUT-VALUE OF ROUTINES

PLANC routines have one explicit in-value data-element and one explicit out-value data-element. Either the in-value or out-value may not be required for a specific routine declaration and the keyword VOID denotes the absence of a data-element.

If an in-value is present in the routine declaration, then executable statements within the routine can refer to the in-value data-element by using the Commercial At character (@). The @ character may be looked upon as the identifier associated with the in-value data-element and of the same data type.

For example :

```
     ROUTINE REAL,VOID (REAL WRITE) : donothing (giveitback)
  % simply return the in-value in the parameter
     @=:giveitback
     ENDROUTINE
```

If the in-value is a composite data type, eg. a record, then the @ character will precede the normal way of referencing a component of the data-element of the composite data type.

For example :

```
     TYPE myrecord = RECORD
                      INTEGER : field1
                      REAL : field2
                     ENDRECORD
  %
  % routine declaration
  %
     ROUTINE myrecord,VOID (INTEGER WRITE) : simplertn (out)
  % return twice the value of the first field in the record
     @.field1*2=:out
     ENDROUTINE
```

Now some code to invoke the routine :

```
     INTEGER : outparam
  % declare a record data-element with initial values
     myrecord : rec1:=(10,23.5)
  % invoke a routine, passing it a record data-element
     rec1 simplertn (outparam)
```

If routines are nested, the Commercial at character (@) refers to the in-value of the inner most routine with respect to the place where the @ is used.

The out-value data-element of a routine will have a value stored into it when a RETURN statement is executed to terminate a routine, see section 7.5. If an expression precedes the RETURN, then the resulting value from evaluation of the expression must be of the correct data type to match the routine declaration. This will be checked at compilation time.

For example :

```
          ROUTINE REAL,REAL : twice
      % double the in-value and put it into the out-value
          REAL : localreal
          @*2.0=:localreal
          localreal RETURN
          ENDROUTINE
```

This could also coded in the following way :

```
          ROUTINE REAL,REAL : twice
      % double the in-value and put it into the out-value
          REAL : localreal
          @=:localreal
          2.0*localreal RETURN
          ENDROUTINE
```

But the simplest way of all is :

```
          ROUTINE REAL,REAL : twice
      % double the in-value and put it into the out-value
          2.0*@ RETURN
          ENDROUTINE
```

Note that in-value and out-value declarations for composite data-elements will result in transfer by reference during execution of a routine invocation, ie. only an address is passed not the entire data-element.

7.3 ROUTINE INVOCATION

A routine will be invoked, by simply executing a statement containing the identifier in the routine declaration. If the routine declaration has an in-value, then the identifier immediately preceding the routine invocation, will indicate the data-element to be used as the in-value. If the routine declaration includes parameters, then the actual parameters in the source program, may be expressions or identifiers separated by commas, enclosed in parentheses, immediately following the routine invocation. For each formal parameter in the routine declaration, there must be an actual parameter in the routine invocation, ie. a data-element, of the formal parameter's data type in the routine declaration. If not, the compiler will give an error message.

For example :

```
    % program code to invoke a routine
        INTEGER : invalue,p2actual
        REAL : p1actual
    % invoke routine with an in-value and 2 actual parameters
        51=:invalue ; 5.35=:p1actual
        invalue artn (p1actual,p2actual)
    % use value returned from routine in the 2nd actual parameter
        p2actual=:localint     % value returned = 1
```

The following routine declaration can be invoked by the above code,

```
        ROUTINE INTEGER,VOID (REAL,INTEGER WRITE) : artn(fp1,fp2)
    % set 2nd parameter : 1, in-value and 1st parameter +ve
    %                     2, not(in-value and 1st parameter +ve)
    %
        IF @>0  AND fp1>0.0 THEN
          1=:fp2
        ELSE
          2=:fp2
        ENDIF
    %
        RETURN
        ENDROUTINE
```

Note that if the actual parameter is of the same base type, but a different modification to the formal parameter, eg. INTEGER4 actual parameter and formal declaration is INTEGER1, then during execution precision may be lost, depending on the value held in the actual parameter.

If the parameter list of a routine declaration comprises only one formal parameter, then the parentheses may be omitted for any invocation.

If the routine invocation is within an expression, then the evaluation will proceed by the normal rules, see Chapter 5 EXPRESSIONS - FORMATION AND EVALUATION, with the routine invocations being treated as operators with priority 11. The resulting value from evaluation of an expression may become the in-value for the routine invocation, by the use of parentheses.

If the routine declaration includes an out-value, and the routine invocation is within an expression, then the out-value returned from the routine invocation will be used for the further evaluation of the expression.

Note that if a routine is declared with an in-value and an out-value, and it is invoked in an expression in the following way, ie. with an assignment operator immediately before and after the routine invocation :

```
        ... i =: rtn =: j
```

then the value of i will be the value stored in j, not the out-value of the routine invocation.

Invocation of a routine within another routine, ie. nested routine invocations, must not be carried out by the use of the IND standard routine.

Examples :

1. A routine invocation within an expression.

```
     % program code to invoke a routine
         INTEGER : localint,invalue,p2actual
         REAL : p1actual
     % invoke routine with an in-value 5, and 2 actual parameters
         5=:invalue ; 5.5=:p1actual
         2+invalue artn (p1actual,p2actual)+3=:localint
     % evaluation becomes 2+1+3, ie. localint=6
```

Routine declaration, to be invoked as above

```
         ROUTINE INTEGER,INTEGER (REAL,INTEGER WRITE) : artn(fp1,fp2)
     % set 2nd parameter - 1, in-value and 1st parameter +ve
     %  and out-value      2, not(in-value and 1st parameter +ve)
     %
         IF @>0  AND fp1>0.0 THEN
           1=:fp2
         ELSE
           2=:fp2
         ENDIF
     % set out-value equal to 2nd parameter
         fp2 RETURN
         ENDROUTINE
```

2. A routine invocation with an expression as an in-value.

```
    % program code to invoke a routine
        INTEGER : localint,int,p2actual
        REAL : p1actual
    % invoke routine with an in-value -4, and 2 actual parameters
    % note, first actual parameter is an expression
        5=:int ; 5.5=:p1actual
        (int-9) artn (2.0*p1actual,p2actual)+3=:localint
    % evaluation becomes (-4) artn (...) +3
    %           then    2+3, ie. localint=5
```

Routine declaration, to be invoked as above

```
        ROUTINE INTEGER,INTEGER (REAL,INTEGER WRITE) : artn(fp1,fp2)
    % set 2nd parameter - 1, in-value and 1st parameter +ve
    %  and out-value      2, not(in-value and 1st parameter +ve)
    %
        IF @>0  AND fp1>0.0 THEN
          1=:fp2
        ELSE
          2=:fp2
        ENDIF
    % set out-value equal to 2nd parameter
        fp2 RETURN
        ENDROUTINE
```

Routines will have functionally different characteristics depending on the presence or absence of an in-value and an out-value data-element. The invocation of a routine will have distinct form for each of the four different possible in-value and out-value configurations.

in-value absent, out-value absent

A routine with no in-value or out-value data-element will be invoked by an executable statement containing nothing other than the routine name, and actual parameters if any have been declared. Since there is no out-value, the routine must terminate an expression. Since there is no in-value, the routine can be preceded by nothing in an expression.

Such an executable statement will carry out a well-defined operation. Communication of values into and out of the routine can only be accomplished by use of routine parameters. This appears like the Subroutine construct of languages such as Fortran or Cobol. In fact, this form of a routine used in conjunction with the STANDARD routine modifier, will create a routine which is callable from a Fortran or Cobol program, and behave like a subroutine.

For example :

```
          ROUTINE VOID,VOID (REAL,REAL,REAL WRITE) :        &
                                                 add(add1,add2,sum)
     % routine which behaves like a subroutine, eg. Fortran
     % add the first two parameters and return the sum in the third
          add1+add2=:sum
          RETURN
          ENDROUTINE
     %
     % code to invoke the 'subroutine' routine
     %
          REAL READ : first:=5.3,second:=6.7
          REAL : total
     %
          add(first,second,total)    % total = 12.0
     % invocation stands alone as an executable statement
```

in-value absent, out-value present

A routine with an out-value, but no in-value will be invoked as part of an executable statement which contains an expression to be evaluated. In the expression containing such a routine invocation, the routine name plus optionally, a parameter list, may be looked upon as an identifier which will have a definite value during evaluation of the expression. Even though the routine is technically an operator with priority 11, a routine of this nature behaves like an identifier with an associated data-element. These characteristics, used with READ only parameters, are similar to a Fortran function subprogram. In fact, this form of a routine with READ only parameters used in conjunction with the STANDARD routine modifier, will create a routine callable from Fortran, and behave like a function subprogram.

For example :

```
    ROUTINE VOID,INTEGER (INTEGER) : twice(invalue)
% routine which behaves like a function, eg. Fortran
% return double the value input
    2*invalue RETURN
%
% invoke the above routine within an expression
%
    INTEGER : int
    5+twice(3)+4=:int   % result is 5+6+4=15
```

in-value present, out-value absent

A routine with an in-value, but no out-value may be invoked within an expression. Since the routine has no out-value, it must terminate the expression. Such a routine will simply store the in-value it receives.

A routine of this form is sometimes referred to as a 'store-into subroutine'. It may be used to store a value into a data structure, while completely separating the actual details of the data structure from the program using the data structure.

For example :

```
  MODULE tables
  EXPORT inentry
% global table and table pointer, to be stored over successive
% routine invocations
  INTEGER : tablepointer
  INTEGER ARRAY : table(1:100)
%
  ROUTINE INTEGER,VOID : inentry
% add another value to the table
%   .
  ENDROUTINE
  ENDMODULE
%
% module to use the table via the above routine - could be
% separately compiled
%
  MODULE usetable
  IMPORT ( ROUTINE INTEGER,VOID : inentry )
  INTEGER ARRAY : stack(0:1000)
%
  PROGRAM : doit
  INTEGER : int1,int2,int3
% executable program - compute value and store in the table
  INISTACK stack
  (int1+int2*int3) inentry
%
  ENDROUTINE
  ENDMODULE
```

For details of MODULE's and EXPORT/IMPORT statements, see sections 8.5 and 8.3 .

The user program can now put values into a table, but does not see the structure of the table. Indeed the MODULE tables, could be recoded to store the table entries in a linked list of RECORD data-elements, and the MODULE usetable would require no change. A matching routine to return a table entry could be written. This routine should have no in-value and an out-value. Then the pair of routines together, could be thought of as a composite data-element, eg. a table with certain characteristics, whose actual implementation details are completely separated from a user of the data-element.

in-value present, out-value present

A routine with both an in-value and an out-value may be invoked within an expression. Since an invocation of such a routine is preceded by a data-element, and returns a data-element, it will represent an operator within the expression if the routine is declared with one parameter. Note, that routine invocations have priority 11, ie. higher than most operators.

A routine of this form is sometimes referred to as a 'store-into function'. It can be used to create operators, analogous to existing operators, eg. + or - for existing data types, eg. BOOLEAN. Further, operators may be created for newly defined data types, eg. operators for a newly defined 'complex' data type.

For example :

```
  % the following two modules must be nested to be able to import
  % the newly defined data type "complex"
    MODULE complexoperators
    TYPE complex = RECORD
                     REAL : realpart,imagpart
                   ENDRECORD
    EXPORT +!,*!
  % add two complex data-elements
  % formula used is (a+i.b)+(c+i.d)=(a+c)+i.(b+d)
    ROUTINE complex,complex (complex) : +!(follow)
    complex : local
    @.realpart+follow.realpart=:local.realpart
    @.imagpart+follow.imagpart=:local.imagpart
    local RETURN
    ENDROUTINE
  % multiply two complex data-elements
  % formula used is (a+i.b)*(c+i.d)=(ac-bd)+i.(ad+bc)
    ROUTINE complex,complex (complex) : *!(follow)
    complex : local
    @.realpart*follow.realpart-@.imagpart*follow.imagpart  &
       =:local.realpart
    @.realpart*follow.imagpart+@.imagpart*follow.realpart  &
       =:local.imagpart
    local RETURN
    ENDROUTINE
 %
 % a nested module - to use the complex data type
 %
    MODULE usecomplex
    IMPORT complex
    IMPORT ( ROUTINE complex,complex (complex) : +! )
    IMPORT ( ROUTINE complex,complex (complex) : *! )
 %
    INTEGER ARRAY : stack(0:1000)
 %
    PROGRAM : docomplex
    complex READ : cpx1:=(1.0,2.0),cpx2(3.0,4.0)
    complex : cpx3
 %
    INISTACK stack
 % add two complex data-elements
    cpx1 +! cpx2 =: cpx3          % result is 4+6i
 % multiply two complex data-elements
    cpx1 *! cpx2 =: cpx3          % result is -5+10i
    ENDROUTINE
    ENDMODULE           % end of usecomplex
    ENDMODULE           % end of complexoperators
```

Note, the data type complex must be IMPORT'ed into a nested module from an outer module.

7.4 PARAMETER_TRANSFER

A routine declaration will declare the data types of any formal parameters to be used by the routine. Any invocation of a routine must include actual parameter data-elements of data types corresponding to those of the declared formal parameters. Parameters of the simple data types are transferred in a different way to parameters of the composite data types.

The simple data types are transferred by value. This means that a routine invocation results in the value stored in the actual parameter data-element being copied into a temporary data-element, created locally in the routine's memory area. During execution of the routine, all references to the formal parameter will operate on the temporary, locally created data-element. The default access mode for parameter data-elements is READ only. The transfer of the actual parameter data-element value to the temporary local data-element, takes place before execution of the routine begins. If WRITE, or READ WRITE has been declared as access mode, during execution the routine may store a value in a formal parameter for return to the invoking routine. Such a value will be transferred to the actual parameter data-element, from the temporary local data-element, after a normal exit from the routine. Such transfers will not take place if an abnormal routine exit occurs, see 7.5. If WRITE only has been declared as access mode, then the temporary local data-element will have an undefined value at the beginning of execution of the routine. Further, any invocation of a routine with any WRITE only parameters, must have explicit actual parameter data-elements for such parameters. Expressions are invalid as actual parameters for such declared WRITE only formal parameters, as they have only a temporary data-element for the resulting value of expression evaluation.

Note that for any routine declared with the routine qualifiers REFERENCE or STANDARD, and an array as a parameter, an invocation of this routine should only use as an actual parameter, an array with the lower bound of each dimension declared as zero. Otherwise the array elements will not be referenced correctly within the invoked routine.

Examples of parameter transfer :

1. A parameter, default READ only access mode.

```
    ROUTINE VOID,INTEGER (INTEGER) : twice (param1)
% param1 refers to the temporary local data-element which has
% received the value of the actual parameter data-element on
% entry to the routine
    2*param1 RETURN
    ENDROUTINE
%
% code to invoke the above routine
%
    INTEGER : int1,int2
%
    twice(5)=:int1       % result is 10
% an expression as the actual parameter
    twice(3+2*4)=:int1  % result is 22
% invocation cannot change value in data-element of int1
    2=:int1
    twice(int1)=:int2   % result is 4
```

2. A parameter, with READ WRITE access mode.

```
    ROUTINE VOID,VOID (INTEGER READ WRITE) : twice (param1)
% param1 refers to the temporary local data-element which has
% received the value of the actual parameter data-element on
% entry to the routine
    2*param1=:param1
% value in the temporary local data-element is transferred back
% to the actual parameter data-element after the RETURN
% statement is executed
    RETURN
    ENDROUTINE
%
% code to invoke the above routine
%
    3=:int
    twice(int)          % after invocation int = 6
% the following is equivalent to the previous invocation
    twice(3=:int)
% Note, following invocation is invalid, it has no explicit
% actual parameter data-element, for the value to be returned
    twice(3+2*5)
```

3. A parameter with WRITE only access mode.

```
    ROUTINE INTEGER,VOID (INTEGER WRITE) : triple (param1)
    3*@ =: param1
% value in the temporary local data-element is transferred back
% to the actual parameter data-element after the RETURN
% statement is executed
    RETURN
    ENDROUTINE
%
% code to invoke the above routine
%
    INTEGER : int
%
    2 triple (int)      % after invocation int = 6
% Note, the following invocation is invalid
    2 triple (3+5)      % no explicit actual parameter
                        % data-element
```

The composite data types are transferred by reference. This means that during execution of a routine, the address of each actual parameter data-element is transferred into the routine. Then each reference to a formal parameter will cause the actual parameter data-element to be referenced directly during execution of the routine.

In Fortran and Cobol parameters are always transferred by reference. Consequently, a routine written in PLANC must include the routine modifier STANDARD, in its declaration, to be callable from Fortran or Cobol.

7.5 EXIT FROM A ROUTINE

Exit from a routine will take place when execution reaches a RETURN, an ERRETURN or an ENDROUTINE statement. Any number of RETURN and ERRETURN statements may appear in a routine.

The general form of a RETURN statement is :

```
        [expression] RETURN
```

where

expression   must be present if the routine has an out-value declared. The resulting value of the expression must be of the data type declared fcr the out-value.

The general form of an ERRETURN statement is :

```
        expression ERRETURN
```

where

expression   is the resulting value of the expression which must be of the data type INTEGER.

A RETURN or an ENDROUTINE may be used for normal exit from a routine. However, if the routine has an out-value declared, then exit from the routine must be via a RETURN or an ERRETURN statement. The PLANC compiler will check that at least one RETURN is present, if the routine is declared with a non VOID out-value.

The RETURN statement will transmit the out-value of a routine back to the invoking routine.

Exit via an ERRETURN statement will transfer control back to the invoking routine. If the invoking routine has a ON ROUTINEERROR statement prior to the routine invocation statement, then control will be transferred to the beginning of that exception handling group of statements. Otherwise, control will be transferred to the next higher level in the routine invocation hierarchy, and so on, until a level is reached containing a routine exception handler, or the outer level is reached where the program execution will terminate, see section 6.8 for conditions to enter the exception handler, and a run-time error message will be issued.

Exit via an ERRETURN statement will make the resulting value of the expression available in the system variable, ERRCODE, which has a data-element of the integer data type.

If an exit via an ERRETURN statement has transferred control to a user routine exception handler, then following completion of the exception handler, control will be transferred in one of the following ways :

- an ENDON acts as if the last executed routine call had executed a RETURN. Note that an out-value data-element or actual parameter data-elements with WRITE access, would contain unpredictable values.
- a GO statement may transfer control to a label.
- a RETURN or ERRETURN will exit from the routine containing the exception handler to its caller.

7.6 ROUTINE TYPE SPECIFICATION AND USER DEFINED ROUTINE TYPE

A routine is a composite data type in PLANC. Thus, a routine is made up of components of other data types. Further, the facility of a user specifying his own composite data types in terms of those already available, also applies to the routine data type.

A user may define a new data type based on the routine data type. This TYPE specification will include :

1) Routine modifier options, eg. STANDARD, INLINE, if required.

2) The data types of the routine's in-value and out-value.

3) The data types of all of the formal parameters, which will be present in any routine data-element of this newly defined TYPE.

Thus, part of the routine header is specified for every routine data-element declared to be of this user defined TYPE. This mechanism may be used to create a family of routines with similar structure, ie. same in-value and out-value data types, same number of parameters and parameter data types.

For example :

```
    TYPE rtnfamily = ROUTINE REAL,VOID (INTEGER WRITE)
```

A possible application might be to create a stack for a particular record data type data-element, with functions such as push, pop, etc., each routine handling one record data-element and the stack :

```
    TYPE stackrec = RECORD
                     INTEGER : i1,i2
                     REAL    :r1,r2
                    ENDRECORD
%
    TYPE stackrtn = ROUTINE VOID,VOID (stackrec READ WRITE)
% declare various routines in the stack handling family
    stackrtn : push (inrec)
% put the record on the global stack
%    .
    ENDROUTINE
%
    stackrtn : pop (outrec)
% return a record from the global stack
%    .
    ENDROUTINE
```

7.7 RECURSIVE ROUTINES

Routines in PLANC may invoke themselves recursively with certain restrictions. For direct recursion, a routine may invoke itself only if it is declared in the outer-most level of a module. This also applies to modules nested within other modules. An alternative explanation is that any routine nested within another routine must not invoke itself recursively. Indirect recursive invocations are allowed at any level of nested routines or nested modules, provided that the chain of routine invocations goes via the routine at the outer level of the module containing the nested routine which is then invoked by indirect recursion.

For example :

```
    ROUTINE VOID,INTEGER (INTEGER) : factorial (number)
  % compute n! (n factorial) recursively
    IF number > 1 THEN
  % invoke factorial again recursively for next lower value
      number*factorial(number-1) RETURN
  % terminal condition of recursion
    ELSE
      1 RETURN
    ENDIF
    ENDROUTINE
  %
  % code to invoke the above recursive routine
  %
    INTEGER : int
  %
    factorial(5)=:int  % result is 5*4*3*2*1 = 120
```

Note that routines with the qualifiers, SPECIAL or INLINE, cannot invoke themselves recursively.

The following examples show which routines may legitimately invoke themselves recursively :

1. A routine declared in the outer level of a module.

```
        MODULE abc
    %
        ROUTINE ...   : rtnyes
    % this routine, rtnyes, may invoke itself recursively
           ROUTINE ...   : rtnno
    % this nested routine, rtnno, may not invoke itself recursively
           ENDROUTINE
        ENDROUTINE
        ENDMODULE
```

2. A routine declared within a nested module.

```
        MODULE outer
    %
          MODULE abc
    %
          ROUTINE ...   : rtnyes
    % this routine, rtnyes, may invoke itself recursively
             ROUTINE ...   : rtnno
    % this nested routine, rtnno, may not invoke itself recursively
             ENDROUTINE
          ENDROUTINE
          ENDMODULE       % end of abc
        ENDMODULE         % end of outer
```

3. Indirect recursion, routines declared in separate modules.

```
        MODULE his
    % necessary EXPORT/IMPORT statements
        ROUTINE ... popeye
    % this routine may invoke "oliveoil"
           ROUTINE ... roughhouse
    %        this routine may invoke "oliveoil"
           ENDROUTINE
        ENDROUTINE
        ENDMODULE         % end of his
    %
        MODULE hers
    % necessary EXPORT/IMPORT statements
        ROUTINE ... oliveoil
    % this routine may invoke "popeye" creating indirect recursion
        ENDROUTINE
        ENDMODULE         % end of hers
```

7.8 SCOPE OF IDENTIFIERS IN PLANC ROUTINES

An identifier may be created in a routine by a normal declaration or a type specification. Identifiers defined within a routine will have a scope including the entire routine. However such identifiers may not have an identifier name which is identical to an identifier whose scope includes this routine, ie. an identifier may not be declared twice within nested routines.

If routine declarations are nested, then identifiers created within the inner routines have the same restriction as above concerning the choice of identifier names. Note that while INLINE routines expansions are inserted at each invocation, this does not restrict the identifier names which may be used locally within the INLINE routine. The INLINE routine may use local identifier names which are the same as names with a scope which includes the invocation of the INLINE routine.

7.9 STANDARD ROUTINES AVAILABLE IN PLANC

The standard routines are listed in this section in alphabetical order.

ADDR

The ADDR standard routine takes as a parameter, an identifier of any data type, ie. simple, composite, predefined or user defined. It will return the address in memory of the corresponding data-element.

Note, if several routines in one module have the same routine name, then the ADDR standard routine will return the address of the first routine declared in the module. If the ADDR standard routine refers to a routine data-element, the routine identifier must not be enclosed in parentheses.

APPEND

The APPEND standard routine will add a record to the end of a linked list of records. For a detailed illustration of the use of APPEND, see section 4.6 .

BIT

The BIT standard routine will store or retrieve a boolean value into/from one bit position of the data-element associated with an identifier. For example :

```
INTEGER : int
BOOLEAN : bl1
TRUE=:BIT(int,3)
BIT(int,3)=:bl1
```

will store a value 1 into bit 3 (the fourth bit from the right) of the integer data-element associated with int. The third bit of int is retrieved and stored into bl1.

BLOCKSIZE

The BLOCKSIZE standard routine will set the blocksize of a file. For a detailed description see section 9.8.

CLOSE

The CLOSE standard routine will terminate the connection of an external file to an internal file number. For a detailed description see section 9.7.

CONVERT

The CONVERT standard routine will carry out conversion between various integer and real data-elements. For a detailed description see section 5.5 .

DISPOSE

The DISPOSE standard routine is used to deallocate dynamically created data-elements. For a detailed description see section 4.5 .

The form of the routine declaration of the DISPOSE standard routine follows :

```
          ROUTINE INTEGER POINTER,VOID                                  &
           : XDISPOSE ALIAS '5DISPOSE' % ALIAS '#DISPOSE' on ND-500
```

where

in-value   is the address of the data-element to be deallocated.

FILESIZE

The FILESIZE standard routine is used to set the size of a file, or to inquire as to the present size of a file. For a detailed description see section 9.9 .

FORCE

The FORCE standard routine will move a value from one data-element to another, regardless of the data types. For a detailed description see section 5.5 .

IND

The IND standard routine will get the value of a data-element. The parameter to the IND standard routine must be an appropriate pointer identifier, to reference the data-element. All data types may have their data-element value picked up in this way, ie. simple, composite, predefined and user defined data types.

INISTACK

The INISTACK standard routine will create a new stack area. For a detailed description see section 8.6 .

INSERT

The INSERT standard routine will add a record to the beginning of a linked list of records. For a detailed illustration of the use of INSERT, see section 4.6 . It may also add a member to a set data-element.

INPUT

The INPUT standard routine may be used for formatted input or for random unformatted INPUT. For detailed description of the various INPUT routines see chapter 9 .

MAXINDEX

The MAXINDEX standard routine will return the declared upper bound of an array. The routine invocation may be used as follows :

```
MAXINDEX(array-identifier,dimension-number)
```

where

array-identifier is the identifier of the array whose upper bound is required

dimension-number is the number (from 1) of the index set, from which the upper bound is required.

Note that the dimension number must be an integer literal, it cannot be an identifier or an expression.

MININDEX

The MININDEX standard routine will return the declared lower bound of an array. The routine invocation may be used as follows :

```
MININDEX(array-identifier,dimension-number)
```

where

array-identifier
is the identifier of the array whose lower bound is required

dimension-number
is the number (from 1) of the index set, from which the lower bound is required.

Note that the dimension number must be an integer literal, it cannot be an identifier or an expression.

NEW

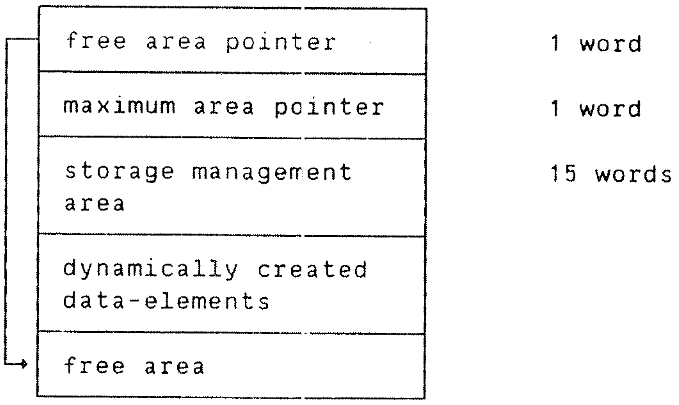
The NEW standard routine will dynamically create unnamed simple or composite data-elements. For a detailed description of the parameters and invocation of the NEW standard routine, see section 4.5 .

If the NEW standard routine dynamically creates a data-element within an explicitly declared array, eg.

```
INTEGER ARRAY : area(1:1000):=0    % see note below
REAL POINTER : rp
NEW REAL IN area =: rp
```

This will create an unnamed real data-element in the array area. The address of the real cata-element will be stored in the real pointer rp.

Some of the array elements of the array will be required for storage management of the memory used for dynamically created data-elements. The details of the storage management are :



For every invocation of the NEW standard routine which creates a data-element within an array, there will be two extra words required, in addition to the storage used for the created data-element.

If a DISPOSE is used to deallocate a data-element, then the area may be reused only if there is a request for a data-element of exactly the same size. Garbage collection, or reorganization of such an area is not carried out.

Note : prior to the first invocation of NEW, for a particular array, the area used for storage management, must be initialized to contain zeroes.

The form of the routine declaration of the NEW standard routine is :

```
        ROUTINE INTEGER,INTEGER POINTER ( INTEGER ARRAY )     &
           : XNEW (arr)  ALIAS '5NEW' % ALIAS '#NEW' on the ND-500
```

where

in-value   is the size of the data-element to be created in bytes.

out-value   address of the created data-element.

parameter   is the array in which the data-element is to be created.

OPEN

The OPEN standard routine will establish the connection of an external file to an internal file number. For a detailed description see section 9.6 .

OUTPUT

The OUTPUT standard routine may be used for formatted output or for random unformatted OUTPUT. For detailed description of the various OUTPUT routines see chapter 9 .

PRED

The PRED standard routine may be used on enumeration data-elements only. It will return the previous enumeration value, within the declared list of enumeration values, to that contained in the data-element which is the parameter for the routine invocation.

For example :

```
        ENUMERATION (good,better,best) : moral
        best=:moral
        PRED(moral)=:moral  % stores the value 'better'
        PRED(good) ...      % will return an unpredictable value
```

REMOVE

The REMOVE standard routine will remove a record from a linked list of records. For a detailed illustration of the use of REMOVE, see section 4.6 . It may also remove a member from a set data-element.

SIZE

The SIZE standard routine returns the number of bytes used for storage of a data-element. It may also be used to get the number of bytes required for any data-element of a specified data type. For a more detailed description see section 3.17 .

SUCC

The SUCC standard routine may be used on enumeration data-elements only. It will return the following enumeration value, within the declared list of enumeration values, to that contained in the data-element which is the parameter for the routine invocation.

For example :

```
ENUMERATION (good,better,best) : moral
better=:moral
SUCC(moral)=:moral  % stores the value 'best'
SUCC(best) ...      % will return an unpredictable value
```

TYPEOF

The TYPEOF standard routine specifies identifiers to be of the same data type as a previously declared identifier. For detailed description see section 3.14 .

7.10 Table of PLANC Standard Routines

Abbreviations used in the following table :

i-v	in-value
o-v	out-value
n/a	not applicable

Standard Function name	brief function description	allowed parameter data type(s)	parameter description
ADDR	get address of a data-element	i-v void o-v any address data-element 1. any data type	n/a an address data-element name of a data-element
APPEND	add a record to the end of a linked list	i-v record o-v void 1. address implied range	append record n/a list specifier
BIT	store a boolean (bit) value	i-v boolean o-v void 1. identifier of a simple type 2. integer liter. or constant	value for store n/a store into data-element bit number
BIT	extract a boolean (bit) value	i-v void o-v boolean 1. identifier of a simple type 2. integer liter. or constant	n/a value stored get value from data-element bit number
BLOCKSIZE	set blocksize of a file	i-v integer o-v void 1. integer	blocksize n/a file number
CLOSE	close a file	i-v void o-v void 1. integer	n/a n/a file number
CONVERT	convert to or from real and integer types	i-v real/integer o-v real/integer 1. REAL/INTEGER	from data-element to data-element target data type
DISPOSE	deallocate dynamically allocated data-element	i-v int. pointer o-v void	data-element address n/a

Standard Function name	brief function description	allowed parameter data type(s)	parameter description
FILSIZE	set/read filesize of a file	i-v integer4 o-v integer4 1. integer	set file size to read file size file-number
FORCE	interpret data-element value as a different data type	i-v data-element o-v data-element 1. any data type	from value to value
IND	get a data-element value via a pointer to it	i-v void o-v any data type 1. pointer data type	n/a value retrieved pointer to the data-element
INISTACK	create a new stack area	i-v void o-v void 1. integer array	n/a n/a area for stack
INSERT	add a record to the head of a linked list	i-v record o-v void 1. address implied range	insert record n/a list specifier
INPUT	formatted input	i-v void o-v integer 1. integer 2. bytes 3. any data-elem.	n/a chs. transferred device number format descriptor input data-elem.
INPUT	random unformatted input	i-v void o-v integer 1. integer 2. integer 3. bytes	n/a chs. transferred file number block number input area
MAXINDEX	get current upper bound of an array	i-v void o-v integer 1. array ident. 2. integer liter. or constant	n/a upper bound name of array index set no.
MININDEX	get current lower bound of an array	i-v void o-v integer 1. array ident. 2. integer liter. or constant	n/a lower bound name of array index set no.
NEW	dynamically create a new data-element	i-v void o-v pointer data-element 1. any data type	n/a adr. of new data-element data type of new data-element

Standard Function name	brief function description	allowed parameter data type(s)	parameter description
OPEN	open a SINTRAN file	i-v void o-v integer 1. integer 2. bytes 3. bytes 4. bytes	n/a chs. transferred file number file access code file name file type
OUTPUT	formatted output	i-v void o-v integer 1. integer 2. bytes 3. specified type	n/a chs. transferred device number format descriptor output data-elem.
OUTPUT	random unformatted output	i-v void o-v integer 1. integer 2. integer 3. bytes	n/a chs. transferred file number block number input area
PRED	get the immediately prior enumeration value	i-v void o-v enum. value 1. enum. ident.	n/a prior value
REMOVE	remove a record from a linked list	i-v record o-v void 1. address implied range	remove record n/a list specifier
SIZE	get storage, in bytes, used by a data type	i-v void o-v integer? 1. identifier or data type	n/a number of bytes
SUCC	get the immediately prior enumeration value	i-v void o-v enum. value 1. enum. ident.	n/a following value
TYPEOF	specify identifiers to be of the same data type	i-v void o-v any data type 1. list of identifiers	n/a type of elem.

8 PROGRAM_STRUCTURE

In order to construct a complete PLANC program which can be executed, the following things must be present :

1) At least one MODULE with its component parts.

2) One MODULE must contain at least one routine, of the special type PROGRAM, to define a main entry point to begin execution.

8.1 BASIC_MODULE

A MODULE is the smallest independent part of a PLANC program which can be compiled separately. Further, it is the minimum entity required to form a program which can be executed as an independent program, providing it contains a main PROGRAM routine, see section 8.2.

In large or complex systems it is usually desirable to group into separate entities, similar functions or data structures.This may serve the purpose of being able to more effectively administer the functions required in the system, or making a single copy of a widely used data structure available to any part of the system from one central place. In PLANC the MODULE is the mechanism to do this, by collecting appropriate or related routines into a suitably chosen number of MODULE's for a system.

The form of a basic MODULE comprises the following components :

1) The declared MODULE name.

2) EXPORT declarations for data-elements, declared in this MODULE, to be made available to other MODULE's, see section 8.3.

3) IMPORT declarations for data-elements from another MODULE, to be accessible within this MODULE, see section 8.3.

4) Declarations and TYPE specifications, local to a MODULE, which will be global to all levels of routines declared within this MODULE. These declarations include all routine declarations for this MODULE.

5) Executable statements,if any, required for this MODULE.

The general form of a MODULE declaration is :

```
        MODULE mod-ident
    %     EXPORT statements for data-elements required externally
    %
    %     body of the module
        ENDMODULE
```

where

mod-ident is an identifier for this module.

Note that any EXPORT statements required for this module, must precede all other declarations. However, TYPE specifications and IMPORT statements may precede an EXPORT statement in a module.

For example :

```
        MODULE mymodule
        EXPORT myint
    % only identifiers global in this module may be EXPORT'ed
        INTEGER : myint
        PROGRAM : mainprogram
    % declarations local to the main program
        INTEGER : locint
    % executable part of main program routine
        ENDROUTINE
        ENDMODULE
```

8.2 MAIN_PROGRAM

A special type of routine is the main PROGRAM. There must be one main PROGRAM routine in a program to be executed. The general form of a routine header of a main PROGRAM routine is :

```
          PROGRAM : routine-name
```

where

routine-name is a valid identifier which is the main entry point to be used to begin program execution.

The main PROGRAM routine has no in-value, out-value or parameters. All other things permitted for routines, eg. declarations, type definitions and inner nested routines, may be used in a main PROGRAM routine. A main program routine must be terminated by an ENDROUTINE statement in the same way as a normal routine.

For example :

```
          PROGRAM : myprogram
     %
     % an inner routine
     %
          ROUTINE VOID,INTEGER (INTEGER) : myroutine (intparam)
     % routine body
          intparam RETURN
          ENDROUTINE
     %
     % local declarations for main PROGRAM routine
     %
          INTEGER : int
     %
     % executable statements
     %
          10=:int
     %
     % end of main PROGRAM
     %
          ENDROUTINE
```

The above main PROGRAM would have to be compiled in a MODULE, then linked with a Loader. The name 'myprogram' will be the main entry point which can be used to begin execution of the program.

8.3 EXPORT/IMPORT - COMMUNICATION BETWEEN MODULES

Modules are used in large systems to group routines and data-elements in some way appropriate to the particular design for the project. It will often be necessary to access data-elements, declared in one module, from one or more other modules. PLANC requires explicit declarations for both the module containing the data-element and the modules wishing to gain access. For the purposes of inter-module communication, routines are treated as other data-elements.

An EXPORT statement, in a module, makes available particular data-elements for access by other modules.

The general form of an EXPORT statement is :

```
EXPORT [(SYSTEM)] identifier[,identifier]...
```

where

identifier is an identifier associated with a data-element declared within this module.

The optional qualifier, (SYSTEM), will make the routine identifier associated with a data-element inaccessible unless IMPORT'ed with the (SYSTEM) qualifier. If this option is used in an EXPORT statement, then it must be used in any matching IMPORT statements. This is of particular interest as an extra protection to avoid naming conflicts for system provided routines, in run-time systems. The ALIAS facility can be used in a similar way, see section 7.1 . Users are strongly advised to use the ALIAS facility if special routine names are required.

It is illegal to EXPORT a family of routines, with the routine name identifier the same as the name of a PLANC predefined standard routine or operator, see section 8.4 for the use of a family of routines.

EXPORT statements must be placed immediately following the MODULE statement.

For example :

```
         MODULE exhibit
         EXPORT bool,vector
         BOOLEAN : bool
         INTEGER ARRAY : vector(1:100)
  7        .
         ENDMODULE
```

An IMPORT statement, specifies data-elements to be used in a module, providing they have been made available in another module by an EXPORT statement.

The general form of an IMPORT statement is :

```
          IMPORT [(option)] declaration[,declaration]...
```

where

declaration is the same as the declaration of the data-element in the module containing the original declaration.

option is either SYSTEM or COMMON.

If the option (SYSTEM) is present in the matching EXPORT statement, then it must also be present in the IMPORT statement.

If the option (COMMON) is used, the identifier(s) may only be used to link to a named COMMON block defined in a Fortran program, see section D.7, Appendix D for more details.

If declarations of different data types are to be included in one IMPORT statement, then each declaration must be included in parentheses.

For example :

```
          IMPORT (INTEGER : i1,i2),(REAL : r1,r2),(BOOLEAN : b1)
```

As an IMPORT statement contains the data type of each IMPORT'ed data-element, all of the normal compilation checks will be carried out on the identifier. These checks apply within the module containing the IMPORT statement. The PLANC compiler checks the correct correspondence with the data-element's data type, declared in the originating module and in the IMPORT statement, if both modules are nested within another module. If the two modules with the corresponding EXPORT/IMPORT statements are not nested within another module, ie. they are separately compiled, then these correspondence checks are not done.

If the data-element IMPORT'ed is a routine, then its declaration in the IMPORT statement must be in parentheses. Further, the list of formal parameter identifiers declared in the routine, must not be included in the IMPORT statement.

For example :

```
          IMPORT ( ROUTINE VOID,VOID (INTEGER) : doit )
```

For families of routines, declared in another single module, the use of ALIAS names is necessary. This allows one or more variants, of routines declared with the same identifier, to be accessed by the IMPORT statement, see section 8.4 .

A user defined data type, specified in a TYPE statement, or identifiers declared in a CONSTANT statement, may be IMPORT'ed into an inner nested module, see section 8.5 .

Examples of the use of EXPORT/IMPORT statements :

1. Some simple data-elements.

```
    MODULE source
    EXPORT int,rl,bool
%
    INTEGER : int
    REAL: rl
    BOOLEAN : bool
%
    ROUTINE VOID,VOID : looknice
%     .
    ENDROUTINE
    ENDMODULE
%
% a separate module which could be compiled separately
%
    MODULE getem
    IMPORT ( INTEGER : int ), ( REAL : rl )
    IMPORT BOOLEAN : bool
%
% now 'int', 'rl' and 'bool' are available in this module
%
    ENDMODULE
```

2. A routine to be accessed from another module.

```
    MODULE service
    EXPORT useful
%
    ROUTINE VOID,INTEGER (INTEGER) : useful (param)
%
% body of the routine
%
    ENDROUTINE
    ENDMODULE
%
% a separate module which could be compiled separately
%
    MODULE getit
    IMPORT ( ROUTINE VOID,INTEGER (INTEGER) : useful )
%
% now 'useful' is available in this module
%
    ENDMODULE
```

For more complex use of routines and EXPORT/IMPORT statements, see section 8.4 .

8.4 ALIAS USE IN A MODULE

A family of routines to create an operator for various data types, may be declared in one module. All the routines will have the same routine name identifier. If the routines are to be invoked by other routines within the same module, then nothing further is required. The PLANC compiler will compile each invocation with a reference to the correct routine, which requires an exact match of the data types of the in-value and the parameters. If there is not an exact match, the compiler will give an error message unless there are corresponding parameters with some data type modifications. For range or precision modification, accurracy may be lost.

For example :

```
    MODULE allinone
    INTEGER ARRAY : stack(0:1000)
%
% define a family of routines for a +++, plus 1 operator
%
% each routine will 'add' 1 for a particular data type
%  and return the result as an out-value
%
    ROUTINE INTEGER,INTEGER : +++
    @+1 RETURN          % return in-value+1
    ENDROUTINE
%
    ROUTINE REAL,REAL : +++
    @+1.0 RETURN        % return in-value+1.0
    ENDROUTINE
%
    ROUTINE BOOLEAN,BOOLEAN : +++
    NOT @ RETURN        % return complement of in-value
    ENDROUTINE
%
% program to invoke the above +++ routines
%
    PROGRAM : myplus
    INTEGER : int    ; REAL : rl    ; BOOLEAN : bool
% executable program
    INISTACK stack
% invoke the integer version of +++
    5 +++ =:int         % result is 6
% invoke the real version of +++
    3.51 +++ =: rl      % result is 4.51
% invoke the boolean version of +++
    TRUE +++ =: bool    % result is FALSE
%
    ENDROUTINE
    ENDMODULE
```

The routine name identifier of a family of routines should not be the same as the name of a PLANC predefined standard routine or operator as it is illegal to EXPORT a family of routines with such a name.

If such a family of routines were created in one module, but the routines were to be invoked from another module, then ALIAS names would be required for each routine in the family. Further, the family would have to be EXPORT'ed from its module and IMPORT'ed into the module containing the routine invocations.

For example :

```
         MODULE family
     %
     % define a family of routines for a +++, plus 1 operator
     %
     % each routine will 'add' 1 for a particular data type
     %  and return the result as an out-value
     %
     % set-up access to the family of routines
         EXPORT +++
     %
         ROUTINE INTEGER,INTEGER : +++ ALIAS 'intplus'
         @+1 RETURN          % return in-value+1
         ENDROUTINE
         ROUTINE REAL,REAL : +++ ALIAS 'realplus'
         @+1.0 RETURN        % return in-value+1.0
         ENDROUTINE
         ROUTINE BOOLEAN,BOOLEAN : +++ ALIAS 'boolplus'
         NOT @ RETURN        % return complement of in-value
         ENDROUTINE
         ENDMODULE           % end of module family
     %
         MODULE usethem
     % set-up access to the module with the +++ routines
         IMPORT( ROUTINE INTEGER,INTEGER : +++ ALIAS 'intplus' )
         IMPORT( ROUTINE REAL,REAL : +++ ALIAS 'realplus' )
         IMPORT( ROUTINE BOOLEAN,BOOLEAN : +++ ALIAS 'boolplus' )
         INTEGER ARRAY : stack(0:1000)
     %
     % program to invoke the above +++ routines from another module
     %
         PROGRAM : myplus
         INTEGER : int    ; REAL : rl    ; BOOLEAN : bool
     % executable program
         INISTACK stack
     % invoke the integer version of +++
         5 +++ =:int         % result is 6
     % invoke the real version of +++
         3.51 +++ =: rl      % result is 4.51
     % invoke the boolean version of +++
         TRUE +++ =: bool    % result is FALSE
     %
         ENDROUTINE
         ENDMODULE           % end of module usethem
```

Note, that these two modules could be compiled together in one file, or separately, prior to execution. In fact if these modules were nested within another module, then the ALIAS names would not be necessary.

The previous example could be coded differently, with the module which is to invoke the routines referring to the unique ALIAS names only. This applies to the IMPORT statements and the routine invocations.

For example :

```
    MODULE family
%
% define a family of routines for a +++, plus 1 operator
%
% each routine will 'add' 1 for a particular data type
%  and return the result as an out-value
%
% set-up access to the family of routines
    EXPORT +++
%
    ROUTINE INTEGER,INTEGER : +++ ALIAS 'intplus'
    @+1 RETURN          % return in-value+1
    ENDROUTINE
%
    ROUTINE REAL,REAL : +++ ALIAS 'realplus'
    @+1.0 RETURN        % return in-value+1.0
    ENDROUTINE
%
    ROUTINE BOOLEAN,BOOLEAN : +++ ALIAS 'boolplus'
    NOT @ RETURN        % return complement of in-value
    ENDROUTINE
    ENDMODULE           % end of module family
%
    MODULE usethem
% set-up access to the module with the +++ routines
% note, that now reference is directly to the ALIAS names
    IMPORT( ROUTINE INTEGER,INTEGER : intplus )
    IMPORT( ROUTINE REAL,REAL : realplus )
    IMPORT( ROUTINE BOOLEAN,BOOLEAN : boolplus )
    INTEGER ARRAY   stack(0:1000)
%
% program to invoke the above +++ routines from another module
%
    PROGRAM : myplus
    INTEGER : int    ; REAL : rl    ; BOOLEAN : bool
% executable program
    INISTACK stack
% invoke the integer version of +++
    5 intplus =:int      % result is 6
% invoke the real version of +++
    3.51 realplus =: rl % result is 4.51
% invoke the boolean version of +++
    TRUE boolplus =: bool % result is FALSE
    ENDROUTINE
    ENDMODULE           % end of module usethem
```

Note, these two modules could be compiled together in one file, or separately, prior to execution. These modules cannot be both nested within one module as the loader must complete the links for ALIAS names.

The family of routines can be given a new family name within the program which will invoke the appropriate routine in the family.

For example :

```
    MODULE family
%
% define a family of routines for a +++, plus 1 operator
%
% each routine will 'add' 1 for a particular data type
%  and return the result as an out-value
%
% set-up access to the family of routines
    EXPORT +++
%
    ROUTINE INTEGER,INTEGER : +++ ALIAS 'intplus'
    @+1 RETURN          % return in-value+1
    ENDROUTINE
%
    ROUTINE REAL,REAL : +++ ALIAS 'realplus'
    @+1.0 RETURN        % return in-value+1.0
    ENDROUTINE
%
    ROUTINE BOOLEAN,BOOLEAN : +++ ALIAS 'boolplus'
    NOT @ RETURN        % return complement of in-value
    ENDROUTINE
    ENDMODULE           % end of module family
%
    MODULE usethem
% set-up access to the module with the +++ routines
% note, that now reference is through a new family name
    IMPORT( ROUTINE INTEGER,INTEGER : plus1 ALIAS 'intplus' )
    IMPORT( ROUTINE REAL,REAL : plus1 ALIAS 'realplus' )
    IMPORT( ROUTINE BOOLEAN,BOOLEAN : plus1 ALIAS 'boolplus' )
    INTEGER ARRAY : stack(0:1000)
%
% program to invoke the above +++ routines from another module
%
    PROGRAM : myplus
    INTEGER : int    ; REAL : rl    ; BOOLEAN : bool
% executable program
    INISTACK stack
% invoke the integer version of +++
    5 plus1 =:int       % result is 6
% invoke the real version of +++
    3.51 plus1 =: rl    % result is 4.51
% invoke the boolean version of +++
    TRUE plus1 =: bool  % result is FALSE
%
    ENDROUTINE
    ENDMODULE           % end of module usethem
```

Note, these two modules could be compiled together in one file, or separately, prior to execution. These modules cannot be both nested within one module as the loader must complete the links for ALIAS names.

The family of routines can be given new individual names within the program which invokes each of the routines in the family.

For example :

```
    MODULE family
%
% define a family of routines for a +++, plus 1 operator
%
% each routine will 'add' 1 for a particular data type
%  and return the result as an out-value
%
% set-up access to the family of routines
    EXPORT +++
%
    ROUTINE INTEGER,INTEGER : +++ ALIAS 'intplus'
    @+1 RETURN          % return in-value+1
    ENDROUTINE
%
    ROUTINE REAL,REAL : +++ ALIAS 'realplus'
    @+1.0 RETURN        % return in-value+1.0
    ENDROUTINE
%
    ROUTINE BOOLEAN,BOOLEAN : +++ ALIAS 'boolplus'
    NOT @ RETURN        % return complement of in-value
    ENDROUTINE
    ENDMODULE           % end of module family
%
    MODULE usethem
% set-up access to the module with the +++ routines
% note, that now we create local names for each routine
    IMPORT( ROUTINE INTEGER,INTEGER : int1 ALIAS 'intplus' )
    IMPORT( ROUTINE REAL,REAL : real1 ALIAS 'realplus' )
    IMPORT( ROUTINE BOOLEAN,BOOLEAN : bool1 ALIAS 'boolplus' )
    INTEGER ARRAY : stack(0:1000)
%
% program to invoke the above +++ routines from another module
%
    PROGRAM : myplus
    INTEGER : int    ; REAL : rl    ; BOOLEAN : bool
% executable program
    INISTACK stack
% invoke the integer version of +++
    5 int1 =:int        % result is 6
% invoke the real version of +++
    3.51 real1 =: rl    % result is 4.51
% invoke the boolean version of +++
    TRUE bool1 =: bool  % result is FALSE
%
    ENDROUTINE
    ENDMODULE           % end of module usethem
```

Note, that these two modules could be compiled together in one file, or separately, prior to execution. These modules cannot be both nested within another module as the loader must complete the links for ALIAS names.

8.5 MODULE STRUCTURE AND SEPARATE COMPILATION

Modules are independent entities which may be compiled separately by the PLANC compiler. Then a Loader must be used to link all the necessary separate modules together. All required links between the separately compiled modules will be resolved, by the Loader as external references. This can only be done successfully if the links between the modules have been correctly defined with EXPORT/IMPORT statements, see 8.3 .

If several routines in a module have the same name, then the Loader would not be able to resolve such an ambiguity, unless ALIAS names have been used to give a unique qualifier name to each routine, see section 8.4 .

TYPE specification and CONSTANT statements may precede all modules on a file. In this case these staments will not be contained within any module. During the compilation, identifiers thus created will be globally available to all modules in the compilation. In fact, user specified data types will appear identical to the data types defined within the PLANC compiler. Further, TYPE specifications to be used in this way may be inserted by an INCLUDE compiler command, see Appendix A .

Modules may be nested within other outer modules to any practical number of levels. If modules are nested, the inner modules can access data-elements declared in outer module levels, only by the usual means of EXPORT/IMPORT statements. This would be exactly the same as if the inner module was removed and compiled as a separate module.

However, nesting of modules does offer extra facilities.

1) If a new data type is specified in an outer level module, then the type specification may be IMPORT'ed to an inner level nested module. If the new data type is to be IMPORT'ed over several levels of nested modules, then it must be IMPORT'ed at every level between the original TYPE specification and the inner level module wishing to access it.

2) Identifiers declared in CONSTANT statements may be accessed in nested modules in exactly the same way as TYPE specifications, without EXPORT statements, but with IMPORT statements at every level between the original TYPE specification and the inner level module wishing to access it.

3) If modules are nested within other modules, then checking of the correspondence of the declared data types in matching EXPORT and IMPORT statements is carried out at compilation time.

For example

```
        MODULE outer
        TYPE goods = INTEGER RANGE (1:128)
    %     .
        MODULE inner1
        IMPORT goods
    %     .
        MODULE inner2
        IMPORT goods
    %     .
        ENDMODULE           % end of inner2
        ENDMODULE           % end of inner1
        ENDMODULE           % end of outer
```

If modules are nested, routines and executable code may only be within the innermost module. However if there are two separate nests of modules within an outer module, then each separate nest of modules may have executable routine within its innermost module.

8.6 DATA-ELEMENT STORAGE AND THE PROGRAM STACK

Allocation strategy of data-elements and d∂tailed memory requirements are described for each PLANC implementation, see Appendix C. However, some aspects of data-element stor∂ge allocation apply to all PLANC compiler implementations.

In PLANC the distinction has been ma∂e between statically and dynamically allocated data-elements.

Statically allocated data-elements include :

1) Global data-elements declared in a basic MODULE.

2) Local data-elements, declared in a routine, whose access is READ only.

3) Data-elements, constructed by the NEW standard routine, within a global data-element, see section 4.5 .

Dynamically allocated data-elements include :

1) Local data-elements, declared in a routine, whose access is not READ only.

2) Data-elements, constructed by the NEW standard routine, within a local data-element or on the program stack, see section 4.5 .

A static data-element may be initialized with a specific value, in its declaration, provided that it is not within a nested routine. Static data-elements may be initialized within a nested routine if it is declared as READ only. Dynamically created data-elements are allocated on a stack, either when a routine is invoked, or when the NEW standard routine is invoked to create a data-element.

The stack used, is referred to as the 'current' stack. The INISTACK standard routine must be used to create a current stack at the beginning of program execution. It may be used during program execution to create further stacks.

The general form of the INISTACK standard routine invocation is :

```
INISTACK int-array
```

where

int-array is an INTEGER ARRAY, of one dimension, with an index set lower bound of zero.

The array, used in an INISTACK invocation, will remain the current stack until another INISTACK invocation, or until the routine with the INISTACK invocation terminates. When a routine terminates and returns to its invoker, all stack space allocated during execution of the routine will be released. The stack pointer will automatically be reset to the value it had prior the routine invocation.

Example of INISTACK use :

```
         MODULE mymodule
    %
         INTEGER ARRAY : stackarray (0:1000)
         PROGRAM main
    %
    % mandatory at the start of the executable program statements
    %
         INISTACK stackarray
    %
         ENDROUTINE
         ENDMODULE
```

8.7 SCOPE_OF_IDENTIFIER_NAMES_IN_PLANC_MODULES

In a module, identifiers may be created by declaration statements, TYPE specification statements or IMPORT statements. All identifiers created within the outer level of the module are available throughout the module, ie. the identifiers have a scope of the outer module only. However, if another module is nested, then the identifiers created in the outer module are available within the nested module in the following ways :

1) Identifiers created in the outer module by the usual declaration statements, eg. INTEGER or ENUMERATION, must have a corresponding IMPORT/EXPORT pair of statements, to make the identifier available within the nested module.

2) Identifiers created in TYPE specification or CONSTANT statements in the outer module, must be IMPORT'ed into the nested MODULE, but no EXPORT statement is to be used in the outer module, see section 8.5 . Only the identifier name is used in IMPORT statements used for this purpose.

3) Identifiers created in the outer module by the use of an IMPORT statement, must have another identical IMPORT statement to make the identifier available in a nested module, ie. an IMPORT statement must appear on every level between the outermost module and the nested module in which it is to be used.

TYPE specification statements and CONSTANT declarations may be made outside, or previous to any module in a compilation. These statements are then treated like compiler commands. Identifiers created in this way are globally available in all modules, separate or nested, without IMPORT statements.

9 INPUT/OUTPUT

The PLANC compiler and run-time system does not have very extensive input/output facilities. A set of standard routines has been provided for input/output, for various of the PLANC data types, to files and devices. One general limitation is that only one data-element may be input/output by a single input/output standard routine invocation. This has been done as it is envisaged that large systems programming projects will design and implement their own set of input/output routines, appropriate to their special needs.

The ROUTINEERROR exception will be activated by errors in any of the input/output or open/close standard routines. If a ROUTINEERROR condition occurs, the system variable, ERRCODE, will contain a value from the file system, specifying the nature of the error.

9.1 Input/Output Terms and Concepts

Input routines control the transfer of data from external media into internal storage. Output routines control the transfer of data from internal storage to external media.

In addition to the data transfer routines, other routines carry out file control operations. The following standard routines are provided in PLANC :

1) INPUT - data transfer.

2) OUTPUT - data transfer.

3) OPEN - file control.

4) CLOSE - file control.

5) BLOCKSIZE - file control.

6) FILESIZE - file control.

Records

A record is a sequence of values or characters which is considered as a single unit by the device it is being read to or written from. It may correspond to a physical entity such as a disc block or a magnetic tape block, but not necessarily.

There are two types of records :

1) Formatted

2) Unformatted

A formatted record is one which is transferred under the control of a format descriptor. Other records are unformatted records. During unformatted transfers, data is transferred on a one-to-one correspondence between external media and internal storage with no conversion or formatting operations.

Files

A file is a sequence of records, existing on an external device, accessible by a PLANC program via the SINTRAN file system.

File Number

A file number is a value in an INTEGER data-element, which specifies a particular file internally within a program. A file number is returned following the execution of the OPEN standard routine.

Format Descriptor

A format descriptor is a parameter in both Input and Output standard routine declarations. It describes the physical characteristics of a value after it has been transferred from a data-element by an output routine, or the physical characteristics before the value is to be transferred into a data-element by an input routine.

9.2 FORMATTED INPUT ROUTINES

The formatted INPUT standard routines transfer one value into a data-element. The general form of an invocation of a formatted INPUT standard routine is :

```
INPUT (file-number,'descriptor',identifier)
```

where

file-number is the file number obtained by the OPEN invocation.

descriptor is the format descriptor.

identifier is associated with the data-element into which the value is to be transferred.

Each of the formatted INPUT standard routines is declared with an out-value. This out-value will return the number of characters which have been transferred.

A field being read by an INPUT standard routine will terminate when either the maximum number of characters specified in the format descriptor has been read, or when a comma character (,), or a carriage return character is encountered.

If a field to be read by a formatted INPUT standard routine contains leading blanks and a numeric value, then the blanks will be recognised as part of the field width but will have no effect on the value transferred into a data-element.

The data types of the parameters of the formatted INPUT standard routines are shown in the general form of the INPUT standard routine declaration :

```
ROUTINE VOID,INTEGER (INTEGER,BYTES,id-type) : INPUT (...)
```

where

id-type is the data type of the data-element to receive the value read. This data type must correspond with that implied by the format descriptor.

In the following sections on the formatted INPUT standard routines the abbreviations used are :

w is an unsigned integer number greater than zero.

d is an unsigned integer number greater than or equal to zero.

Format Descriptors

The following are the format descriptors available for the formatted INPUT standard routines :

Iw Ow	-	Integer field descriptor
Fw.d Ew.d	-	Floating-point numeric field descriptors
Aw	-	Alphanumeric data field descriptor
Lw	-	Boolean data field descriptor

Note that if w or w.d is omitted, a maximum number of characters (default for each data type) will be used.

9.2.1 I FORMAT, INTEGER INPUT STANDARD ROUTINE

The Iw descriptor is for an integer value to be transferred into an INTEGER data-element from a field of up to w character positions.

The input field consists of an optional minus sign followed by a string of digits, ie. the same as an integer literal.

The field width described by an integer format descriptor can be overridden by the use of any non-numeric character as a delimiter between successive integer values to be read.

Examples :

value input	descriptor	internal value
1	I1	1
1	I5	1
10	I5	10
-15	I5	-15
1234	I2	12

The parameter data types of the integer INPUT standard routine are shown in the routine declaration :

```
        ROUTINE VOID,INTEGER                               &
            ( INTEGER, BYTES, INTEGER4 READ WRITE )
```

9.2.2 O FORMAT, OCTAL INPUT STANDARD ROUTINE

The Ow descriptor is for an octal value to be transferred into an INTEGER data-element from a field of up to w character positions.

The input field consists of a string of digits with no sign.

Examples :

value input	descriptor	internal value,dec
1	O1	1
10	O5	8
1234	O2	10

The parameter data types of the octal INPUT standard routine are shown in the routine declaration :

```
        ROUTINE VOID,INTEGER                               &
            ( INTEGER, BYTES, INTEGER4 READ WRITE)
```

9.2.3 F FORMAT, FIXED DECIMAL POINT INPUT STANDARD ROUTINE

The Fw.d descriptor is for a fixed decimal point value to be transferred into a REAL data-element from a field of w character positions.

The input field consists of an optional minus sign, followed by a string of digits optionally containing a decimal point. If there is no decimal point, the rightmost d digits are interpreted as the fractional part of the value. The rules are the same as for a REAL literal, see section 2.7.2 . If the input field has enough space, the value may be written in exponent form, see section 9.2.4.

Examples :

value input	descriptor	internal value
1.2	F5.0	1.0
-1.2	F5.0	-1.0
1.2	F5.1	1.2
-1.2	F5.1	-1.2
33	F10.3	33.0
3.2543	F10.3	3.254

The parameter data types of the fixed decimal point INPUT standard routine are shown in the routine declaration :

```
ROUTINE VOID,INTEGER ( INTEGER, BYTES, REAL READ WRITE )
```

9.2.4 E_FORMAT, FIXED DECIMAL POINT NORMALIZED WITH EXPONENT INPUT STANDARD ROUTINE

The Ew.d descriptor is for a fixed decimal point value, normalized with an exponent, to be transferred into a REAL data-element from a field of up to w character positions.

The input field may have the same form as described above for the F descriptor. This field may optionally be followed by an exponent of the form Enn or E-nn, where nn is limited by the default REAL data type characteristics of the particular machine implementation, see Appendix C. The value from the input field will be multiplied by 10 to the power nn, to get the internally held value.

Examples :

value input	descriptor	internal value
1.2	E5.0	1.0
-1.2	E5.0	-1.0
1.2E2	E5.1	120.0
1.2E-2	E5.1	0.012
33	E10.3	33.0
3.2543E4	E10.3	32540.0
987654E-3	E10.3	987.654

The parameter data types of the fixed decimal point normalized with exponent INPUT standard routine are shown in the routine declaration :

```
ROUTINE VOID,INTEGER ( INTEGER, BYTES, REAL READ WRITE )
```

9.2.5 A FORMAT, ALPHANUMERIC INPUT STANDARD ROUTINE

The Aw descriptor is for an alphanumeric string to be transferred into a BYTES data-element from a field of up to w character positions.

If more than w characters are input, then the first w characters only will be stored in the data-element.

Examples :

value input	descriptor	internal value
1	A1	1
____1	A5	1
1____	A5	1____
abcde	A3	abc

The data types of the alphanumeric INPUT standard routine are shown in the routine declaration :

```
ROUTINE VOID,INTEGER ( INTEGER, BYTES, BYTES READ WRITE )
```

9.2.6 L FORMAT, BOOLEAN INPUT STANDARD ROUTINE

The Lw descriptor is for a boolean value to be transferred into a BOOLEAN data-element from a field of up to w character positions.

The input field is scanned for the first occurrence of one of the letters T or F, and the BOOLEAN data-element will be set to TRUE or FALSE accordingly. If no T or F is found in the input field, then the BOOLEAN data-element will be set to a value FALSE.

Examples :

value input	descriptor	internal value
T	L1	TRUE
T__	L5	TRUE
F	L3	FALSE
xyz	L3	FALSE

The data types of the boolean INPUT standard routine are shown in the routine declaration :

```
ROUTINE VOID,INTEGER                               &
  ( INTEGER, BYTES, BOOLEAN READ WRITE )
```

Note that the out-value will contain the character position, relative to 1, that the T or F has been found in.

9.3 RANDOM_UNFORMATTED_INPUT_STANDARD_ROUTINE

The random unformatted INPUT standard routine reads a record of data from a file, into a BYTES array data-element. The record may be selected randomly from any location within a file. The general form of an invocation of a random unformatted INPUT standard routine is :

```
INPUT (file-number,rec-number,array-ident)
```

where

file-number is the file number obtained by invocation of the OPEN standard routine.

rec-number is the record number within the file.
Note : the first record is number 0.

array-ident is an identifier associated with the BYTES data-element into which the value is to be transferred.

The parameter data types of the random unformatted INPUT standard routine are shown in the following routine declaration :

```
ROUTINE VOID,INTEGER ( INTEGER, INTEGER, BYTES )
```

The out-value of the random unformatted INPUT standard routine will be the number of characters actually transferred by the routine and this may be used within an expression.

9.4 FORMATTED OUTPUT STANDARD ROUTINES

The formatted OUTPUT standard routines transfer one value from a data-element to a file or a device. The general form of an invocation of a formatted OUTPUT standard routine is :

```
        OUTPUT (file-number,'descriptor',identifier)
```

where

file-number is the file number obtained by invocation of the OPEN standard routine.

descriptor is the format descriptor.

identifier is associated with the data-element into which the value is to be transferred.

Each of the formatted OUTPUT standard routines is declared with an out-value. This out-value will return the number of characters which has been transferred to the file or device. The field width part of a descriptor may be omitted and the out-value will have to be used to find out how many characters have been transferred.

If the value transferred does not fill the width specified for the field, then usually leading blanks will be inserted by the formatted OUTPUT standard routines.

If the internal value is too large to fit into the field width specified, then the output field will be filled with asterisk (*) characters.

The data types of the parameters of the formatted OUTPUT standard routines are shown in the general form of the OUTPUT routine declaration :

```
        ROUTINE VOID,INTEGER (INTEGER,BYTES,id-type) : OUTPUT (...)
```

where

id-type is the data type of the data-element whose value is to be output. This data type must correspond with that implied by the format descriptor.

In the following sections on the formatted OUTPUT standard routines the abbreviations used are :

w is an unsigned integer number greater than zero.

d is an unsigned integer number greater than or equal to zero.

Format Descriptors

The following are the format descriptors available for the formatted OUTPUT standard routines :

Iw	-	Integer field descriptor
Ow	-	Octal field descriptor
Zw	-	Octal field descriptor, with leading zeroes
Fw.d	-	Floating-point numeric field descriptors
Ew.d		
Dw.d		
Aw	-	Alphanumeric data field descriptor
Lw	-	Boolean data field descriptor

Note that if w or w.d is omitted, the minimum number of characters required to output the data-element will be used.

9.4.1 I FORMAT, INTEGER OUTPUT STANDARD ROUTINE

The Iw descriptor is for a value to be transferred from an INTEGER data-element to a field of w character positions, as a decimal value.

The value will be right-justified in the field. If the value is negative, one of the character positions will be used for a minus sign.

Examples :

internal value	descriptor	output
1	I1	1
1	I5	1
+10	I5	10
-15	I5	-15
1234	I4	1234
-1234	I4	****

The parameter data types of the integer OUTFUT standard routine are shown in the routine declaration :

```
ROUTINE VOID,INTEGER ( INTEGER, BYTES, INTEGER4 )
```

9.4.2 O AND Z FORMAT, OCTAL OUTPUT STANDARD ROUTINE

The Ow descriptor is for a value to be transferred from an INTEGER data-element as an octal value, to a field of up to w character positions.

The value will be right-justified in the field. If the value is negative, one of the character positions will be used for a minus sign.

Fields output with an Ow descriptor will contain leading space characters. The Zw descriptor will give leading zero characters.

Examples :

internal value,dec	descriptor		output
1	O1		1
1	O5		1
10	O5		12
10	Z5		00012
-5	O6		177773
4095	O5		7777
-4095	O5	(170001)	*****

The parameter data types of the octal OUTPUT standard routine are shown in the routine declaration :

```
ROUTINE VOID,INTEGER ( INTEGER, BYTES, INTEGER4 )
```

9.4.3 F_FORMAT, FIXED_DECIMAL_POINT_OUTPUT_STANDARD_ROUTINE

The Fw.d descriptor is for a value to be transferred from a REAL data-element, as a fixed point value, into a field of w character positions.

The w character positions will include a decimal point, and an optional minus sign. If the value does not fill the entire field, then the leading character positions will be blank filled.

The value output will be rounded to the number of decimal places specified, if necessary.

Examples :

internal value	descriptor		output
1.2	F5.0		1.
-1.2	F5.0		-1.
1.2	F5.1		1.2
-1.2	F5.1		-1.2
-10.33	F10.3		-10.330
12.3496	F5.2	(rounded)	12.35
1055.22	F5.2		*****

The parameter data types of the fixed decimal point OUTPUT standard routine are shown in the routine declaration :

```
ROUTINE VOID,INTEGER ( INTEGER, BYTES, REAL )
```

9.4.4 E FORMAT, FIXED DECIMAL POINT NORMALIZED WITH EXPONENT OUTPUT STANDARD ROUTINE

The Ew.d descriptor is for a value to be transferred from a REAL data-element to fixed decimal point normalized with exponent, into a field of w character positions.

The value output will be scaled to have one digit before the decimal point. There will be d digits after the decimal point. The exponent will comprise the letter E, a sign and two digits which are the power of ten to multiply the preceding value by.

Examples :

internal value	descriptor	output
1.2	E8.0	1.E+00
-1.2	E8.0	-1.E+00
120.0	E8.1	1.2E+02
0.012	E8.1	1.2E-02
.033	E13.3	3.300E-02
3.2543E4	E10.3	3.254E+04

The parameter data types of the fixed decimal point normalized with exponent OUTPUT standard routine are shown in the routine declaration :

```
ROUTINE VOID,INTEGER ( INTEGER, BYTES, REAL )
```

9.4.5 D FORMAT, FIXED DECIMAL POINT NORMALIZED WITH EXPONENT OUTPUT STANDARD ROUTINE

The Dw.d descriptor is for a value to be transferred from a double precision REAL data-element to fixed decimal point normalized with exponent, into a field of w character positions.

The value output will be in exactly the same format as that described above for the E descriptor.

Examples :

internal value	descriptor	output
.033	D13.3	3.300E-02
3.2543E4	D10.3	3.254E+04

The parameter data types of the fixed decimal point normalized with exponent OUTPUT standard routine are shown in the routine declaration :

```
ROUTINE VOID,INTEGER ( INTEGER, BYTES, REAL )
```

9.4.6 A/AL FORMAT. ALPHANUMERIC OUTPUT STANDARD ROUTINE

The Aw/ALw descriptor is for an alphanumeric string to be transferred from a BYTES data-element into a field of w character positions.

The character string will be output as ASCII characters. If the field width w is greater than the length of the string, then the string will be right-justified in the field and trailing character positions blank filled. If the AL descriptor is used then the character string will be left-justified and leading character positions blank filled.

Note that a single dollar character ($) in the string to be output will be converted, during output, to carriage return+line feed characters. To print a single dollar character ($), two consecutive dollar characters must be present in the string.

Examples :

internal value	descriptor	output
abcde	A5	abcde
abc	A5	__abc
abc	AL5	abc__

The data types of the alphanumeric OUTPUT standard routine are shown in the routine declaration :

```
ROUTINE VOID,INTEGER ( INTEGER, BYTES, BYTES )
```

9.4.7 L FORMAT. BOOLEAN OUTPUT STANDARD ROUTINE

The Lw descriptor is for a value to be transferred from a BOOLEAN data-element into a field of w character positions.

The right-most character position of the output field will have the letter T if the BOOLEAN data-element has a value TRUE, and the letter F if the BOOLEAN data-element has the value FALSE. The leading character positions of the output field will be blank filled.

Examples :

internal value	descriptor	output
TRUE	L1	T
FALSE	L5	____F

The parameter data types of the boolean OUTPUT standard routine are shown in the routine declaration :

```
ROUTINE VOID,INTEGER ( INTEGER, BYTES, BOOLEAN )
```

9.5 RANDOM UNFORMATTED OUTPUT STANDARD ROUTINE

The random unformatted OUTPUT standard routine writes a record of data from a BYTES array data-element to a file. The record location may be selected randomly from within the file. The general form of an invocation of a random unformatted OUTPUT standard routine is :

```
OUTPUT (file-number,rec-number,array-ident)
```

where

file-number is the file number obtained by the OPEN invocation.

rec-number is the record number within the file.
Note : the first record is number 0.

array-ident is an identifier associated with the BYTES data-element from which the value is to be transferred.

The parameter data types of the random unformatted OUTPUT standard routine are shown in the following routine declaration :

```
ROUTINE VOID,INTEGER ( INTEGER, INTEGER, BYTES )
```

Since the random unformatted OUTPUT standard routine has an out-value, the number of characters actually transferred by the routine, may be used within an expression.

9.6 OPEN_FILE

The OPEN standard routine will return a file number, corresponding to the named file, to be used within the program to execute input/output operations. An invocation of the OPEN standard routine will take the form :

```
OPEN (file-number,file-access,file-name,file-type)
```

where

file-number   is the file number obtained by invocation of the OPEN standard routine.

file-access   is the type of input/output which is to be executed with this file. For details see MON 50 in the SINTRAN Reference Manual.

file-name   is the SINTRAN file name.

file-type   is the SINTRAN file type.

Note : the default type is SYMB.

The data types of the formal parameters may be seen in the OPEN standard routine declaration :

```
ROUTINE VOID,VOID                                            &
  ( INTEGER READ WRITE, BYTES, BYTES, BYTES ) : OPEN (...)
```

9.7 CLOSE_FILE

The CLOSE standard routine will terminate the connection of a particular external file to an internal file number. An invocation of the CLOSE standard routine will take the form :

```
CLOSE (file-number)
```

where

file-number   is the internal file number within the program.

The data types of the formal parameters may be seen in the CLOSE standard routine declaration :

```
ROUTINE VOID,VOID (INTEGER) : CLOSE (...)
```

9.8 SET BLOCKSIZE OF A FILE

The BLOCKSIZE standard routine will set the blocksize of a file which has been previously OPEN'ed. The block size may be set to any number greater than or equal to 1. The form of the routine invocation is :

```
int BLOCKSIZE (file-number)
```

where

file-number   is the internal file number within the program.

int   is an integer identifier.

The value passed into the BLOCKSIZE standard routine must be the block size in bytes.

The data types of the formal parameters may be seen from the routine declaration :

```
ROUTINE INTEGER,VOID (INTEGER) : BLOCKSIZE (...)
```

9.9 SET/CHECK SIZE OF A FILE

The FILESIZE standard routine may be used either to set the size of a file, in bytes, or to inquire as to the present size of a file.

To set the size of a file, the form of the routine invocation is :

```
int FILESIZE (file-number)
```

where

file-number   is the internal file number within the program.

int   is an INTEGER4 identifier.

The value passed to the FILESIZE standard routine is the file size in bytes.

The parameter data types may be seen from the routine declaration :

```
ROUTINE INTEGER4,VOID (INTEGER) : FILESIZE (...)
```

If the file size is required, then the routine invocation should be :

```
FILESIZE(file-number)=:int
```

where

file-number   is the file number of the open file.

int   is an INTEGER4 identifier.

The parameter data types may be seen from the routine declaration :

```
ROUTINE VOID,INTEGER4 (INTEGER) : FILESIZE(...)
```

APPENDIX A

COMPILER COMMANDS

0.1 COMPILER_INVOCATION

The compiler is invoked from SINTRAN by the command :

```
@PLANC-100                on the ND-100
@PLANC-MC68

@ND-500 PLANC-500         on the ND-500
```

The compiler responds with a notification of the version in use. It then prompts by writing on the terminal :

```
*
```

which indicates that the compiler is in command mode.

The command names can be abbreviated. Only the number of letters needed to make it unique need be typed, but more may be given if required (ie. for readability or documentation). The parameters for a command can be written on the same line as the command name but separated from it by one or more blanks and at most one comma. Alternatively, if parameters are expected but not given, the compiler will prompt for them in turn.

Most commands may also be written as part of the source program, but in this case all parameters must be on the same line as the command name, and the command name must be preceded by a dollar character ($). Blanks may appear before the $ and the command name. Such commands can only be written between statements. They cannot occur in the middle of a statement, or between successive continuation lines of a statement.

0.2 COMPILATION_OF_SOURCE_PROGRAMS

The most important command is that which determines the program to be compiled and where the output is to be placed. This is written :

```
$COMPILE source list object
```

where

source — is the name of the file, or unit number, containing the PLANC program to be compiled. This parameter cannot be omitted. If TERMINAL or unit 1 is specified, input is accepted from the terminal, line by line, until a $EOF command is encountered in the input stream. If a file name is specified, it must obey the usual SINTRAN syntactic form and conventions. Default type is SYMB.

list — is the name of the file or unit number to which the source listing will be printed by the compiler. The format of the output will be suitable for printing and will contain the ASCII characters, line feed (LF), carriage return (CR), and form feed (FF) for carriage control.

If 0 is specified or the parameter is omitted, the listing is suppressed. The default list file type is SYMB.

object — is the name of the file, or unit number, which will contain the compiled relocatable version of the program. This is the input to the loader when creating an executable program. See the respective Loader manuals for details.

If 0 is specified, no relocatable code is generated, but a complete compilation takes place, thus giving any diagnostic messages that may occur.

The default type of the object file is BRF on the ND-100, NRF on the ND-500 and NRF for the MC68000.

If a unit number is given, it must be octal without any trailing letter B.

Any diagnostics generated by the compiler are listed on the terminal, and also on the list file, if they are not the same. The messages may be warnings or errors.

The end of the source text is the end-of-file or a $EOF encountered in the source file.

0.3 HELP

In command mode, the command :

```
        $HELP
```

will list all available commands together with their possible parameters.

HELP itself has no parameters.

0.4 COMPILER_TERMINATION

The command :

```
        $EXIT
```

will return control to SINTRAN after all source, list, and object files have been closed.

0.5 END_OF_FILE

The command :

```
        $EOF
```

signifies that the reading of the current file is complete. Reading continues at the next outer INCLUDE level.

0.6 IMMEDIATE_PREPARATION_OF_EXECUTABLE_PROGRAMS

This section applies to the ND-100 only.

An executable program may be prepared and output to a file, by using the command :

```
          $PROG-FILE file-name
```

where

file-name   is the name of a file to receive the executable program.

The default file type is PROG.

If the COMPILE command is used subsequent to the PROG-FILE command, then the compiler will generate the executable program directly on to this file. The COMPILE command will still generate an object file if it is specified, in addition to the PROG file.

The executable program is completed automatically, by loading the PLANC library (1 or 2 bank, depending on the setting of the $SEPARATE-DATA option), when the $EXIT is taken out of the compiler. A list of entry points and addresses will be output.

The $PROG-FILE command can be issued at most once during any invocation of the compiler.

To complete the executable program, libraries or other object files may be added by using the command :

```
          $LOAD file-name[,file-name]...
```

where

file-name   is the name of an object file or library.

The default type of the file loaded will be BRF.

Any error messages which appear while the $LOAD command is being executed can be found in the ND Relocating Loader manual (ND-60.066).

To define entry points in the loader table use the command :

```
        $DEFINE entry-name,value,mode
```

where

entry-name — specifies the name of an entry point. If an asterisk (*) is used, the current address will be used as the next load address. If a question mark (?) is used, a map of undefined entries will be output. If this parameter is blank, a map of defined entries will be output.

value — specifies the load address in octal.

mode — may be P to specify a program area, or D to specify a data area.

0.7 INCLUDING_TEXT_FROM_OTHER_SOURCE_FILES

Other files can be incorporated in the source program at the points indicated by the command :

```
        $INCLUDE filename
```

where

filename — is the name of the file or unit number to be read. The parameter cannot be omitted; the default file type is SYMB.

The reading of the source program by the compiler is switched to the named file and continues until a $EOF command is encountered. The file is then closed and the text following the $INCLUDE command is read. The named file may itself contain further $INCLUDE commands, but no more than 16 incomplete $INCLUDE's may be in existence at any one time.

For example, a number of separate modules may require the same user defined data type. The TYPE specification may be held on a file called COMDEF:SYMB. Then by writing :

```
        $INCLUDE COMDEF
```

at the appropriate point in each module, the TYPE specification is brought into the source file. Thus only one copy of the TYPE specification is kept, and all modules have identical copies of it.

0.8 COMPILE-TIME_CONSTANTS

In certain cases it may be desirable to set a parameter value to be tested by a $IF group of a command external to the source text being compiled, ie. prior to the outmost module level. The normal CONSTANT statement as defined in the PLANC language may be used in command mode.

0.9 CONDITIONAL_COMPILATION

It is possible to select parts of a file or program to be used in a particular compilation, depending on various parameters and their values. There is a set of commands which may be used for this purpose, within the source program. These commands are :

```
$IF    expression $THEN
  - PLANC source statements or compiler commands
$ELSIF expression $THEN
  - PLANC source statements or compiler commands
$ELSE
  - PLANC source statements or compiler commands
$ENDIF
```

where

expression — is an expression, which when evaluated, will give a result of TRUE or FALSE.

The expression may contain literals and constant identifiers as operands for any legitimate PLANC operators, eg. arithmetic and relational.

There may be zero or more instances of $ELSIF in a $IF command. The $ELSE may be omitted.

Within a group of commands, only those lines which lie between the first occurrence of <u>expression</u> which has the value TRUE, or the $ELSE command if all the expressions are FALSE, and the next command of the group, are included as valid source lines. The rest are listed, without line numbers, but are otherwise ignored.

The $IF groups may be nested to 11 levels.

All groups within INCLUDE'd text must be complete before the INCLUDE is terminated.

For example :

```
CONSTANT maxsize1=255,maxsize2=32767
CONSTANT size=1000
$IF size <= maxsize1 $THEN
  INTEGER1 : index
$ELSIF size > maxsize1 AND size <= maxsize2 $THEN
  INTEGER2 : index
$ELSE
  INTEGER4 : index
$ENDIF
```

In this case, size has a value 1000 which will result in the line of code, INTEGER2 ... being included in the compilation.

0.10 COMPILE_TIME_MACROS

Another method of conditional compilation is to define a macro, which may be invoked within the source lines, and then substitute text where macro name appears. Parameters may be used within the macro expansion to control the particular text output from the macro.

The general form of a macro definition is :

```
        $MACRO macname [(parameter[,parameter]... )]
          macro body
        $ENDMACRO
```

where

macname is the name to be used to invoke the macro.

parameter is a valid identifier name.

macro body is text to be expanded by a macro invocation.

The macro name must be formed according to the rules for PLANC identifiers. It will be used to invoke the macro from within the source lines of code.

The names of formal parameters of the macro definition are formed according to the rules for PLANC identifiers. Within the macro body the value of each formal parameter may be referenced during macro expansion, by the formal parameter name enclosed by double quote characters ("). The double quote character may not be used for any other purpose within the macro body.

The macro body may contain text which will be output unchanged during the macro expansion, or modified by substitution of thé value of actual parameters. It may also contain other compiler commands, eg. $IF ... $ENDIF, with the exception of another $MACRO command, ie. nested macro definitions are not allowed. However, it should be noted that any compiler commands within a macro will be carried out at the time that the macro is being expanded, and its output going into the source of the PLANC program, prior to compilation of the PLANC source code.

Example of a macro definition :

```
        $MACRO exmac (param1,param2)
          "param1""param2" $ENDMACRO
```

An actual parameter may be any text string of characters, not including the comma, right parenthesis or double quote characters, ie. , or) or " characters. However, if a comma or a right parenthesis is required within an actual parameter, the entire actual parameter must be enclosed by double quote characters. The actual parameter value will be substituted wherever it has been referenced within the macro body.

For example, the above macro definition may be invoked by the following :

```
exmac(INTEGER,2) : i,j
exmac(REAL,4) : r,s
```

will generate

```
INTEGER2 : i,j
REAL4 : r,s
```

The macro body may contain macro invocations, ie. macro invocations may be nested. Macro invocations may be recursive, ie. a macro may invoke itself from within its own macro body.

0.11 CROSS_REFERENCE_LISTING_AND_LINKAGE_INFORMATION

The command may be used for obtaining an identifier cross-reference listing :

```
        $CROSS-REFERENCE filename
```

where

filename   is the name of a file to be used as a temporary work area. The default file type is XREF. The file must be on a mass storage device.

This command will list all the identifiers and the line numbers where they are used. The output is on the listfile, and it follows the source listing.

A list of the routine call hierarchy may be obtained by using the command :

```
        $CALL-HIERARCHY ON
```

and this option may switched off by,

```
        $CALL-HIERARCHY OFF
```

The CALL-HIERARCHY listing follows the source listing and precedes the cross-reference listing if it is present. The initial value is OFF.

Detailed linkage information may be obtained with the command :

```
        $LINKAGE-REFERENCE file-name
```

where

file-name   is the name of a work file.

This command will produce a sorted list of all EXPORT'ed/IMPORT'ed items from the outermost module level. Use of the LINKAGE-REFERENCE command, prior to one or more $COMPILE commands, will cause a return to command mode after each compile.

The layout of the list output is as follows :

```
End of source listing
******************************************************************

ROUT1     A_LINKR*  B_LINKR
ROUT2     B_LINKR*  A_LINKR
VAR1      A_LINKR*  B_LINKR
VAR2      B_LINKR*  A_LINKR
```

If an item is EXPORT'ed from a module, the module name will be marked with an asterisk (*).

The LINKAGE-REFERENCE command and the CROSS-REFERENCE command must not be used together in one compile.

0.12 LISTING CONTROL

The listing of source lines on the listfile may be controlled by the use of the command :

```
$LIST ON
```

will cause lines of the source text to be output to the listfile. It resumes the listing from a previous LIST OFF command.

```
$LIST OFF
```

will suppress output going to the listfile. The initial value is ON.

A skip to a new page may be requested by using the command :

```
$EJECT
```

which will output a form feed to the listfile.

The line numbers printed in the source listing may be changed, in order to continue from a different number by using the command :

```
$LINE-BIAS line-number
```

where

line-number is the number to continue line numbers from.

0.13 RUN TIME OPTIONS FOR THE ND-100

The execution of a PLANC program may be modified by the following options provided by the compiler.

The code and data of a program may be generated for separate memory areas by the use of the command :

```
$SEPARATE-DATA ON
```

and this option may switched off by,

```
$SEPARATE-DATA OFF
```

The initial value is OFF.

The extra instructions of the ND-100/CE model may be generated by the use of the command :

```
$ND100-EXTENDED ON
```

and this option may be switched off by,

```
$ND100-EXTENDED OFF
```

The initial value is OFF.

Optimization of memory requirements and execution speed will be attempted by the compiler with the following option :

```
$OPTION SQUEEZE ON
```

and this option may be switched off by,

```
$OPTION SQUEEZE OFF
```

The initial value is OFF.

Each access to an array element will be checked at either compile time or during execution with the following option :

```
$OPTION ARRAY-INDEX-CHECK ON
```

and this option may be switched off by,

```
$OPTION ARRAY-INDEX-CHECK OFF
```

This option may be used in several places in a program to switch checking on and off, as required.

The initial value is OFF.

0.14 DATA_TYPE_DEFAULTS

The number of significant digits of the REAL data type may be altered by using the command :

```
          $REAL-PRECISION number
```

where

number is the number of significant digits required.

0.15 CREATION_OF_LIBRARIES

To create a library from one or more outer level modules in one compilation, use the command :

```
          $LIBRARY-MODE ON
```

and this option may be switched off by,

```
          $LIBRARY-MODE OFF
```

The LIBRARY option will generate a preceding BRF or NRF library mark for each outer level module in the compiled file. The loader will not load a module unless there is an unresolved reference to an EXPORT'ed identifier in the module.

If the EXPORT'ed identifier has one or more ALIAS names, an ALIAS must be present in the EXPORT statement as well as in each relevant routine declaration. Further, the ALIAS in the EXPORT statement may use the following general form :

```
          EXPORT ... ALIAS 'name' [OR 'name']...
```

The list of ALIAS names is not permitted in an ALIAS used for a normal routine declaration.

If OFF is used, these library marks are suppressed and the loader will load the module anyway. The initial value is OFF.

For details of library marks and files see ND Relocating Loader manual (ND-60.066) on the ND-100, or the ND-500 Loader/Monitor manual (ND-60.136) on the ND-500.

0.16 DEBUGGING

The output from the compiler can be made to include information for use by the Symbolic Debugger. In order to have the debug information generated by the compiler use the command :

```
        $DEBUG-MODE ON
```

and this option may be switched off by,

```
        $DEBUG-MODE OFF
```

For detailed descriptions of how to use the facilities of the Symbolic Debugger see the Symbolic Debugger Reference Manual, ND-60.160 . The initial value is OFF.

0.17 ASSEMBLER_CODE_IN_PLANC_PROGRAMS

Assembly code may be placed within PLANC source statements and it will be translated by an in-line assembler for the appropriate target machine.

Assembly code lines must begin with a dollar character ($) followed by an asterisk character (*). Mulitple instructions on one line are separated by a semicolon character (;).

The syntax of machine instructions submitted to the inline assembler is described in the following manuals :

ND-100 Reference Manual	ND-06.020
ND-500 Reference Manual	ND-05.010

MC68000 16 BIT MICROPROCESSOR User's Manual (third edition)

Chapter 2, Appendices A and B in the MC68000 manual are of particular relevance.

PLANC declared variables or labels may be used as operands in assembler instructions and the in-line assembler will generate the appropriate references. However the PLANC identifiers, used in the assembler instructions, must be used without the special addressing mechanisms, eg. base registers or indirect, as these will be generated for each PLANC identifier.

Beware of possible name conflicts between PLANC identifiers and assembler mnemonics, eg. I for indirection in the ND-100.

Examples :

```
        $* LDA 0,X; SAD SHR 20; SAT 4; RDIV ST          % ND-100 Code

        $* W1 DIV4 B.24B:S,4,W2                         % ND-500 Code

        $* MOVE 22B(A6),D0; EXT.L D0; DIVS #4B,D0       % MC68000 Code
```

0.18 DATE_COMMAND

The DATE command puts todays date (of the compilation) into a string. The date is in the following format :

```
        month dd, 19yy
```

and may obtained by the following declaration,

```
        BYTES READ : date:= $DATE  % a blank must precede the $
```

For example the identifier date will receive a string as follows :

```
        DECEMBER 25, 1947
```

0.19 OPTION_COMPILER_COMMAND

The OPTION command is used to switch on or off some optional facilities of the PLANC compiler. These facilities have been described in this chapter. The general form of this command, to switch an option on is :

```
        $OPTION option-name ON
```

and to switch an option off is :

```
        $OPTION option-name OFF
```

The options available are :

1) HELP

2) SQUEEZE

3) ARRAY-INDEX-CHECK

APPENDIX B

ERROR MESSAGES

O.1 COMPILER_MESSAGES

AMBIGUOUS COMMAND

Abbreviation of the command name has resulted in a non-unique command name.

ARRAY BOUNDS CONFLICT WITH A PREDECLARATION

No further explanation.

ARRAY BOUNDS MISSING

An array declaration must have explicit array bounds, unless initial values imply the array bounds.

COMMAND NOT PERMITTED WITHIN A MODULE

Certain compiler commands must only be used as global to the outermost module level.

CONFLICTING DATA TYPES IN CORRESPONDING IMPORT/EXPORT

The corresponding IMPORT/EXPORT statements of communicating modules have different data types in the declaration of one data-element.

DATA TYPE NOT PREVIOUSLY SPECIFIED

An identifier has been used as a user defined data type without a type specification.

EQUIVALENCE MAY CAUSE STORAGE CONFLICT

The use of equivalence (=) here for overlapping data-elements could cause storage conflicts because of different length or storage layout of different data types. (ND-100 only)

EXITFOR ALREADY PRESENT WITHIN THE LOOP

There is already one EXITFOR within this FOR-ENDFOR loop.

EXITWHILE ALREADY PRESENT WITHIN THE LOOP

There is already one EXITWHILE within this loop.

EXPONENT IS TOO LARGE

See Appendix D.

EXPORTED IDENTIFIER IMPORTED IN AN OUTER MODULE

No further explanation.

EXPRESSION DOES NOT STORE A VALUE

No further explanation.

IDENTIFIER ALREADY SPECIFIED/DECLARED
The identifier has already appeared in a declaration statement or a type specification statement.

IDENTIFIER IN EXPORT, BUT NO DECLARATION
The identifier which has been used in an export statement has not been declared within this module.

IDENTIFIER USED IN DOT NOTATION IS NOT A RECORD
No further explanation.

IDENTIFIER USED IN DOT NOTATION IS NOT A RECORD COMPONENT
No further explanation.

ILLEGAL CHARACTER
A character has been used in a context in which it is not allowed, eg. a digit as the first character of an identifier name or a real exponent containing a non-numeric character.

ILLEGAL CONSTRUCTION OF $IF-$ENDIF COMMAND
No further explanation.

ILLEGAL CONTROL IDENTIFIER
The data type of the control identifier of the FOR statement does not match the data type of the FOR list values.

ILLEGAL DATA-ELEMENT TO BE CONVERTED
The size of the data-element referred to by a FORCE or CONVERT standard routine does not match the target data type. There may be no conversion routine available.

ILLEGAL DATA TYPE
The data type of an identifier has been used illegally.

ILLEGAL FORMAL PARAMETER IN MACRO
A macro definition parameter list contains an identifier name which conflicts with a previous declaration.

ILLEGAL INLINE INVOCATION
It is illegal to have an invocation of an INLINE routine within another INLINE routine, ie. nested INLINE invocations are not allowed.

ILLEGAL MODULE TERMINATION
The module structure has not been correctly terminated by an ENDMODULE statement.

ILLEGAL NESTED MACRO DEFINITION
No further explanation.

ILLEGAL PARAMETER REFERENCE IN MACRO BODY
When referring to a macro parameter within the macro body, the parameter must be bounded by double quote characters.

ILLEGAL PREDECLARATION
The predeclared identifier has appeared previously in a predeclaration statement, or it may not be used in this context.

ILLEGAL OPERAND FOR STORE OPERATOR
No further explanation.

ILLEGAL SYNTAX
The compiler has been unable to correctly translate this statement. This may be due to a missing or misplaced delimiter, misspelled keyword or scope problems.

ILLEGAL TO EXPORT THIS IDENTIFIER
No further explanation.

ILLEGAL TO IMPORT THIS IDENTIFIER
No further explanation.

INCASE CONTAINS INVALID VALUE
The INCASE part of a CASE statement has either an invalid value, eg. which is not a member of the set being used, or a value which has occured in a previous INCASE of this CASE statement.

INCOMPATIBLE DATA TYPES
A pointer data-element must be initialized to its corresponding data type.

INCONSISTENT DIMENSIONS
The index set(s) in an array declaration do not correspond to the number of array keywords in the declaration.

INISTACK INVOCATION MISSING
A PROGRAM routine must contain an INISTACK invocation to initialize the stack area at run-time.

INITIALIZATION VALUES OVERFLOW DECLARED SIZE
The number of elements declared for an array is less than the number of values to be initially placed in this array.

INSUFFICIENT BUFFER SPACE FOR COMPILER
The compiler has insufficient buffer space, eg. for macro definitions, expansions or INLINE routine declarations or invocations.

INITIAL VALUE ILLEGAL HERE
No further explanation.

INVALID ACTUAL PARAMETER, FORMAL PARAMETER DECLARED AS WRITE
The actual parameter in the routine invocation is invalid because the formal parameter in the routine declaration has been declared as WRITE or READ WRITE.

INVALID ARRAY FOR INISTACK INVOCATION
The array in the INISTACK invocation must be global or imported, declared with one dimension only and a lower bound of zero.

INVALID COMMAND
No further explanation.

INVALID CONDITIONAL EXPRESSION
No further explanation.

INVALID PARAMETER
An invalid parameter has been used in a compiler command.

INVALID PARAMETER LIST
In a routine declaration the number of formal parameters does not match the declared data types. In a macro invocation, the number of parameters is incorrect.

INVALID TYPE FOR IN-VALUE/OUT-VALUE/PARAMETER
The data type of a routine in-value, out-value or parameter must not be a routine. Note that a pointer to a routine data-element may be used.

INVALID USE OF KEYWORD
A valid keyword has been used in a statement illegally.

LINE IS TOO LONG
No further explanation.

MAX. NO. OF ARRAY ELEMENTS EXCEEDED
The number of elements declared for an array has exceeded the compiler's available memory space. (ND-100 only)

MISPLACED $ENDMACRO COMMAND
No further explanation.

MISPLACED STATEMENT
It is not legal to have this statement at this point in the program.

MISSING KEYWORD, ENDIF/ENDCASE/ENDFOR/ENDDO OR ENDON
No further explanation.

MORE SUBSCRIPTS THAN IN THE ARRAY DECLARATION
No further explanation.

MULTIDIMENSIONAL ARRAY NOT ALLOWED HERE
In some statements an array is allowed, but only a one dimensional array.

NEGATIVE BOUND ILLEGAL
No further explanation. (ND-100 only)

NO MORE SPACE FOR LOCAL DATA-ELEMENTS
No further explanation. (ND-100 only)

NOT IMPLEMENTED
No further explanation.

NOT PREVIOUSLY DECLARED
An identifier has been used without a declaration of an associated data-element, or without a type specification.

QUALIFIER REQUIRED FOR THIS RECORD COMPONENT
This record component identifier has been specified in more than one record. Consequently a record identifier must be used as a qualifier to uniquely reference the desired component data-element.

REQUIRE ELSE OR ALL POSSIBLE VALUES USED IN INCASE PARTS
A CASE statement must include all possible values in its INCASE parts, or an ELSE must be present.

INVALID TYPE FOR IN-VALUE/OUT-VALUE/PARAMETER
The data type of a routine in-value, out-value or parameter must not be a routine. Note that a pointer to a routine data-element may be used.

ROUTINE WITH AN OUT-VALUE REQUIRES A RETURN
A routine which is declared with an out-value must contain at least one return statement.

SET MEMBER OVERLAP
A set member value has been used more than once in initializing the set data-element.

SQUEEZE OPTION GENERATES INCORRECT CODE FOR THIS ROUTINE
Optimization of this routine generates incorrect execution code. The SQUEEZE option must be switched off in order to compile this routine correctly. (ND-100 only)

STORAGE OVERFLOW IN COMPILER
No further explanation.

TARGET MACHINE ADDRESS IS TOO LARGE
During a cross-compilation an address for the target machine is required, but is too large for the compiler on this machine.

TOO MANY LEVELS OF MODULE NESTING
This is limited by the space available to the compiler.

TOO MANY NESTED INCLUDES, MACRO/INLINE EXPANSIONS
There are too many nested INCLUDE's, nested macro expansions or INLINE routine invocations for the storage available to the compiler.

UNABLE TO EVALUATE EXPRESSION AT COMPILE-TIME
The expression contains identifiers whose values are not constant at compile time.

WRITE DECLARATION ILLEGAL IN READ ONLY RECORD
If a record data-element has been declared as READ only, its component data-elements must not be declared as WRITE only.

0.2 RUN-TIME MESSAGES

- NO ON ROUTINEERROR HANDLER, ERRETURN= value
  A routine has taken an ERRETURN exit and there is no exception handler specified to which control can be passed. The ERRETURN value may have been set in the user code or it may be from SINTRAN, see the SINTRAN Reference Manual (ND-60.128).

- ASSERT VIOLATION AT address
  If the condition in an ASSERT statement is evaluated, and gives a resulting value FALSE, and the program has no ON ASSERTFALSE exception handler, the program has terminated execution at the 'address' in the message.

- STACK OVERFLOW AT address
  The requirements for storage have exceeded that available, and the program has no ON STACKERROR exception handler, so the program has terminated execution at the 'address' in the message.

APPENDIX C

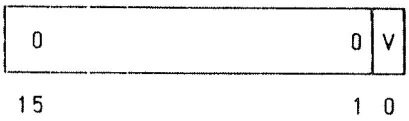
MACHINE DEPENDENT LANGUAGE FEATURES IN PLANC

```
                2 bytes ND-100
    word size = 2 bytes MC68000
                4 bytes ND-500
```

0.1 STORAGE MAPPING

PLANC data-elements are stored in the following way :

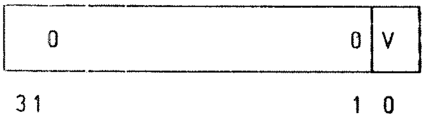
BOOLEAN (ND-100/MC68000)



Bits 15-1 : set to 0

Bit 0 (V) : 0 = FALSE
1 = TRUE

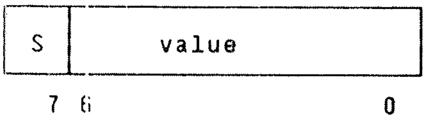
BOOLEAN (ND-500)



Bits 31-1 : set to 0

Bit 0 (V) : 0 = FALSE
1 = TRUE

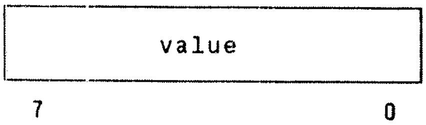
INTEGER1



Bit 7 : 0 = greater than or equal to zero
1 = negative

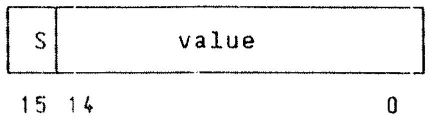
Bits 6-0 : value held in twos-complement form

BYTE



Bits 7-0 : unsigned integer value

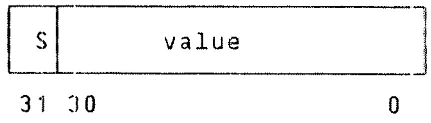
INTEGER2



Bit 15 : 0 = greater than or equal to zero
1 = negative

Bits 14-0 : value held in twos-complement form

INTEGER4



Bit 31 : 0 = greater than or equal to zero
1 = negative

Bits 30-0 : value held in twos-complement form

INTEGER RANGE

Data types whose base type is integer range, will require storage for each data-element depending on the values specified for the upper and lower bounds. Each data-element will be allocated the smallest available addressable unit which has enough bits to contain the next higher power of 2, greater than the number of values in the specified range.

On the ND-100 the addressable units used are 1 word (16 bits) and 2 words (32 bits). On the ND-500 the addressable units used are 1 byte (8 bits), 2 bytes (16 bits) and 4 bytes (32 bits). On the MC68000 the addressable units used are 1 byte (8 bits), 2 bytes (16 bits) and 4 bytes (32 bits).

For example :

```
INTEGER RANGE (0:32)
```

will require 6 bits to hold 33 distinct values. On the ND-100 1 word will be used, ie. 16 bits. On the ND-500 and MC68000 1 byte will be used, ie. 8 bits.

If INTEGER RANGE is the base type of a SET data-element, then the data-element may have waste bits depending on the range specified. In the above example 31 bits of space would be wasted in each data-element, ie. a 64 bit data-element is allocated, although only 33 bits are used.

If a CASE statement uses an INTEGER RANGE for its multiple possibilities, then bits may be wasted in the same way as in a SET data-element. A number of words (in a table of addresses) may be wasted, ie. the size of the table of addresses will be a power of two entries.

ENUMERATION

An ENUMERATION data-element will occupy one word on the ND-100/MC68000 and the ND-500 respectively. The data-element will contain an integer value corresponding to the position in the list of possible ENUMERATION values declared. The first ENUMERATION value will be counted as zero. Hence the maximum possible number of distinct ENUMERATION values in one declaration is 32768 for the ND-100/MC68000 and 2147483648 for the ND-500.

POINTER

Pointer data-elements for all data types, except arrays, will occupy one word (2 bytes) on the ND-100 and 4 bytes on the ND-500 respectively. On the MC68000, a pointer data-element will occupy two words (4 bytes).

Since the ND-100 has word addressing only, for array elements or record components which are smaller than one word, a pointer to an array element or a record component will contain the address of the word containing the data-element. It will not necessarily be the exact address of the data-element. However for statements such as expressions with assignment operators, the run-time system will access the data-element correctly.

This may affect addressing of array elements where the elements are smaller than one word, eg. INTEGER RANGE (0:7) PACKED, or components of a packed record.

On the ND-500/MC68000 byte addressing is available, so a pointer, to data-elements which are array elements or record components, may contain a byte address.

ARRAY POINTER

An array pointer will require 3 pieces of information per dimension declared for the array. But the first element of the three, for the first declared dimension, is an address (a pointer data-element). All the rest of the elements are default integer data-elements. For example a two dimensional array will have 6 elements in its array pointer data-element, the first of which is an address.

Following is a diagram of the layout of an array pointer data-element. Each part is a default integer size except the first which is an address.

address	used for computing element addresses
lower bound 1	first dimension
upper bound 1	
constant1	1+ upper bound 1 - lower bound 1
lower bound 2	second dimension
upper bound 2	
constant2	1+ upper bound 2 - lower bound 2
etc	

An array may be declared with n dimensions as follows :

```
        ar(low1:high1,low2:high2,...).
```

The address in the first element of the array descriptor, ie. the array pointer, is used for computing addresses of any element of the array. This address is an imaginary point in memory, which is obtained by setting each index to zero, regardless of the declared bounds. This imaginary point in memory would be the address of the first element of the declared array, if all of its lower bounds were declared as zero.

The address, of the imaginary point in memory, is obtained by computing an offset and subtracting it from the actual memory address, where the first element of the array is located. The following formulae may be used to compute the offset, in array element units:

```
        low1*constant1+low2                        2 dimensional array
and     (low1*constant1+low2)*constant2+low3       3 dimensional array
```

and so on for arrays of more dimensions.

The result from the above formulae, in array element units, must be multiplied by the length of an array element in machine addressing units, ie. bytes for the ND-500 and words for the ND-100. Note that on the ND-100 a byte occupies one half word, and consequently the result of the formulae must be even to give a valid address offset. Beware that the data type, of the array elements of a PACKED array, may modify the computation by which array element units are converted to machine addressing units.

The above formulae may also be used for computing the address of any element of the array. Substitute each subscript value for the corresponding lower bound values, and the formulae will give an offset in array element units. This offset must be converted to machine addressing units and then added to the address in the first word of the array descriptor, giving the address of a specific array element.

<u>REAL</u> (32-bit floating-point hardware)

S	exponent	mantissa
31	30 22	21 0

Bit 31 : 0 = greater than or equal to zero
1 = negative

Bits 30-22 : Binary exponent
Stored with a bias of 256 (400 octal). This is a power of 2 that the mantissa must be multiplied by. A value of 256 means that the mantissa is the value.

If the exponent is 0, the whole value is zero.

Bits 21-0 : mantissa
Stored without the 0.5 (0.1 binary) excess, unless the value is zero. The binary point is one place to the left of the mantissa. The mantissa is normalised so that
0.5 <= mantissa < 1.0

This gives an accuracy of 7 significant digits.

<u>REAL</u> (48-bit floating-point hardware)

S	exponent	mantissa
47	46 32	31 0

Bit 47 : 0 = greater than or equal to zero
1 = negative

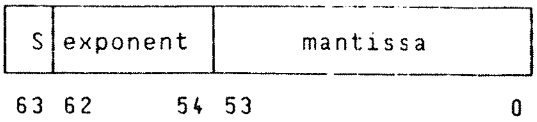
Bits 46-32 : Binary exponent
Stored with a bias of 16384 (40000 octal). This is a power of 2 that the mantissa must be multiplied by. A value of 40000B means that the mantissa is the value.

If the exponent is 0, the whole value is zero.

Bits 31-0 : mantissa
Stored with <u>all</u> bits included. The binary point is immediately to the left of bit 31.

This gives an accuracy of 9 significant digits.

REAL8 (64-bit floating-point)



Bit 63 : 0 = greater than or equal to zero
1 = negative

Bits 62-54 : Binary exponent
Stored with a bias of 256 (400 octal). This is a power of 2 that the mantissa must be multiplied by. A value of 256 means that the mantissa is the value.

If the exponent is 0, the whole value is zero.

Bits 21-0 : mantissa
Stored without the 0.5 (0.1 binary) excess, unless the value is zero. The binary point is one place to the left of the mantissa. The mantissa is normalised so that

0.5 <= mantissa < 1.0

This gives an accuracy of 15 significant digits.

ARRAY

The storage required for an array data-element is simply the number of elements declared times the storage required for one element of the array.

The array elements are stored in ascending order of the subscript values. Arrays of more than one dimension are stored with the last index changing most rapidly. This is identical to the scheme used for PASCAL and different from the scheme used in Fortran.

The maximum number of elements of an array is limited by the way subscripts are stored internally. Subscripts are stored in a signed default integer data-element. Hence, on the ND-100/MC68000, the maximum number of elements which an array may be declared with (this depends on the number of dimensions and the upper and lower bound values of each dimension), is 32K, ie. 32768.

On the ND-100, a PACKED ARRAY which is declared with 8-bit integer elements, must not have a negative lower bound in any of its index sets.

On the ND-100, note that due to the scheme of computing the memory addresses of array elements, the declared lower index bounds must result in

1) the first element of a PACKED INTEGER modified array being an odd byte, and

2) the first element of a PACKED BOOLEAN array being any bit within a word.

This may be achieved on the ND-100, for arrays of two or more dimensions having elements smaller than one word in the following way. The lower bound of the last dimension and the number of values in the index set, must be a multiple of the number of elements per word.

RECORD

The storage required for a record data-element is simply the total storage required by all the component data-elements, plus any waste space between the component data-elements due to the alignment requirements of each component.

SET

A SET data-element will have one bit per possible member, ie. the data-element will require the number of bits corresponding to the maximum number of members declared. The bits are grouped into words. If a bit has the value one, then that corresponding possible value is a member of the SET data-element.

0.2 STORAGE ALIGNMENT

The following tables give the size in bytes and storage alignment of each of the different data-element data types on each machine.

Table 1 : ND-10 or ND-100 with 48-bit Floating-point Hardware

data type	length in bytes	alignment(Note 1)
BOOLEAN	2	word
INTEGER (INTEGER2)	2	word
INTEGER1	1	word
BYTE	1	word
INTEGER2	2	word
INTEGER4	4	word
REAL	6	word
REAL8 (Note 2)	8	word
ENUMERATION	2	word
ARRAY	variable	word
RECORD	variable	word
SET	2*((members+15)/16)	word

Table 2 : ND-10 or ND-100 with 32-bit Floating-point Hardware

All data types not listed in table 2 are the same as in table 1.

data type	length in bytes	alignment(Note 1)
REAL	4	word

Table 3 : ND-500

data type	length in bytes	alignment(Note 1)
BOOLEAN	4	word
INTEGER (INTEGER4)	2	word
INTEGER1	1	byte
BYTE	1	byte
INTEGER2	2	half-word
INTEGER4	4	word
REAL8	8	word
ENUMERATION	4	word
ARRAY	variable	word
RECORD	variable	word
SET	2*((members+15)/16)	word

Table 4 : MC68000

data type	length in bytes	alignment(Note 1)
BOOLEAN	2	word
INTEGER (INTEGER2)	2	word
INTEGER1	1	byte
BYTE	1	byte
INTEGER2	2	word
INTEGER4	4	word
REAL8	8	word
ENUMERATION	2	word
ARRAY	variable	word
RECORD	variable	word
SET	2*((members+15)/16)	word

Notes

1. For the ND-10 and the ND-100, a word is 16 bits, ie. 2 bytes. For the ND-500, a word is 32 bits, ie. 4 bytes. For the MC68000 a word is 16 bits, ie. 2 bytes.

2. The REAL8 data type is identical on all machines. For the ND-10 and the ND-100 the implementation is by software routines and is relatively slow.

0.3 PACKED OPTION

The PACKED option may be used on arrays and records. It will affect the alignment of the data-elements of simple data types within the composite data-element.

In arrays and records, each BOOLEAN data-element will be stored in 1 bit.

In arrays using the PACKED option INTEGER RANGE and ENUMERATION component data-elements will have space allocated as described for INTEGER RANGE data-elements. One exception is for ARRAY PACKED data-elements on the ND-100, where the smallest unit of space used is 1 byte (8 bits).

In records using the PACKED option INTEGER RANGE and ENUMERATION component data-elements will require the next higher power of 2 bits, greater than the number of bits necessary, to hold the required values.

In a PACKED record, no data-elements of the simple data types will be split across a word boundary.

The following declarations :

```
TYPE minrec = RECORD PACKED
                INTEGER RANGE (-8:7) : ir % requires 4 bits
                BYTE : onebyte          % requires 8 bits
                BOOLEAN : flag          % requires 1 bit
                ENUMERATION (a,b,c) : ch  % requires 2 bits
                BYTES : chars(0:4) % require 8 bits/element
              ENDRECORD
```

would require the following storage on the ND-100 :

15	11	7	3 0	
ir(4)	onebyte(8)		f(1)ch(2) W	1 bit waste
chars(0)	(8 bits)	chars(1)		no waste
chars(2)		chars(3)		no waste
chars(4)		W........	..........W	8 bits waste

Note the BYTES array chars has an implicit PACKED within the predefined data type. This causes its elements to be stored two to a word here. If this array had been declared as INTEGER RANGE (0:255), then its elements would have been stored one to a word within the above record.

The above record would require the following storage on the ND-500 :

31 15 0

ir(4) onebyte(8) f(1) ch(2) W	chars(0) (8 bits) chars(1)
chars(2) chars(3)	chars(4) next data-elem.

The above record would require the following storage on the MC68000 :

15 11 7 3 0

ir(4)	onebyte(8)		f(1) ch(2) W	1 bit waste
chars(0)	(8 bits)	chars(1)		no waste
chars(2)		chars(3)		no waste
chars(4)		next data	element	

The PACKED option used on an array or record, only affects alignment of entire composite data type data-elements declared within the array or record. The PACKED option may be used on an array or record declared as an array element or record component. Thus in the above examples, the array chars is word aligned on the ND-100 and byte aligned on the ND-500.

APPENDIX D

MIXED LANGUAGE PROGRAMMING

0.1 INTRODUCTION

PLANC has a standard calling sequence for its routine invocations. This will facilitate the interfacing of programs and subprograms written in other languages and those written in PLANC. This interface is described in detail first and then examples showing how to use it to interface to other languages on both the ND-100 and ND-500 follow.

The following general advice should be observed for interfacing to PLANC routines :

1) All PLANC routines invoked by another language routine should be STANDARD.

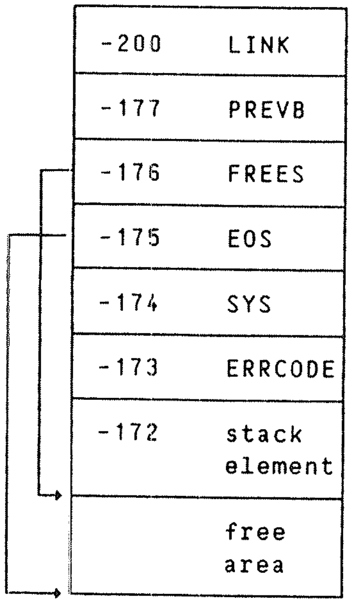
2) All routines IMPORT'ed into a PLANC routine should be STANDARD.

Each PLANC routine holds its local variables in a local data area. If a program comprises a number of routines, the local data area for each subprogram may be dynamically allocated from a single stack, or from multiple stacks created by INISTACK invocations. The B-register must always address the appropriate stack element during execution of a PLANC routine.

The actual parameter list of STANDARD routines consists of a sequence of words, one for each parameter. For explicit data-elements or expressions with a temporary data-element, the corresponding word contains the address of the data-element. For arrays the word contains the address of the imaginary element of the array, with all indexes set to zero, which is used for computing memory addresses of each array element.

If a number of routines are written in a language other than PLANC, it may be necessary to have two or more routines with the same ALIAS name, but each routine having a different number of parameters. While this is not allowed in PLANC, IMPORT statements may be written for such a group of routines, written in some other language, in order to invoke the routine accordingly.

0.2 INTERFACING WITH PLANC ON THE ND-100



offset from the B-reg (octal) in bytes		content
-200	LINK	link register, address for normal return
-177	PREVB	previous B-register, Reloaded on exit
-176	FREES	- points to the free area of stack which immediately follows this stack element
-175	EOS	- points to the word immediately following the whole stack
-174	SYS	- run-time system use
-173	ERRCODE	ERRCODE (value)
-172	stack element	first parameter address if any
	free area	free area of the stack

When PLANC invokes a STANDARD routine, the registers are used as follows :

L = return address
B = current stack element; must be restored on return
T = number of parameters
A = parameter list address
D = unused
X = unused
P = entry point of called routine

On return from a routine, the out-value of the routine is returned as follows :

BOOLEAN,BYTE,INTEGER1,INTEGER2, ENUMERATION	A-register
INTEGER4	AD-register
REAL (32-bit floating-point hardware)	AD-register
REAL (48-bit floating-point hardware)	TAD-register
REAL8	A-register points to the result
POINTER	A-register
RECORD,ARRAY,SET (address)	A-register

If a routine has [expression] ERRETURN, the value of expression is set in the ERRCODE position in the invoker's stack element.

For two-bank programs, all parameter values and their descriptors must be in the data bank.

The PLANC Run-time Entry and Exit Routines

The heading of a PLANC routine contains an invocation of either 5INIT or 5ENTR in the run-time system in order to establish a new stack entry of a required size. 5INIT is invoked from main programs and from routines with INISTACK invocations, otherwise 5ENTR is called.

The tasks of 5INIT and 5ENTR are :

- Establish the stack entry with sufficient space.
- Save the return address, in LINK. In a main program the return address is set to 5QUIT.
- Save the previous B-register value (PREVB).
- Update the free stack pointer (FREES) and end of stack address (EOS).
- Check for stack overflow.

On routine exit either 5LEAV, for RETURN and ENDROUTINE, or 5ERET, for ERRETURN, are invoked.

The tasks of 5LEAV and 5ERET are :

- Restore the B-register to its value upon entry to the routine.
- Return to the location following the JPL instruction of the actual invocation, through 5ERET, or skip to the next location, through 5LEAV. The location following the JPL instruction contains either a jump to ROUTINEERROR group of statements or another 5ERET invocation.

Example of a main program layout in MAC equivalent :

```
         SAX    0              % main entry
         JPL    I (5INIT
         .                     % stack space requirement
         .                     % stack array address
         .                     % entire stack array size in words
         .                     % two bank flag
         0                     % unused
     %
     % executable code
     %
         JPL    I (5LEAV       % exit from routine
```

Example of layout of a routine containing an INISTACK invocation in MAC equivalent :

```
         COPY SL DX            % return address
         JPL    I (5INIT
         .                     % stack space requirement
         .                     % stack array address
         .                     % entire stack array size in words
         .                     % two bank flag
         0                     % unused
     %
     % executable code
     %
         JPL    I (5LEAV       % exit from routine
```

Example of routine layout (not containing an INISTACK invocation) in MAC equivalent :

```
         COPY SL DX            % return address
         JPL    I (5ENTR
         .                     % stack space requirement
     %
     % executable code
     %
         JPL    I (5LEAV       % exit from routine
```

0.3 INTERFACING WITH PLANC ON THE ND-500

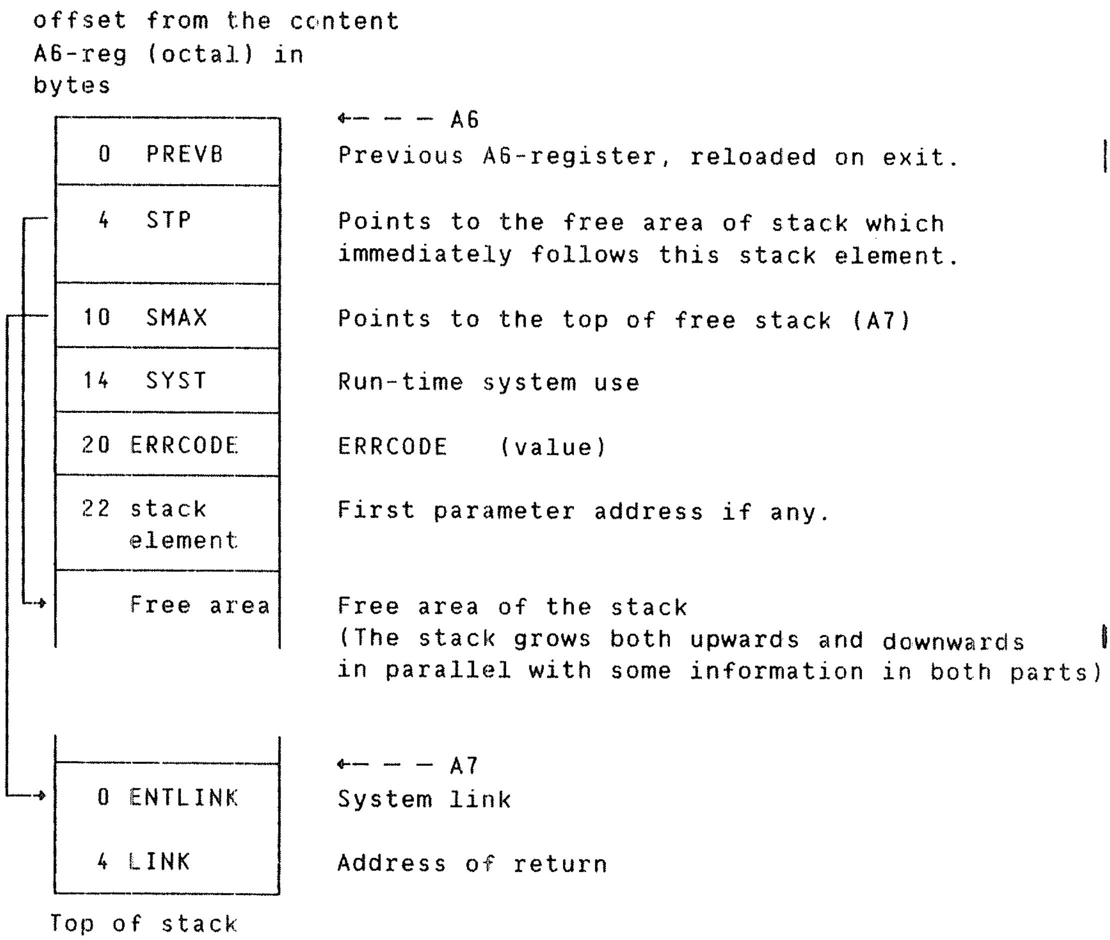
offset from the B-reg (octal) in bytes		content
0	PREVB	previous B-register, Reloaded on exit.
4	RETA	link register, address for normal return.
10	FREES	points to the free area of stack which immediately follows this stack element.
14	ERRCODE	ERRCODE (value).
20	N	number of parameters.
24	stack element	first parameter address if any.
	free area	free area of the stack.

On return from a routine, the out-value of the routine is returned as follows :

BOOLEAN,BYTE,INTEGER1,INTEGER2, ENUMERATION	I1-register
REAL (32-bit floating-point)	A1-register
REAL8 (64-bit floating-point)	D1-register
POINTER	I1-register
ARRAY,RECORD,SET (address)	I1-register

If a routine has [expression] ERRETURN, the value of expression is set in the ERRCODE position in the invoker's stack element. Further an ERRETURN exit will set 1 in the STATUS.K bit (the K status bit in the signalling and synchronization status). A normal routine exit will set 0 in the STATUS.K bit.

0.4 INTERFACING WITH PLANC ON THE MC68000



offset from the content
A6-reg (octal) in
bytes

		←— — — A6
0	PREVB	Previous A6-register, reloaded on exit.
4	STP	Points to the free area of stack which immediately follows this stack element.
10	SMAX	Points to the top of free stack (A7)
14	SYST	Run-time system use
20	ERRCODE	ERRCODE (value)
22	stack element	First parameter address if any.
	Free area	Free area of the stack (The stack grows both upwards and downwards in parallel with some information in both parts)
		←— — — A7
0	ENTLINK	System link
4	LINK	Address of return

Top of stack

On return from a routine, the out-value of the routine is returned as follows:

BOOLEAN,BYTE,INTEGER1,INTEGER2, ENUMERATION	D1-register
REAL8 (64-bit floating-point)	A1-register as pointer
POINTER	A1-register
ARRAY,RECORD,SET (address)	A1-register

If a routine has (expression) ERRETURN, the value of expression is set in the ERRCODE position in the invoker's stack element. Further an ERRETURN exit will return according to LINK (in stack upper part) while normal return jumps back to LINK + 2.

0.5 INVOKING PLANC FROM FORTRAN

All PLANC routines called from Fortran must be STANDARD. All arrays transferred from Fortran, should be accessed in PLANC as if they had been declared with a lower index bound of 0.

Example 1 - a simple subroutine call

To call a subroutine with no complex arithmetic actual arguments, the following can be written in Fortran :

```
      EXTERNAL PLSUBR
      INTEGER I
      REAL R
C INVOKE A SUBROUTINE WRITTEN IN PLANC
      CALL PLSUBR(I,R)
```

and the corresponding PLANC code is :

```
      MODULE msubr
      EXPORT plsubr
      INTEGER ARRAY : stack(0:1000)
      ROUTINE STANDARD VOID,VOID(INTEGER,REAL) : plsubr(int,rl)
      INISTACK stack
  % body of routine
      ENDROUTINE
      ENDMODULE
```

Example 2 - a simple function call

To invoke a function which returns a non-complex arithmetic result.

In Fortran :

```
      EXTERNAL PLFUNC
      REAL X,Y,PLFUNC
      DOUBLE PRECISION D
C INVOKE A FUNCTION WRITTEN IN PLANC
      Y=PLFUNC(X,D)
```

and in PLANC :

```
      ROUTINE STANDARD VOID,REAL(REAL,REAL8) : plfunc(rl,db)
      INISTACK stack
  % PLANC REAL8 is the same as Fortran DOUBLE PRECISION
      ... RETURN
      ENDROUTINE
```

Example 3 - use of logical arguments on the ND-100 :

Fortran LOGICAL*2 corresponds to PLANC BOOLEAN. Fortran LOGICAL*4 is the following PLANC data type :

```
         TYPE boolean4 = RECORD
                          BOOLEAN : unused % first word always zero
                          BOOLEAN : value  % contains actual value
                         ENDRECORD
```

LOGICAL*4 cannot be returned from a PLANC STANDARD routine.

In Fortran :

```
         EXTERNAL PLBOOL
         LOGICAL PLBOOL,V
         LOGICAL*4 M4
         V=PLBOOL(V,M4)
```

In PLANC :

```
         ROUTINE STANDARD VOID,BOOLEAN (BOOLEAN,boolean4) :  &
                                                  plbool(m,m4)
         INISTACK stack
         IF m4.value THEN
           m RETURN
         ENDIF
         NOT m RETURN
         ENDROUTINE
```

On the ND-500 :

Fortran LOGICAL*4 corresponds to PLANC BOOLEAN. The Fortran LOGICAL*2 data type has no direct equivalent in PLANC. Fortran LOGICAL*2 can be handled in PLANC in the following way :

In Fortran :

```
         EXTERNAL PLBOOL
         LOGICAL PLBOOL,V
         LOGICAL*2 M2
         V=PLBOOL(V,M2)
```

In PLANC :

```
         ROUTINE STANDARD VOID,BOOLEAN (BOOLEAN,INTEGER2) : &
                                               plbool (m,m2)
         INISTACK stack
     % the 2 integers must be contiguous in memory
         INTEGER2 : int1,int2
         BOOLEAN : bool1=int1
         m2=:int2
         0=:int1
         IF bool1 THEN
           m RETURN
         ENDIF
         NOT m RETURN
         ENDROUTINE
```

Example 4 - character string arguments

Since Fortran passes character strings through a descriptor, PLANC routines must accept these as records. It is often most convenient to recast the Fortran string descriptor as a PLANC bytes pointer. Thus :

On the ND-100 :

```
TYPE ftnstring = RECORD
                  BYTES : ftnchars (0: -1) % ch. data
                 ENDRECORD  % a blank must precede -1

TYPE ftndesc = RECORD PACK
 ftnstring POINTER        : cstring  % address of string
 INTEGER RANGE (0:1B)   : coddbyte % left/right byte start
 INTEGER RANGE (0:17B)    : cunused  % unused
 INTEGER RANGE (0:3777B) : clength  % length of string
               ENDRECORD
```

Then in Fortran :

```
CHARACTER H*20
INTEGER I,J
EXTERNAL HSUB
CALL HSUB( H(I:J) )
```

which can be picked up in PLANC by :

```
    ROUTINE STANDARD VOID,VOID (ftndesc) : hsub(hij)
    BYTES POINTER : bp
      INISTACK stack
      ADDR( hij.cstring.ftnchars                               &
            (hij.coddbyte : hij.clength-1+hij.coddbyte) )=:bp
% bp now contains the address of the Fortran character string
    ENDROUTINE
```

On the ND-500 :

```
         TYPE ftnstring = RECORD
                           BYTES : ftnchars (0: -1) % ch. data
                          ENDRECORD  % a blank must precede -1

         TYPE ftndesc = RECORD
                         INTEGER RANGE (0:7777777777B) : clength
                         ftnstring POINTER             : cstring
                        ENDRECORD
```

Then in Fortran :

```
         CHARACTER H*20
         INTEGER I,J
         EXTERNAL HSUB
         CALL HSUB( H(I:J) )
```

which can be picked up in PLANC by :

```
         ROUTINE STANDARD VOID,VOID (ftndesc) : hsub(hij)
         BYTES POINTER : bp
           INISTACK stack
           ADDR( hij.cstring.ftnchars(0 : hij.clength-1) )=:bp
     % bp now contains the address of the Fortran character string
         ENDROUTINE
```

Example 5 - functions returning a character value

The definition of character data types must be made as in example 4. But in this case there can be no true return value for the function, so the PLANC code must simulate the return.

On the ND-100 :

In Fortran :

```
         CHARACTER H*20,HFUNC*10
         EXTERNAL HFUNC
         H(1:10)=HFUNC(...)
```

In PLANC :

```
         ROUTINE STANDARD VOID,VOID : hfunc
         BYTES POINTER : bp
         ftndesc POINTER : dreg
           INISTACK stack
           $* COPY SD DA; STA dreg   % return value descriptor
           ADDR( dreg.cstring.ftnchars                        &
             (dreg.coddbyte : dreg.clength-1+dreg.coddbyte) )=:bp
           '0123456789'=:IND(bp)     % set 'return value'
         ENDROUTINE
```

On the ND-500 :

In Fortran :

```
         CHARACTER H*20,HFUNC*10
         EXTERNAL HFUNC
         H(1:10)=HFUNC(...)
```

In PLANC :

```
         ROUTINE STANDARD VOID,VOID : hfunc
         BYTES POINTER : bp
         ftndesc POINTER : rreg
           INISTACK stack
           $* R=:B.rreg   % return value descriptor
           ADDR( rreg.cstring.ftnchars(0 : rreg.clength-1) )=:bp
           '0123456789'=:IND(bp)     % set 'return value'
         ENDROUTINE
```

0.6 INVOKING FORTRAN FROM PLANC

All Fortran subprograms invoked from PLANC must be IMPORT'ed as STANDARD routines. Fortran functions have out-values, but no Fortran routines have in-values.

Example 1 - a simple subroutine

Invoke a Fortran subroutine with non-complex arithmetic dummy arguments.

In PLANC :

```
          IMPORT (ROUTINE STANDARD VOID,VOID(REAL,REAL8) : fsubr)
      %
          REAL : r
          REAL8 : d
          ...fsubr(r,d)
```

In Fortran :

```
          SUBROUTINE FSUBR(R,D)
          REAL R
          DOUBLE PRECISION D
      C
          END
```

Example 2 - a simple function

Invoke a Fortran function returning a non-complex arithmetic result.

In PLANC :

```
          IMPORT ( ROUTINE STANDARD VOID,INTEGER(INTEGER4) : ifunc )
      %
          INTEGER : k
          INTEGER4 : kd
            ifunc(kd)=:k
```

In Fortran :

```
          INTEGER FUNCTION IFUNC(KD)
          INTEGER*4 KD
          IFUNC=...
          RETURN
          END
```

Example 3 - use of logical arguments

PLANC BOOLEAN is the same as LOGICAL in Fortran, LOGICAL*2 on the ND-100 and LOGICAL*4 on the ND-500. LOGICAL*4 on the ND-100 or LOGICAL*2 on the ND-500 may be simulated as in example 3 of the previous section.

On the ND-100 :

In PLANC :

```
          IMPORT ( ROUTINE STANDARD VOID,BOOLEAN(boolean4):lfunc )
     %
          boolean4 : m4
            IF lfunc (m4) THEN ...
```

In Fortran :

```
          LOGICAL FUNCTION LFUNC(M4)
          LOGICAL*4 M4
          LFUNC=...
          RETURN
          END
```

On the ND-500 :

In PLANC :

```
          IMPORT ( ROUTINE STANDARD VOID,BOOLEAN(INTEGER2):lfunc )
     % the 2 integers must be contiguous in memory
          INTEGER2 : int1,int2
          BOOLEAN : bool1=int1
     % put a value in the boolean data-element
          ...=:bool1
            IF lfunc (int2) THEN ...
```

In Fortran :

```
          LOGICAL FUNCTION LFUNC(M2)
          LOGICAL*2 M2
          LFUNC=...
          RETURN
          END
```

Example 4 - character string arguments

Fortran handles character strings by means of descriptors, which can be declared in PLANC as in example 4 of the previous section. These descriptors must be created in PLANC before invocation of the Fortran subprogram takes place.

In PLANC :

```
         IMPORT ( ROUTINE STANDARD VOID,VOID(ftndesc) : hsub )
         ftndesc : fd
         BYTES : arg(1:100)       % begins in left byte of word
         INTEGER : i,j
     % now transfer arg(i:j) to Fortran
         ADDR(arg(i)) FORCE ftnstring POINTER=:fd.cstring
     % the following 2 lines are for the ND-100 only
         1-(i MOD 2) =:fd.coddbyte % left/right byte
         0=:fd.cunused
     %
         j-i+1=:fd.clength           % length of string
         hsub(fd)                    % invoke Fortran subprogram
```

In Fortran :

```
         SUBROUTINE HSUB(FD)
         CHARACTER FD*(*)
     C
         END
```

Example 5 - character functions

Characters cannot be returned by Fortran to PLANC as out-values. The memory area for the returned string must be allocated before invoking the function and a special calling sequence is required.

In PLANC :

```
    IMPORT (ROUTINE STANDARD VOID,VOID : hfunc)
    ftndesc : fd
    BYTES : val(0:19)       % value returned here
    ftndesc POINTER : fdp
%
    ADDR(val(0)) FORCE ftnstring POINTER =:fd.cstring
% the following 2 lines are required for the ND-100 only
    0=:fd.coddbyte
    0=:fd.cunused
%
    MAXINDEX(val,1)-MININDEX(val,1)+1=:fd.clength
    ADDR(fd)=:fdp
% on the ND-100 use :
    $* LDA fdp              % return descriptor address
% on the ND-500 use :
    $* R:=fdp               % return descriptor address
%
    hfunc                   % put result in 'val'
```

In Fortran :

```
    CHARACTER *(*) FUNCTION HFUNC
    HFUNC=...
    RETURN
    END
```

O.7 ACCESSING FORTRAN COMMON FROM PLANC

A COMMON block may be defined in a Fortran main program, a subprogram or a BLOCK DATA subprogram. Fortran COMMON blocks may be accessed from a PLANC main program or a subprogram.

A PLANC program can access a Fortran COMMON block by using the COMMON option in the IMPORT statement which enables the appropriate linkage to be established.

For example :

```
         BLOCK DATA
         COMMON /COMBLOC/INT1,INT2,INT3
         DATA INT1/10/,INT2/101/,INT3/58/
         END

         MODULE usecommon
         TYPE comrec = RECORD
                          INTEGER : i1,i2,i3
                        ENDRECORD
         IMPORT (COMMON) comrec:combloc
     %
     % rest of program
     %
         INTEGER : int
     % acces a value in the COMMON block
         combloc.i2=:int
```

This technique may also be used for RT programs, written in PLANC, which are to access RTCOMMON.

0.8 *INVOKING PLANC FROM COBOL*

On both the ND-100 and the ND-500, a COBOL program may call a routine written in PLANC. The PLANC routine must be declared as STANDARD. Parameters are transferred by reference between PLANC and COBOL. The data types which correspond in PLANC and COBOL are as follows :

-PLANC	COBOL
INTEGER2, 16-bits	PIC S9(n) COMPUTATIONAL where 1<=n<=4
INTEGER4, 32-bits	PIC S9(n) COMPUTATIONAL where 5<=n<=10
REAL, 32/48-bits	COMPUTATIONAL-2 (ND-100)
REAL8, 64-bits	COMPUTATIONAL-2 (ND-500)
BYTES (0 as lower bound)	PIC X(n) where n is the number of bytes

COMPUTATIONAL-2 variables may only be used as a parameter in a subroutine call to or from COBOL, or to convert to/from COMPUTATIONAL-3 variables.

Parameters from COBOL must start on a word boundary, on the ND-100.

For example :

In COBOL :

```
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        01  PLANC-INT2      PIC S9(4) COMP        VALUE 123.
        01  PLANC-INT4      PIC S9(6) COMP        VALUE 123456.
        01  CB-REAL         PIC S9(3)V9(6) COMP-3 VALUE -2.71.
        01  PLANC-REAL      COMP-2.
        01  PLANC-BYTES     PIC X(10)         VALUE "A123456789"
        01  PLANC-BYTES-WDS PIC S9(4) COMP        VALUE 5.
    * NUMBER OF CHARACTERS PER WORD IS DIFFERENT ON THE ND-500
        PROCEDURE DIVISION.
        PARA-1.
    * CONVERT THE INTERNAL COBOL FORM TO THE PLANC REAL FORM
        MOVE CB-REAL                  TO PLANC-REAL.
    * INVOKE A PLANC SUBROUTINE
        CALL "PLANCSUB" USING         PLANC-INT2
                                      PLANC-REAL
                                      PLANC-INT4
                                      PLANC-BYTES
                                      PLANC-BYTES-WDS.
```

In PLANC :

```
        ROUTINE STANDARD  VOID,VOID                                  &
         ( INTEGER2, REAL, INTEGER4, BYTES, INTEGER2 ) : PLANCSUB &
         ( int2, rl, int4, string, stringwords )
        INISTACK stack
    % may now access values passed from COBOL and return values
    % to COBOL in the normal manner
        RETURN
        END
```

0.9 INVOKING COBOL FROM PLANC

On both the ND-100 and the ND-500, a PLANC program may call a routine written in COBOL. Parameters are transferred by reference between PLANC and COBOL. The data types which correspond in PLANC and COBOL are as follows :

-PLANC	COBOL
INTEGER2, 16-bits	PIC S9(n) COMPUTATIONAL where 1<=n<=4
INTEGER4, 32-bits	PIC S9(n) COMPUTATIONAL where 5<=n<=10
REAL, 32/48-bits	COMPUTATIONAL-2 (ND-100)
REAL8, 64-bits	COMPUTATIONAL-2 (ND-500)
BYTES (0 as lower bound)	PIC X(n) where n is the number of bytes

COMPUTATIONAL-2 variables may only be used as a parameter in a subroutine call to or from COBOL, or to convert to/from COMPUTATIONAL-3 variables.

For example :

In PLANC :

```
         ROUTINE VOID, VOID (...) : callcobol (...)
         IMPORT (  ROUTINE STANDARD VOID, VOID                 &
                     ( INTEGER2,REAL,INTEGER4,BYTES ) : CBSUB   )
     %
         INTEGER2 : int2
         REAL : rl
         INTEGER4 : int4
         56=:int2
         54.12345=:rl
         123456=:int4
     % invoke a COBOL subroutine
         CBSUB(int2,rl,int4,'string')
```

In COBOL :

```
        IDENTIFICATION DIVISION.
        PROGRAM-ID. CBSUB.
        DATA DIVISION.
        WORKING-STORAGE SECTION.
        01  CB-REAL                  PIC S9(3)V9(6) COMP-3.
        LINKAGE SECTION.
        01  PLANC-INT2               PIC S9(4) COMP.
        01  PLANC-INT4               PIC S9(6) COMP.
        01  PLANC-REAL               COMP-2.
        01  PLANC-STRING             PIC X(6).
        PROCEDURE DIVISION USING     PLANC-INT2
                                     PLANC-REAL
                                     PLANC-INT4
                                     PLANC-STRING.
        PARA-1.
      * CONVERT THE PLANC REAL VALUE TO THE INTERNAL COBOL FORM
        MOVE PLANC-REAL              TO CB-REAL.
```

0.10 INVOKING PLANC FROM BASIC

All PLANC routines called from BASIC must be STANDARD. All arrays transferred from BASIC, should be accessed in PLANC as if they had been declared with a lower index bound of 0.

Example 1 - a simple subroutine call

To call a subroutine, the following can be written in BASIC :

```
        10 EXTERNAL PLSUBR
        20 INTEGER I
        30 REAL R
        40 REM
        50 REM INVOKE A SUBROUTINE WRITTEN IN PLANC
        60 REM
        70 CALL PLSUBR(I,R)
```

and the corresponding PLANC code is :

```
        MODULE msubr
        EXPORT plsubr
        INTEGER ARRAY : stack(0:1000)
        ROUTINE STANDARD VOID,VOID(INTEGER,REAL) : plsubr(int,rl)
        INISTACK stack
    % body of routine
        ENDROUTINE
        ENDMODULE
```

Example 2 - a simple function call

To invoke a function

In BASIC :

```
        10 EXTERNAL PLFUNC
        20 REAL X,Y,PLFUNC,Z
        30 REM
        40 REM INVOKE A FUNCTION WRITTEN IN PLANC
        50 REM
        60 Y=PLFUNC(X,Z)
```

and in PLANC :

```
        ROUTINE STANDARD VOID,REAL( REAL,REAL ) : plfunc(r1,r2)
        INISTACK stack
        r1+r2 RETURN
        ENDROUTINE
```

Example 3 - character string arguments

Since BASIC passes character strings through a descriptor, PLANC routines must accept these as records. It is often most convenient to recast the BASIC string descriptor as a PLANC bytes pointer. Thus,

On the ND-100 :

```
TYPE basicstring = RECORD
                     BYTES : basicchars (0: -1) % ch. data
                    ENDRECORD  % a blank must precede -1

TYPE basicdesc = RECORD PACK
 basicstring POINTER      : cstring % address of string
 INTEGER RANGE (0:1B)     : c1unused % unused
 INTEGER RANGE (0:17B)    : c2unused % unused
 INTEGER RANGE (0:3777B) : clength  % length of string
                 ENDRECORD
```

Then in BASIC :

```
10 EXTERNAL HSUB
20 A$="MY FRIEND"
30 CALL CHSUB(A$)
```

which can be picked up in PLANC by :

```
     ROUTINE STANDARD VOID,VOID (basicdesc) : chsub(hij)
     BYTES POINTER : bp
       INISTACK stack
       ADDR( hij.cstring.basicchars(0:hij.clength-1) )=:bp
% bp now contains the address of the BASIC character string
%
% a string value may be returned as follows
       '123456789'=:IND(bp)      % set 'return value'
     ENDROUTINE
```

0.11 INVOKING BASIC FROM PLANC

All BASIC subprograms invoked from PLANC must be IMPORT'ed as STANDARD routines. BASIC functions have out-values, but no BASIC routines have in-values.

Example 1 - a simple subroutine

Invoke a BASIC subroutine with non-complex arithmetic dummy arguments.

In PLANC :

```
        IMPORT ( ROUTINE STANDARD VOID,VOID(REAL,REAL) : bsubr )
    %
        REAL : r1,r2
        ...bsubr(r1,r2)
```

In BASIC :

```
        10 SUBROUTINE BSUBR(R1,R2)
        20 REAL R1,R2
        30 REM  BODY OF THE SUBROUTINE
        40 END
```

Example 2 - a simple function

Invoke a BASIC function returning a non-complex arithmetic result.

In PLANC :

```
        IMPORT ( ROUTINE STANDARD VOID,INTEGER(INTEGER4) : ifunc)
    %
        INTEGER : k
        INTEGER4 : kd
          ifunc(kd)=:k
```

In BASIC :

```
        10 FUNCTION IFUNC(KD)
        20 INTEGER IFUNC
        30 DOUBLE KD
        40 IFUNC=...
        50 END
```

Example 3 - character string arguments

BASIC handles character strings by means of descriptors, which can be declared in PLANC as in example 3 of the previous section. These descriptors must be created in PLANC before invocation of the BASIC subprogram takes place.

In PLANC :

```
        IMPORT ( ROUTINE STANDARD VOID,VOID(basicdesc) : chsub )
        basicdesc : bd
        BYTES : arg(0:100)        % begins in left byte of word
        INTEGER : i,j
    % now transfer arg(i:j) to BASIC. NB i must be an even value.
        ADDR(arg(i)) FORCE basicstring POINTER=:bd.cstring
    % set up special descriptor constants
        0=:bd.c1unused
        10B=:bd.c2unused
    %
        j-i+1=:bd.clength           % length of string
        chsub(fd)                   % invoke BASIC subprogram
```

In BASIC :

```
        10 SUBROUTINE CHSUB(FD$)
        20 REM  BODY OF SUBROUTINE
        30 END
```

0.12 INVOKING PLANC FROM MAC

A MAC program, running on the ND-100, may invoke a routine written in PLANC. The PLANC routine should be declared as STANDARD. The contents of the B-register and the L-register are described in section 0.2 .

The MAC program must set up the A-register to contain the address of the list of parameter addresses and the T-register to contain the number of parameters.

Example of a MAC program invoking a PLANC routine :

```
  )9BEG
  )9EXT SUBR
  LDA (PLIST          % set up address of the list of
                      % parameter addresses
  SAT N               % set up the number of parameters
  JPL I SUBR          % invoke the routine
% routine will return here
%
% executable code
%

%
% list of parameter addresses
%
  PLIST,PARAM1        % address of first parameter
        PARAM2        % address of second parameter
% to n parameters
  )FILL
  )END
```

Note that there is no Loader check of mixing two bank PLANC routines with MAC routines.

0.13 INVOKING_MAC_FROM_PLANC_ON_THE_ND-100

A PLANC program, running on the ND-100, may invoke a routine written in MAC. The MAC routine should be IMPORT'ed as STANDARD. The contents of the B-register and the L-register are described in section 0.2 .

On entry to the MAC routine, the A-register contains the address of the list of parameter addresses, the T-register contains the number of parameters.

Example of a MAC routine :

```
        )9BEG
        )9ENT SUBR

        SUBR,  SWAP SA DB
               STA SAVB      % saves value in the B-register
               LDA I 0,B     % value of first parameter
    %
               LDA I N-1,B   % value of the n'th parameter
    %
    % executable code
    %
               LDA SAVB
               COPY SA DB
               EXIT          % return to invoker
        SAVB,0
        )9END
```

Note that there is no Loader check of mixing two bank PLANC routines with MAC routines.

APPENDIX E

USING SINTRAN MONITOR CALLS

0.1 SINTRAN MONITOR CALLS

A number of SINTRAN monitor call routines are available to be called from PLANC, provided as part of the PLANC run-time system. The definition of what monitor calls do can be found in the SINTRAN Reference Manual (ND-60.128). This section contains a description of those monitor calls relevant to PLANC programs with the routine name, the data types of the in-value, out-value and parameters, plus any notes which relate to the particular use of such a monitor call from PLANC. The list is in the sequence of the monitor call numbers.

Any monitor calls not listed here, may be called from a PLANC routine if a suitable interface routine is constructed by the user. If this is done, the user must load the interface routine before the PLANC run-time library.

0.2 MONITOR CALLS AVAILABLE ON THE ND-100 AND THE ND-500

MON0 - LEAVE

```
        ROUTINE VOID,VOID : MON0
```

MON1 - INBT

```
        ROUTINE VOID,BYTE ( INTEGER ) : MON1 (dev)

    %  parameter  : dev = logical device number
    %  out-value  : 8 bit character
```

MON2 - OUTBT

```
        ROUTINE BYTE,VOID ( INTEGER ) : MON2 (dev)

    %  in-value   : 8 bit character
    %  parameter  : dev = logical device number
```

MON3 - ECHOM

```
        ROUTINE VOID,VOID                                  &
           ( INTEGER,INTEGER,BOOLEAN ARRAY PACKED )        &
           : MON3 (dev,mode,table)

    % parameters : dev   = logical device number
    %            : mode  = echo strategy
    %            : table = 8 words containing bit map if mode=7
```

MON4 - BRKM

```
        ROUTINE VOID,VOID                                          &
           ( INTEGER,INTEGER,BOOLEAN ARRAY PACKED,INTEGER )        &
           : MON4 (dev,mode,table,max)

    % parameters : dev   = logical device number
    %            : mode  = break strategy
    %            : table = 8 words containing bit map if mode=7
```

MON11 - TIME

```
        ROUTINE VOID,INTEGER4 : MON11

    % out-value  : time in basic units
```

MON12 - SETCM

```
        ROUTINE VOID,VOID ( BYTES ) : MON12 (command)

    % parameter  : command = command string
```

MON13 CIBUF

```
         ROUTINE VOID,INTEGER ( INTEGER ) : MON13 (dev)

    % parameter  : dev = logical device number
    % out-value  : previous value of A-register or error value
                   (ROUTINEERROR exit will be taken)
```

MON14 - COBUF

```
         ROUTINE VOID,VOID ( INTEGER ) : MON14 (dev)

    % parameter  : dev = logical device number
```

MON16 - MGTTY

```
         ROUTINE VOID,INTEGER ( INTEGER ) : MON16 (dev)

    % parameter  : dev = logical device number
    % out-value  : terminal type
```

MON17 - MSTTY

```
         ROUTINE INTEGER,VOID ( INTEGER ) : MON17 (dev)

    % input para : terminal type
    % parameter  : dev = logical device type
```

MON21 - M8INB (ND-100)

```
         TYPE IW = INTEGER WRITE
         ROUTINE VOID,VOID                                     &
            ( INTEGER,IW,IW,IW,IW,IW )                         &
            : MON21(dev,w1,w2,w3,w4,num)

    % parameters : dev = logical device type
    %            : w1  = byte 1 and 2
    %            : w2  = byte 3 and 4
    %            : w3  = byte 5 and 6
    %            : w4  = byte 7 and 8
    %            : num = number of bytes read
```

MON21 - M8INB (ND-500)

```
         ROUTINE VOID,VOID                                     &
            ( INTEGER,INTEGER WRITE,BYTES )                    &
            : MON21(dev,num,inbytes)

    % parameters : dev = logical device type
    %            : num = number of bytes read
    %            : inbytes = bytes input
```

MON22 - M8OUT

```
        ROUTINE VOID,VOID                                     &
           ( INTEGER,INTEGER,INTEGER,INTEGER,INTEGER )        &
           : MON22(dev,w1,w2,w3,w4)

    % parameters : dev = logical device type
    %            : w1  = BYTES(0:1)
    %            : w2  = BYTES(2:3)
    %            : w3  = BYTES(4:5)
    %            : w4  = BYTES(6:7)
```

MON24 - B8OUT

```
        ROUTINE INTEGER,VOID ( INTEGER ARRAY ) : MON24 (ival)

    % in-value    : logical device number
    % parameter   : ival = 8 bytes to be written
```

MON30 - GETRT

```
        ROUTINE VOID,INTEGER : MON30

    % out-value   : address of RT-description
```

MON32 - MSG

```
        ROUTINE VOID,VOID ( BYTES ) : MON32 (MSG)

    % parameter   : msg = bytes to be written
```

MON41 - ROBJE

```
        ROUTINE VOID,VOID ( INTEGER,INTEGER ARRAY )           &
           : MON41 (dev,obj)

    % parameters : dev = logical device number
    %            : obj = the file object entry
```

MON43 - CLOSE

```
        ROUTINE VOID,VOID ( INTEGER ) : MON43 (dev)

    % parameter   : dev = logical device number
```

MON44 - RUSER

```
        ROUTINE VOID,VOID ( BYTES,INTEGER ARRAY )             &
           : MON44 (user,usentry)

    % parameters  : user    = user name
    %             : usentry = user entry
```

MON45 - DBRK (ND-100 only)

```
          ROUTINE VOID,VOID ( INTEGER ARRAY,INTEGER )        &
             : MON45 (rblock,bhand)

     % parameters  : rblock = register block
     %             : bhand  = break-point execution routine address
```

MON47 - SBRK (ND-100 only)

```
          ROUTINE VOID,VOID ( INTEGER ARRAY ) : MON47 (rblock)

     % parameter  : rblock = register block
```

MON50 - OPEN

```
          ROUTINE VOID,INTEGER ( BYTES,BYTES,INTEGER )       &
             : MON50 (file,default,code)

     % parameters : file    = file name
     %            : default = default file type
     %            : code    = access code
     % out-value  : logical device number
```

MON54 - MDLFI

```
          ROUTINE VOID,VOID ( BYTES ) : MON54 (mem)

     % parameter  : mem = file name to be deleted
```

MON62 - RMAX

```
          ROUTINE VOID,INTEGER4 ( INTEGER ) : MON62 (dev)

     % parameter  : dev = logical device number
     % out-value  : number of bytes
```

MON63 - B4INW

```
          ROUTINE VOID,INTEGER ARRAY ( INTEGER ) : MON63 (ldn)

     % parameter  : ldn = logical device number
     % out-value  : BYTES (0:7) READ
```

MON64 - ERMSG

```
          ROUTINE INTEGER,VOID : MON64

     % in-value   : error number to be printed
```

MON65 - QERMS

```
          ROUTINE INTEGER,VOID : MON65

     % in-value   : error number to be printed
```

MON66 - ISIZE

```
        ROUTINE VOID,INTEGER ( INTEGER ) : MON66 (dev)

  % parameter  : dev = logical device number
  % out-value  : number of bytes in input buffer
```

MON70 - COMND

```
        ROUTINE VOID,VOID ( BYTES ) : MON70 (command)

  % parameter  : command = command to be executed
```

MON71 - DESCF

```
        ROUTINE VOID,VOID ( INTEGER ) : MON71 (dev)

  % parameter  : dev = logical device number
```

MON72 - EESCF

```
        ROUTINE VOID,VOID ( INTEGER ) : MON72 (dev)

  % parameter  : dev = logical device number
```

MON73 - SMAX

```
        ROUTINE INTEGER4,VOID ( INTEGER ) : MON73 (dev)

  % in-value   : maximum byte pointer
  % parameter  : dev = logical device number
```

MON74 - SETBT

```
        ROUTINE INTEGER4,VOID ( INTEGER ) : MON74 (dev)

  % in-value   : byte pointer
  % parameter  : dev = logical device number
```

MON75 - REABT

```
        ROUTINE VOID,INTEGER4 ( INTEGER ) : MON75 (dev)

  % parameter  : dev = logical device number
  % out-value  : byte pointer
```

MON76 - SETBS

```
        ROUTINE INTEGER,VOID ( INTEGER ) : MON76 (dev)

  % in-value   : block size in words
  % parameter  : dev = logical device number
```

MON104 - HOLD

```
         ROUTINE VOID,VOID ( INTEGER,INTEGER ) : MN104 (ntu,tu)

     % parameters  : ntu = number of time units in wait state
     %             : tu  = time mode
```

MON113 - CLOCK

```
         ROUTINE VOID,VOID ( INTEGER ARRAY WRITE ) : MN113 (cal)

     % parameter  : cal = time return array
```

MON114 - TUSED

```
         ROUTINE VOID,INTEGER4 : MN114

     % out-value   : cpu time used
```

MON117 - RFILE

```
         ROUTINE VOID,VOID                                      &
            ( INTEGER,INTEGER,INTEGER ARRAY,INTEGER,INTEGER )   &
            : MN117 (dev,zero,dadr,bl,words)

     % parameters : dev    = logical device number
     %            : zero   = return parameter
     %            : dadr   = destination array
     %            : bl     = number of file block where data starts
     %            : words  = number of words to be transferred
```

MON120 - WFILE

```
         ROUTINE VOID,VOID
            ( INTEGER,INTEGER,INTEGER ARRAY,INTEGER,INTEGER )
            : MN120 (dev,zero,dadr,bl,words)

     % parameters : dev    = logical device number
     %            : zero   = return parameter
     %            : dadr   = destination array
     %            : bl     = number of file block where data starts
     %            : words  = number of words in the extent
```

MON122 - RESRV

```
ROUTINE VOID,INTEGER ( INTEGER,INTEGER,INTEGER )  &
   : MN122 (dev,iof,iret)

% parameters : dev  = logical device number
%            : iof  = input - or output part
%            : iret = return status
% out-value  : return status depending on iret
```

MON123 - RELES

```
ROUTINE VOID,VOID ( INTEGER,INTEGER ) : MN123 (dev,iof)

% parameters : dev = logical device number
%            : iof = input- or output part
```

MON132 - MCALL (ND-100 only)

```
ROUTINE VOID,VOID ( INTEGER,INTEGER ) : MN132 (subr,newsg)

% parameters : subr = subroutine address
%              newsg = new segment to be loaded
```

MON141 - IOSET (ND-100 only)

```
ROUTINE VOID,INTEGER                                  &
   ( INTEGER INTEGER,INTEGER,INTEGER )                &
   : MN141 (dev,iof,iprog,ccode)

% parameters : dev   = logical device number
%            : iof   = input - or output part
%            : iprog = RT description
%            : ccode = control code
% out-value  : status
```

MON143 - RSIO

```
ROUTINE VOID,VOID                                     &
   ( INTEGER WRITE,INTEGER WRITE,INTEGER WRITE,       &
     INTEGER WRITE )                                  &
   : MN143 (mode,inpt,outpt,usn)

% parameters : mode  = executing mode
%            : inpt  = file number of command input file
%            : outpt = file number of command output file
%            : usn   = user number the program is running under
```

MON144 - MAGTP

```
     ROUTINE VOID,INTEGER                                        &
        ( INTEGER,INTEGER ARRAY,INTEGER,INTEGER,                 &
          INTEGER WRITE )                                        &
        : MN144 (fc,madr,dev,maxw,readw)

% parameters : fc    = function to be performed
%            : madr  = memory area to be used
%            : dev   = logical device number
%            : maxw  = device dependent
%            : readw = device dependent
% out-value  : status= read status value for function 20B and 24B
                       otherwise zero
```

MON161 - INSTR

```
     ROUTINE VOID,INTEGER                          &
        ( INTEGER,BYTES,INTEGER,INTEGER )          &
        : MN161 (dev,dar,dno,dte)

% parameters  : dev = logical device number
%             : dar = input data buffer
%             : dno = maximum nuber of characters to be read
%             : dte = terminal character
% out-value   : status return
```

MON162 - OUTST

```
     ROUTINE VOID,INTEGER                                  &
        ( INTEGER,BYTES,INTEGER ) : MN162 (dev,dar,dno)

% parameters : dev = logical device number
%            : dar = array of data destination
%            : dno = number of characters to be written
% out-value  : status return
```

MON167 - REENT (ND-100 only)

```
     ROUTINE INTEGER,VOID : MN167

% in-value   : segment number
```

MON263 - GDEVT

```
     ROUTINE VOID,VOID                                     &
        ( INTEGER,INTEGER,INTEGER WRITE,INTEGER4 WRITE ) &
        : MN263 (dev,ioflag,devtype,attr)

% parameters : dev     = logical device number
%   input    : ioflag  = input/output flag
%   output   : devtype = device type
%            : attr    = device attributes
```

MON310 - T8INB (ND-100 only)

```
        ROUTINE VOID,VOID                                      &
           ( INTEGER,BYTES WRITE,INTEGER WRITE )               &
           : MN310 (dev,string,num)

    % parameters :
    %   input    : dev    = logical device number
    %   output   : string = character string read
    %            : num    = number of bytes read
```

MON312 - MOINF (ND-100 only)

```
        ROUTINE VOID,BOOLEAN ( INTEGER ) : MN312 (num)

    % parameters : num = monitor call number
    % out-value  : monitor call present or not
```

MON412 - FSCNT (ND-500 only)

```
        ROUTINE VOID,VOID                                      &
           ( INTEGER,INTEGER,INTEGER,INTEGER WRITE )           &
           : MN412 (fno,lseg,ctype,segno)

    % parameters : fno   = file number
    %            : lseg  = logical segment number
    %            : ctype = connect type
    %            : segno = segment number
```

MON413 - FSDCNT (ND-500 only)

```
        ROUTINE VOID,VOID                                      &
           ( INTEGER,INTEGER )                                 &
           : MN413 (fno,segno)

    % parameters : fno   = file number
    %            : segno = segment number
```

APPENDIX E

BNF SYNTAX DESCRIPTION OF PLANC

This appendix contains a Backus-Naur Form (BNF) syntax description of the PLANC language.

Notation used in this appendix

[and], ..., (and)
Square brackets, ellipsis and parentheses are used in the same way as described in Notation in This Manual.

|, ::=, <symbol>
are used in the usual way defined for Backus-Naur Form.

BNF Syntax

```
<identifier>::= <letter> |@ |
                <letter>[<identifier char>]...<letter> |
                <letter>[<identifier char>]...<digit>

<identifier char>::= <letter> |<digit> |_

<number>::= <decimal number> |<octal number> |
            <floating point number>

<decimal number>::= [-]<unsigned decimal number>

<unsigned decimal number>::= <decimal digit>...

<decimal digit>::= <octal digit> |8 |9

<octal digit>::= 0 |1 |2 |3 |4 |5 |6 |7

<floating point number>::=
      <decimal number>.<unsigned decimal number>
                          [E<decimal number>]

<character literal>::= # <character>

<string>::= '<character>...'
```

```
<simple type specification statement>::=
      TYPE<type identifier> = <simple type>

<type identifier>::= <identifier>

<simple type>::= <standard data type> |<enumeration type> |
                 <pointer type> |<modified type>

<standard data type>::= INTEGER |REAL |BOOLEAN |LABEL |VOID

<enumeration type>::= ENUMERATION(<identifier>
                                  [,<identifier>]...)

<pointer type>::= <qualification> POINTER

<qualification>::= <type identifier> |<simple type> |
                   <array type> |<record type> |<set type>

<modified type>::= <range modified type> |
                   <precision modified type> |
                   <read modified type>

<range modified type>::= INTEGER RANGE(<lower>:<upper>)

<lower>::= <constant expression>

<upper>::= <constant expression>

<precision modified type>::= REAL PRECISION (<precision>)

<precision>::= <constant expression>

<read modified type>::= <simple data type><access mode>

<access mode>::= READ |WRITE

<constant specification>::= <identifier> |<expression>

<simple type declaration statement>::=
      <simple type specifier>:<identifier clause>
      [,<identifier clause>]...

<simple type specifier>::= <simple type> |<type identifier> |
                           TYPEOF(<identifier>)
```

```
<identification clause>::= <construction clause> |
                           <equivalence clause>  |
                           <postponement clause>

<construction clause>::= <identifier>[:=<expression>]

<equivalence clause>::= <identifier>=<identifier>

<postponement clause>::= <identifier>?

<composite type specification statement>::=
      <array                type specification statement> |
      <record               type specification statement> |
      <variant part record  type specification statement> |
      <enumeration set      type specification statement> |
      <routine              type specification statement>

<array type specification statement>::=
      TYPE<type identifier> = <array type>

<mode specification>::= <storage mode> |<access mode>

<storage mode>::= PACKED

<access mode>::= READ |WRITE

<array type declaration statement>::=
      <array type specifier>:<array identification clause>
      [,<array identification clause>]...

<array type specifier>::= <array type> |<type identifier>

<array identification clause>::=
      <dimension clause>[<initialization part>] |
      <array initialization clause>

<dimension clause>::=
      <identifier>(<index set>[,<index set>...)

<array initialization clause>::= <identifier>
                                    <initialization part>

<initialization part>::= :=<initial array values>

<initial array values>::= <initial array values> |
                          [,<initial array values>]... |
                          (<expression>[,<expression>]...)

<index set>::= <expression>:<expression>

NEW<array type specifier> (<sub-array index set>
                             [,<sub-array index set>]...)

<sub-array specification>::=
      <identifier>(<sub-array index set>
        [,<sub-array index set>]...)

<sub-array index set>::= <expression>:<expression> |
                         <expression>
```

```
<record type specification statement>::=
   TYPE<type identifier> = RECORD [<mode specification>]...
                             [<data declaration statement>]...
                           ENDRECORD

<data declaration statement>::=
      <simple type declaration statement> |
      <array  type declaration statement> |
      <record type declaration statement> |
      <set    type declaration statement>

<mode specification>::= <storage mode> |<access mode>

<storage mode>::= PACKED

<access mode> ::= READ |WRITE

<variant part record specification>::=
   TYPE<record type identifier>=
            <base record>RECORD[<mode specification>]...
                           [<declaration statement>]...
                         ENDRECORD

<base record>::= <type identifier>

<record type declaration statement>::=
      <record type specifier>:<record identification clause>
           [,<record identification clause>]...

<record type specifier>::= <type identifier>

<record identification clause>::=
                   <identifier>[:=<initial record values>]

<initial record values>::=
                   (<initial value>[, = <initial value>]...)

<initial value>::= <expression> |
                   <initial array values> |
                   <initial record values>
```

```
<set type specification statement>::=
      TYPE<type specifier> = <set type>

<set type>::= <base type> SET

<base type>::= <type identifier> |<range modified type> |
               <enumeration type>

<set declaration statement>::= <set declaration>
                                 [,<set declaration>...]

<set declaration>::=
      <set type specifier>:<identifier>[:=(member list>)]

<set type specifier>::= <type specifier> |<set type>

<member list>::= <member list element>
                    [,<member list element>]... |
                 <identifier>

<member list element>::= <expression> |
                         <expression>:<expression>
```

```
<action>::= [<label>:]<expression> |
            [<label>:]<sequencing control statement> |
            [<label>:]<exception handler>

<label>::= <identifier>

<action sequence>::= <action>[,<action>...]

<expression>::= <value expr> |<void expression>

<value expr>::= <data-element> |
                [<value expr>]<operator><expression> |
                <value expr>[<assignment op> |
                <store-into function call> |
                (<value expr>) |<function call>

<void expr>::= <store-into subroutine call> |
               <subroutine call> |
               <value expr>[<assignment op>]
                              <store-into subroutine call> |
               <value expr>[<assignment op>]<subroutine call>

<constant expression>::= <constant> |
                         <constant expression><operator>
                                                   <constant>

<constant>::= <constant identifier> |<literal>

<data-element>::= <literal> |<identifier>[,<identifier>]...
                  <identifier>(<index>[,<index>]...)

<index>::= <value expr>

<operator>::= + |* |/ |ABS |MOD |AND |OR |XOR |NOT |SHIFT |
              = |>< |>= | <= |> |< |IN |<assignment op>

<assignment op>::= =: |:=:
```

```
<sequencing statements>::= <go statement> |
      <if statement> |<case statement> |
      <for statement> |<do statement> |
      <while statement> |<assert statement> |
      <return statement> |<do-while statement> |
      <for-while statement>

<go statement>::= GO<label>

<if statement>::=
      IF<expression>THEN<action sequence>
         [ELSIF<expression>THEN<action sequence>]
         [ELSE<action sequence>]
      ENDIF

<condition>::= <expression>

<case statement>::=
      CASE<expression>
         INCASE<member list>
               <action sequence>
         [INCASE<member list>
                <action sequence>]...
         [ELSE<action sequence>]
      ENDCASE

<for statement>::=
      FOR <identifier> IN <set> DO
          <action sequence>
      [EXITFOR <action sequence>]
      ENDFOR

<do statement>::= DO <action sequence> ENDDO

<while statement>::= WHILE <expression>

<do-while statement>::=
      DO
        [<action sequence>]
        WHILE <expression>
        [<action sequence>]
        [EXITWHILE <action sequence>]
      ENDDO

<for-while statement>::=
      FOR <identifier> IN <set> DO
        [<action sequence>]
        WHILE <expression>
        [<action sequence>]
        [EXITFOR <action sequence>]
        [EXITWHILE <action sequence>]
      ENDFOR
```

```
<assert-statement>::= ASSERT <expression>

<return statement>::= RETURN |<expression>RETURN |
                      <expression>ERRETURN

<exception handler>::= ON <exception>[<exception>]...DO
                           <action sequence>
                       ENDON

<exception>::= ROUTINEERROR |OVERFLOW |ASSERTFALSE |
               RANGEERROR |POINTERERROR |STACKERROR |
<routine type specification statement>::=
      TYPE <type identifier> = <routine type>

<routine type>::= ROUTINE [INLINE] [STANDARD] [REFERENCE]
                                               [SPECIAL]
                   <type in>,<type out>[(<parameter type>
                                       [<parameter type>]...)]

<type in> ::= <type identifier>

<type out>::= <type identifier>

<parameter type>::= <type identifier>[<access>]

<access>  ::= READ |WRITE

<routine declaration>::=
      <routine heading> <routine body> ENDROUTINE |
      <postponed routine declaration>

<routine heading>::=
      <routine type specifier>:<routine name>
      [(<formal par>[,<formal par>]...)]

<routine type specifier>::= <type identifier> |<routine type>

<routine name>::= <identifier>

<formal par>::= <identifier>

<postponed routine declaration>::=
                   <routine type specifier>:<routine name>?
                                        [,<routine name>?...]

<routine body>::= [<local declaration>]...<action sequence>

<local declaration>::= <declaration statement> |
                       <routine declaration> |
                       <type specification statement>

<routine call>::= <routine name>[<parameter list>]

<parameter list>::= <identifier> |
                    <expression>[,<expression>]...)

<data declaration statement>::=
      <simple type declaration statement>,|
      <array type declaration statement> |,
      <record type declaration statement> |
      <set type declaration statement>

<subroutine call>           ::= <routine call>

<function call)             ::= <routine call>

<store-into subroutine call>::= <routine call>

<store-into function   call>::= <routine call>
```

```
<main program>::=
      <main program heading> <main program body> ENDROUTINE

<main program heading>::= PROGRAM : <identifier>

<main program body>::=
      [<local declaration>]...<action sequence>

<basic module>::=
      <module header> <basic module body> ENDMODULE

<module header>::=
      MODULE <identifier>[<header statement>]

<header statement>::= <import statement> |
                      <export statement> |
                      <type specification statement> |
                      <constant statement>

<import statement>::=
   IMPORT [SYSTEM] [COMMON] <import unit> [,<import unit>]...

<export statement>::=
   EXPORT [SYSTEM] [COMMON] <identifier> [,<identifier>]...

<type specifier statement>::=
                     <simple type specification statement> |
                     <composite type specification statement>

<basic module body>::= [<declaration unit>]...

<declaration unit>::= <data declaration statement> |
                      <main program> |
                      <routine declaration>

<compound module>::=
      <module header> <compound module body> ENDMODULE

<compound module body>::= <module>...

<module>::= <compound module> |<basic module>
```

APPENDIX G

PLANC IMPLEMENTATION RESTRICTIONS

This appendix describes various restrictions which may cause users difficulties. Some may appear in the text of the manual, but apply to more than one part of it, so they are listed here to make it easier to find them.

1) A statement containing either a MACRO call, an INLINE routine call or a $INCLUDE command, may be terminated by a semicolon, no other statements may follow the semicolon.

2) The IND standard routine cannot have as a parameter a pointer which qualifies a routine with an in-value.

3) If the ADDR standard routine has a parameter which is a routine data-element, this parameter must not be enclosed in parentheses.

4) If the ADDR standard routine has as a parameter, a routine with an outvalue, the outcome of the ADDR routine invocation will be the address of the routine, not the out-value of the routine.

5) Within a routine, the MININDEX, MAXINDEX and IN standard routines cannot have as an actual parameter, any of the routine's formal parameters, if the routine has been declared with the STANDARD modifier. Note that the compiler does not detect this condition or give any error message.

6) The ON OVERFLOW statement does not detect overflow conditions for unsigned integer data-elements.

7) It is illegal to EXPORT a family of routines, with the routine name identifier the same as the name of a PLANC predefined standard routine or operator, see section 8.4 for the use of a family of routines.

8) The following TYPE declaration is illegal, but the compiler does not give any error message:

```
TYPE A=RECORD
           ...
       ENDRECORD
TYPE B=A          % illegal TYPE declaration.
```

9) If a family of routines is declared, it is not adequate to have formal parameters with an identical data type and different access modifiers, the formal parameters must have distinct data types. For example :

```
ROUTINE VOID,VOID(INTEGER      ): RUT?
ROUTINE VOID,VOID(INTEGER WRITE): RUT?
```

The compiler can not distinguish between the two declarations and will give a compile error message.

11) ON ROUTINEERROR does not work correctly in routines declared as INLINE.

12) A routine name declared in predeclaration, must not appear later, as the identifier, in a PROGRAM statement. The compiler does not detect this or give an error message.

13) The $SEPARATE-DATA and $DEBUG-MODE commands must be used outside the outermost module level.

14) If assembler code refers to a routine, then if the routine referred to is not on the same scope level, the reference will not be compiled correctly. In particular, beware of reference to a nested routine.

15) The Break function of the Symbolic Debugger must be used with care. If a Break upon routine entry is specified, then the in-value which will be used within the routine will not be displayed correctly by the Debugger, although as execution continues the correct value will be used in the routine. The same result may be achieved with a Break on the first executable statement of the routine and then the in-value may be displayed correctly by the Debugger.

    A similar difficulty occurs in specifying a Break-return within a routine, where the out-value of the routine may not be displayed by the Debugger, since it has not yet been stored. The out-value can only be correctly displayed by the Debugger after execution of the statement which invokes the routine has been completed.

    If any of the conditional execution constructs such as ELSE, INCASE, EXITFOR and EXITWHILE are followed, on the same line, by executable source code, the Debugger Break function will only stop at the code which ends just prior to the conditional construct, not the code following it.

16) If a data-element has been declared with WRITE access only, a statement which tries to fetch a value from such a data-element will not generate an error message during compilation but the results are unpredictable.

17) It is legitimate in an invocation of a user written routine, which is declared with an out-value, not to store the out-value. The compiler will not give any warning or error messages.

18) If a POINTER for a data type, which has not yet been defined, is declared then space will be allocated as if the POINTER data-element is for any of the simple data types, ie. usually one word.

    For example :

```
% n.b. the TYPE norec has not yet been defined
                 norec pointer: bp   % allocates one word only
```

19) If $ENDIF is used as a parameter in a macro call, then it must be terminated by at least one space.

20) On the ND-100, if a routine is declared with a formal parameter which is REAL8 and with WRITE or READ WRITE access modified, and a routine invocation contains a REAL4 actual parameter, the compiler automatically carries out a conversion. However a value which should be stored into the actual parameter will not be correctly stored.

21) If a component of a RECORD PACKED or an ARRAY PACKED data-element is a different size from an addressable element, or not aligned with an addressable element of the same size, then use of the ADDR standard routine to write values into the component data-element may overwrite adjacent memory areas.

Beware that parameters of routines declared as STANDARD or REFERENCE transfer values by passing addresses, ie. implicitly using the ADDR standard routine, so the above difficulties may arise.

22) If ARRAY-INDEX-CHECK is switched on, and a subarray is used with bounds outside those declared for the original array, the compiler does not give any warning. Further, during execution the checking of array element accesses will be carried out incorrectly after reference to such a subarray, which has bounds outside those of the original array.

23) There are a number of difficulties in invoking inner nested routines :

i) Inner routines, ie. other than the outermost level, may not invoke themselves recursively.

ii) An inner level routine which is predeclared, or invoked by the IND standard routine, will not be executed correctly.

iii) If an invocation of an inner level routine is to have as an actual parameter another inner level routine, then the actual parameter will be transferred correctly only if it is the first parameter in the parameter list.

iv) Inner routines which are declared as STANDARD or REFERENCE, will not be executed correctly.

24) ADDR(ADDR(an ARRAY data-element)) will not work correctly. The correct result may be achieved by using two statements with an explicitly declared ARRAY POINTER data-element.

25) The standard routine MARKSTACK will be removed in a future version of the compiler, so users are advised to avoid its use.

26) The keyword PACK will be removed in a future version of the compiler, so users are advised to avoid its use.

27) Names used in PLANC programs in IMPORT/EXPORT statements, will be truncated to the first ten characters normally. However, note that NRL on the ND-100 uses only the first seven characters, so names must be distinct from each other, up to the seventh character to avoid problems.

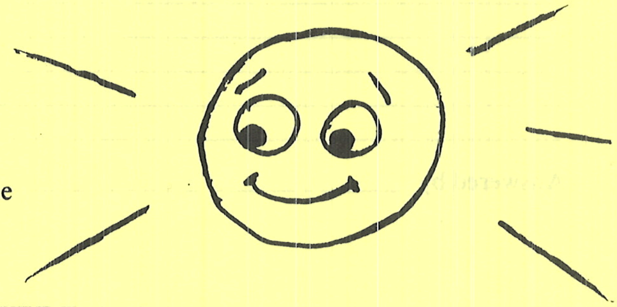
Index

* * * * * * * * * * SEND US YOUR COMMENTS!!! * * * * * * * * * *

Are you frustrated because of unclear information in this manual? Do you have trouble finding things? Why don't you join the Reader's Club and send us a note? You will receive a membership card - and an answer to your comments.

Please let us know if you
- * find errors
- * cannot understand information
- * cannot find information
- * find needless information

Do you think we could improve the manual by rearranging the contents? You could also tell us if you like the manual!!



* * * * * * * * * HELP YOURSELF BY HELPING US!! * * * * * * * * *

Manual name: SINTRAN III Reference Manual

Manual number: ND-60.128.04

What problems do you have? (use extra pages if needed)

Do you have suggestions for improving this manual?

Your name:

Date:

Company:

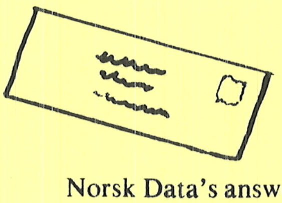
Position:

Address:

What are you using this manual for?

Send to:

Norsk Data A.S.
Documentation Department
P.O. Box 4, Lindeberg Gård
Oslo 10, Norway



Norsk Data's answer will be found on reverse side

Answer from Norsk Data

Answered by   Date

I   I
I   I
I   I

Norsk Data A.S.
Documentation Department
P.O. Box 4, Lindeberg Gård
Oslo 10, Norway

Systems that put people first

NORSK DATA A.S JERIKOVN. 20 P.O. BOX 4 LINDEBERG GÅRD OSLO 10 NORWAY
TEL.: 02 - 30 90 30 - TELEX: 18661