CORAL 66
LANGUAGE REFERENCE
MANUAL

MICRO FOCUS

CORAL 66

LANGUAGE REFERENCE

MANUAL

Version 3

Micro Focus Ltd.

Issue 2
March 1982

This language definition reproduces material from the British Standard BS5905 which is hereby acknowledged as a source.

Micro Focus Ltd.
58, Acacia Road,
St. Johns Wood,
London NW8 6AG
Telephones:
01 722 8843/4/5/6/7
Telex:
28536 MICROF G

CORAL 66 LANGUAGE REFERENCE MANUAL

AMENDMENT RECORD

AMENDMENT NUMBER	DATED	INSERTED BY	SIGNATURE	DATE

PREFACE

This manual defines the programming language CORAL 66 as implemented in Version 3 of the CORAL compilers known as RCC80 and RCC86. Now that British Standard BS5905 has become the main source for CORAL implementation its structure and style have been adopted.

Compared to previous versions of the RCC compilers Version 3 incorporates 'TABLE' and 'OVERLAY', and allows additional forms of Real numbers in line with the British Standard. Compiler error reporting has been modified slightly - in particular all error messages now have identifying numbers and are listed in appendices to the operating manuals.

Micro Focus has now assumed direct responsibility for production of all RCC CORAL manuals and believes that this will offer users a better service than has been possible in the past.

AUDIENCE

This manual is intended as a description of CORAL 66 for reference and assumes familiarity with use of the language.

MANUAL ORGANIZATION

The manual contains the following Chapters and Appendices:

"Chapter 1. Introduction", which introduces the language and describes its structure and design.

"Chapter 2. Scoping", which describes the naming of variables, and the scope of names used.

"Chapter 3. Data Referencing", which describes the manner in which data can be referenced and the grouping of data.

"Chapter 4. Place Referencing", which describes the use of labels and switches in a program.

"Chapter 5. Expressions", which describes the types of expression available.

"Chapter 6. Statements", which describes the types of statement available.

"Chapter 7. Procedures", which describes the use of procedures.

"Chapter 8. Communicators", which describes the manner in which communication between programs can be specified.

"Chapter 9. Names and Constants", which describes the specification of names and constant values.

"Chapters 10. Text in a Program", which describes the documentation of a program, macro processing and source text inclusion facilities.

"Appendix A. Syntax Rules", which has a cross reference to the main text for each syntax rule.

"Appendix B. Procedure Parameters", which is a summary of the correspondence between formal and actual parameters.

"Appendix C. Language Symbols", which lists the symbols available.

"Appendix D. Character Set", which lists available characters.

"Appendix E. CORAL 66 Constraints", which duplicates information on constraints in CORAL 66 described in the CORAL 66 Operating Guide.

"Appendix F. Implementation Specific Features", which summarises those features of this implementation of CORAL 66 that are not specified in BS 5905.

RELATION TO THE BRITISH STANDARD SPECIFICATION

Users may wish to compare this implementation with standard CORAL 66 as specified in British Standard BS 5905.

The structure of this manual follows the structure of BS 5905 and includes material from BS 5905. Features additional to the standard are indicated by grey shading. Where the standard is less specific than this description of CORAL, or where one of a number of alternatives has been chosen from the standard, there is a single vertical line in the left hand margin.

A summary of implementation specific features is given in Appendix G.

NOTATION IN THIS MANUAL

Throughout this manual the following notation is used to describe the format of data input or output:

1. All words printed in small letters are generic terms representing names which will be devised by the programmer.

2. When material is enclosed in square brackets [ ], it is an indication that the material is an option which may be included or omitted as required.

3. The first reference to a new term in text is underlined for emphasis. The term is then used without redefinition in the remainder of the manual. These first references are included in the Index to the manual.

Headings are presented in this manual in the following order of importance:

CHAPTER n
TITLE — Chapter Heading

ORDER ONE HEADING
ORDER TWO HEADING
Order Three Heading
Order Four Heading — Text 3 lines down

Order Five Heading: Text on same line

Numbers one (1) to nine (9) are written in text as letters e.g. one.
Numbers ten (10) upwards are written in text as numbers e.g. 12.

See RELATION TO THE BRITISH STANDARD SPECIFICATION in this Preface for the use of left hand margin bars and grey shading in this manual.

RELATED PUBLICATIONS

For details of operation of the CORAL 66 software refer to the appropriate version document:

RCC80 or RCC86 CORAL 66 Operating Guide for use with your Operating System

For details of Operating System, Messages, and File Structures refer to the appropriate Operating System User manuals.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

CHAPTER 2

SCOPING

CHAPTER 3

DATA REFERENCING

CHAPTER 4

PLACE REFERENCING

CHAPTER 5

EXPRESSIONS

CHAPTER 6

STATEMENTS

CHAPTER 7

PROCEDURES

CHAPTER 8

COMMUNICATIONS

CHAPTER 9

NAMES AND CONSTANTS

CHAPTER 10

TEXT IN A PROGRAM

ILLUSTRATIONS

TABLES

CHAPTER 1

INTRODUCTION

CORAL 66

Coral 66 is a general purpose programming language based on Algol 60. Originally designed by the Royal Radar Establishment, Malvern in 1966 it was formally defined in "Official Definition of Coral 66" published by HMSO and more recently in a British Standard, BS 5905, herein referred to as "The British Standard".

The language is intended for use in basic software, such as compilers and operating systems, and for real time applications of small computers where typically execution time and storage overheads are critical and input/output requires specialized code.

CORAL 66 AND MICROPROCESSORS

There are some aspects of microprocessor architecture that were not envisaged when CORAL 66 was designed. Some compromises have been made in balancing a desire to provide a full and standard implementation against the need to exploit the microprocessor efficiently.

Some particular features of this kind are:-

1. CORAL assumes that data of type INTEGER can be used to contain machine addresses whereas the addressing range of a processor such as the 8086 exceeds the capacity of its natural integers.

2. CORAL defines that consecutive INTEGER variables occupy storage addresses differing by one, which is not true of most modern byte-addressed processors.

3. CORAL preset data must be statically initialized. When programs are to be stored in read-only memory such data must be stored with the instructions, and it will not be possible to change the value of preset data during the course of the program.

4. CORAL does not define a byte or character data type for efficient storage of small integers and graphic characters.

THE CORAL 66 PROGRAM

A distinction is made between symbols and characters. Characters, standing only for themselves, may be used in strings or as literal constants. Apart from such occurrences, a program shall be regarded as a sequence of symbols, each visibly representable by a unique character or combination of characters. The symbols of the language are defined in the British Standard but the characters are not.

NOTATION FOR LANGUAGE SYMBOLS

Two notations are used to write CORAL 66 text. One notation uses single quotes to delimit words which are to be treated as single symbols, while the other uses upper and lower case instead of quotes.

Using Single Quotes

In this notation:-

* Words surrounded by single quotes are treated as single symbols - note that each such symbol must have its own pair of quotation marks even when adjacent to others.

* Lower case letters may be mixed with upper case in such symbols and in identifiers. If so used a lower case letter is treated as equal to the corresponding upper case letter so that the identifiers

```
eXAmple
Example
```

  are not distinct names.

* Layout characters may be embedded in identifiers and in language symbols e.g.

```
'BEGIN'
' GO TO'
```

Using Upper and Lower Case

In this notation:-

* Words in upper case are treated as single symbols. When adjacent to each other such symbols at least one layout character must separate them.

* Identifiers must use only lower case - any change of case will terminate the current item.

* Layout characters may be embedded in identifiers but not in language symbols - for example it is essential to write

    GOTO

    rather than

    GO TO

* The language symbol, END, must be enclosed in single quotes where it occurs to indicate the end of a CODE statement.

LAYOUT CHARACTERS

Except where they are used in strings and 'CODE' statements, layout characters, that is space, horizontal tab, carriage return, page feed, line feed and rubout, are ignored by the compilers.

A program is made up of symbols (e.g. 'BEGIN',=,4) and arbitrary identifiers, which, by declaration, specification or setting, acquire the status of single symbols. Identifiers are names referring to objects, which are classified as follows:

(a) data (numbers, arrays of numbers, tables)
(b) places (labels and switches)
(c) procedures (functions and processes)

A program may be compiled in more than one unit. To make it possible to refer to chosen objects in more than one unit the names and types of such objects are written outside the body of each unit in communicators. A CORAL 66 program unit comprises the syntax shown in Figure 1-1.

```
'CORAL' name of program or unit
Optional communicators
'BEGIN'
Body of program unit
'END'
'FINISH'
```

Figure 1-1. Program Unit Syntax.

The body of the program unit with its enclosing 'BEGIN' and 'END' form a block.

When a program is compiled one of its units, designated as the master segment, is identified as the entry point of the program and the program starts running from the beginning of its outermost block. A complete CORAL 66 program is supplied in your CORAL 66 Operating Guide.

SYNTACTIC METALANGUAGE

The syntactic metalanguage used to describe CORAL syntax consists of rules, where each rule has on its left-hand side a class name, such as 'Statement'. Such names appear in lower case without spaces and with an initial capital letter. On the right-hand side of a rule are the various alternative expansions for the class name. These alternatives are each printed on a new line. Where a single alternative spreads over more than one line of print, the additional lines are inset in relation to the starting position of the alternative. Each alternative expansion consists of a sequence of items separated by spaces. The items themselves are either further class names or terminal symbols, such as 'BEGIN'. The class name 'Void' is used for an empty class. For example, a typical pair of rules might be

```
Specimen    =   'ALPHA' Sign
                'BETA' Sign
Sign        =   +
                -
                Void
```

Examples of Specimen complying with these rules are 'ALPHA'+ and 'BETA'.

A complete summary of syntax rules is available for reference in Appendix A of this manual.

CHAPTER 2

SCOPING

A named object may be brought into existence for part of a program and may have no existence elsewhere (but see PRESERVATION OF VALUES in Chapter 3).

The part of the program in which it is declared to exist is known as its scope. One effect of scoping is to increase the freedom of choosing names for objects whose scopes do not overlap. Another effect is economy of computer storage space. The scope of an object is determined by the block structure of the program.

BLOCK STRUCTURE

A block is a statement consisting, internally, of one or more declarations followed by one or more statements punctuated by semicolons and all bracketed by 'BEGIN' and 'END'.

The syntax is as shown in Figure 2-1.

```
Block       =   'BEGIN' Declist; Statementlist 'END'
Declist     =   Dec
                Dec; Declist
Dec         =   Datadec
                Overlaydec
                Switchdec
                Proceduredec
Datadec     =   Numberdec
                Arraydec
                Tabledec
```

Figure 2-1. Block Syntax.

The declarations have the purpose of fully classifying new objects and providing them with names (identifiers). As a statement can itself be a block, merely by having the right form, blocks can be nested to an arbitrary depth. Except for global objects (see GLOBALS), the scope of an object shall be the block in which it is declared and within this block the object is said to be local. The scope penetrates inner blocks, where the object is said to be non-local.

CLASHING OF NAMES

Two objects that have the same name cannot have identical scopes. If two objects have the same name and their scopes overlap, the clash of definitions could give rise to ambiguity. Typically, a clash occurs when an inner block is opened and a local object is declared to have the same name as a non-local object that already exists. In this situation, the non-local object continues to exist through the inner block (i.e., a variable maintains its value) but becomes temporarily inaccessible. The local meaning of the identifier always takes precedence.

GLOBALS

A program unit may refer to global objects. Such objects may be used in any unit of the program as their scope is the entire program. To become global an object is named in a communicator written before the body of the program unit. For some types of object, such as 'COMMON' data references, this takes the form of a declaration, and is the only declaration required. Other types of object, specifically 'COMMON' labels, 'COMMON' switches and 'COMMON' procedures, must be fully defined within the outermost block of a program unit. This means that 'COMMON' labels must be set, and 'COMMON' switches and procedures must be declared, in one of the outermost blocks of the program. Such objects are merely specified in a 'COMMON' or 'EXTERNAL' communicator (see 'COMMON' COMMUNICATORS in Chapter 8) and are treated as local in every outermost block of the program. Global objects declared in communicators are treated as non-local. All globals are non-local in all the inner blocks of any program unit. With these requirements for locality, questions of clashing are resolved as described under CLASHING OF NAMES earlier in this chapter.

LABELS

Any statement may be labelled by writing in front of it an identifier and a colon. The scope of a label is the smallest block embracing the statement that is labelled. Thus labels can be used before they have been set. It also follows that the only means of entering a block is through its 'BEGIN'. It is possible to jump into an outermost block from a different program unit by the use of a 'COMMON' label, 'COMMON' switch or 'COMMON' procedure.

NOTE: Labels alone do not convert a compound statement into a block. For a 'BEGIN', 'END' pair to constitute a block they must be the outermost such pair in a program unit or there must be one or more declarations following 'BEGIN'. It is thus possible, though undesirable, to jump into a compound statement.

RESTRICTIONS CONNECTED WITH SCOPING

No identifier other than a label must be used before it has been declared or specified. Specifications means that the type of object to which an identifier refers has been declared, but not necessarily the full definition of the object (see COMMON COMMUNICATORS in Chapter 8). Typically a procedure identifier is specified as referring to a certain type of procedure with certain types of parameters by the heading of the procedure declaration, but the procedure is not fully defined until the end of the declaration as a whole.

In this implementation all identifiers local to the same block may be considered to be declared simultaneously. As an example of this, assume that two procedures, F and G are declared in succession in a particular block, then each may call the other or itself, and each may refer to identifers that are local to the same block whatever their order of declaration.

CHAPTER 3

DATA REFERENCING

NUMERIC TYPES

There are three numeric types:

1. Floating point
2. Integer
3. Byte

Except in certain part-word table elements (see Part Word Table Elements), all three types shall be signed. Numeric type shall be indicated by the terminal symbols: 'FLOATING', 'INTEGER' or 'BYTE'.

The numeric type 'BYTE' occupies one memory byte i.e. 8 bits; the numeric type 'INTEGER' occupies two bytes (16 bits); the 'FLOATING' size and format depends on the compiler and options used. 'FLOATING' is not accepted if 'FLOATING' is excluded from the compiler when generated or by compile time option.

SIMPLE REFERENCES

The simple objects of data are single numbers of 'FLOATING' 'INTEGER' or 'BYTE' types. Simple references shall refer to such objects e.g.

```
'INTEGER'  I,J,K;
'BYTE'     X,Y;
```

The declarations may include assignment of initial values; this is known as presetting.

Requirements for presetting are specified later in this chapter. The syntax for a number declaration is shown in Figure 3-1.

```
Numberdec       = Unsetnumberdec Presetlist
Unsetnumberdec  = Numbertype Idlist
Idlist          = Id, Idlist
```

Figure 3-1. Number Declaration Syntax.

ARRAY REFERENCES

An array is restricted to a one-dimensional or two-dimensional set of numbers that are all of the same type. An array is represented by a suitably declared identifier with, for each dimension, a lower and upper index bound in the form of a pair of integer constants, e.g.

```
'INTEGER' 'ARRAY' B[0:10];
'FLOATING' 'ARRAY' C[1:3,1:3];
```

The lower bound cannot exceed the corresponding upper bound. If more than one array is required with the same numeric type and the same

dimensions and bounds, a list of array identifiers separated by commas may replace the single identifiers shown in the above examples. Arrays with the same numeric type but different bounds or dimensions may also occur in a composite declaration e.g.:

```
'INTEGER' 'ARRAY' P,Q,R[1:3],S[1:4],T,U[1:2,1:3];
```

An array identifier refers to an array in its entirety, but its use in statements is confined to the communication of the array reference to a procedure. Elsewhere, an array identifier must be indexed so that it refers to a single array element. Indices have the form of arithmetic expressions, separated by commas, enclosed in square brackets after the array identifier. Each index is then evaluated to an integer as specified in EVALUATION OF EXPRESSIONS in Chapter 5. The indices of a two-dimensional array are evaluated in the order of occurrence when reading the text from left to right. The syntax rules for an array declaration that includes a presetting facility (see PRESETTING OF SIMPLE REFERENCES AND ARRAYS later in this Chapter) are shown in Figure 3-2.

```
Arraydec        =  Unsetarraydec Presetlist
Unsetarraydec   =  Numbertype 'ARRAY' Arraylist
Arraylist       =  Arrayitem
                   Arrayitem, Arrayitem
Arrayitem       =  Idlist [Sizelist]
Sizelist        =  Dimension
                   Dimension, Dimension
Dimension       =  Lowerbound:Upperbound
Lowerbound      =  Signedinteger
Upperbound      =  Signedinteger
```

Figure 3-2. Array Declaration Syntax.

PACKED DATA

PRELIMINARY

There are two methods of referring to packed data; one in which an unnamed field is selected from any computer byte or adjacent pair of bytes (see PART PRIMARIES in Chapter 5) and the other in which the data format is declared in advance. In the latter method the format is replicated to form a table. A group of several bytes (n) may be partitioned into bit fields (where no field may extend into more than two bytes), and the same partitioning shall be applied to as many such groups (m) as are required. The total data space in bytes for a table is then the multiple of bytes in a group by number of groups (nm). Each group shall be known as a table-entry. The fields must be named, so that a combination of field identifier and entry index selects data from one or more computer bytes, known as a table-entry. The elements in an entry may occupy overlapping fields and may leave unfilled spaces in the entry.

TABLE DECLARATION

A table declaration serves two purposes:

1. To provide the table with an identifier, and to associate this identifier with an allocation of storage sufficient for the width and number of entries specified, e.g.

   'TABLE' APRIL [8,30]

   is the beginning of a declaration for the table APRIL with 30 entries each eight bytes wide, requiring an allocation of 180 bytes in all.

2. To specify the structure of an entry by declaring the elements contained within it, as specified in TABLE-ELEMENT DECLARATION in this Chapter. Data-packing in this implementation involves no 'slack' bytes, each entry occupies the declared number of bytes and the next entry follows immediately.

The general syntax for a table declaration is given in Figure 3-3.

```
Tabledec   =  'TABLE' Id Tableform Presetlist
              'TABLE' Id [Width,Length][Elementdeclist Elementpresetlist]
Tableform  =  [Width,Length][Elementdeclist]
Elementdeclist
           =  Elementdec
              Elementdec;Elementdeclist
Width      =  Integer
Length     =  Integer
```

Figure 3-3. Table Declaration Syntax.

NOTE: Requirements for the two presetting mechanisms are specified in PRESETTING OF TABLES later in this Chapter.

TABLE-ELEMENT DECLARATION

General

A table-element declaration associates an element name with a numeric type and with a particular field of each and every entry in the table. The field must be a whole 'BYTE' 'INTEGER' or 'FLOATING' number or a part of one or two computer bytes and the form of declaration differs accordingly. The syntax for an element declaration is given in Figure 3-4.

```
Elementdec      =  Id Numbertype Byteposition
                   Id Partwordtype Byteposition Bitseparator Bitposition
Byteposition    =  Signedinteger
Bitseparator    =  'BIT'
                   ,
Bitposition     =  Integer
```

Figure 3-4. Element Declaration Syntax.

Bitposition must be numbered from zero upwards, and the least significant bit of a word is designated bit-position zero. Normally, table-elements should be located so that they fall within the declared width of the table but compilers do not check the limits. To improve program legibility the language word 'BIT' is provided as an alternative to the comma. The meaning of Bitposition is specified in Part-Word Table Elements later in this Chapter.

Whole Numbertype Table-Elements

As specified above the form of declaration for whole 'BYTE', 'INTEGER' or 'FLOATING' table-elements is shown in Figure 3-5.

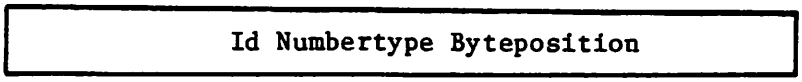

Figure 3-5. Whole Numbertype Table Element Syntax

EXAMPLES:

For example:

```
TICKETS 'INTEGER' 0
```

declares a 'pseudo-array' of elements named TICKETS. (True array elements are located consecutively in store - see STORAGE ALLOCATION.) Each element shall refer to a (signed) integer at byte-position zero in an entry. Similarly, the example:

```
WEIGHT 'FLOATING' 2
```

locates WEIGHT from byte position 2 onwards.

Part-Word Table-Elements

Elements that occupy fields that are not whole 'INTEGER', 'FLOATING' or 'BYTE' numbers, aligned on computer bytes, must be declared as follows:

```
RAIN 'UNSIGNED' (4) 6,0
HUMIDITY 'UNSIGNED' (6) 7,0;
TEMPERATURE (10) 7,6;
```

for part-word integer elements. The number of bits required for the field are given in brackets followed by the byte- and bit-position of the field within the entry. Byte-position is the byte in which the least significant (and lowest addressed) bit of the field is located, and bit-position is the position of the least significant bit in the specified byte.

The word 'UNSIGNED' increases the capacity of the field for positive numbers at the expense of eliminating negative numbers. For example, RAIN 'UNSIGNED' (4) above, allows numbers from zero to 15, while TEMPERATURE (10) allows from -2^9 to (2^9-1).

The syntax of partwordtype, for substitution in the syntax in Figure 3-4 is shown in Figure 3-6.

```
Partwordtype   =   Elementscale
                   'UNSIGNED' Elementscale
Elementscale   =   (Totalbits).
```

Figure 3-6. Syntax for Partwordtype in Element Declaration.

Part-word integer elements may be up to 15 bits long if 'UNSIGNED' or 16 bits if signed, but their starting bit-position and length must be such that they do not extend into more than two computer bytes.

COMPLETE TABLE DECLARATION

The complete table declaration built up so far as an illustrative example might be:

```
'TABLE' APRIL [6,30]
              [TICKETS 'INTEGER' 0;
               WEIGHT 'FLOATING' 2;
               RAIN 'UNSIGNED' (4) 6,0;
               SUNSHINE 'UNSIGNED' (4) 6,4;
               HUMIDITY 'UNSIGNED' (6) 7,0;
               TEMPERATURE (10) 7,6]
```

All the numbers used to describe and locate fields must be constants.

REFERENCES TO TABLES AND TABLE-ELEMENTS

A table-element is selected by indexing its field identifier. To continue with the example used above, the rain for 6 April would be written RAIN [5]. (An entry shall always have the conventional lower bound of zero). In use, the names of table-elements are always indexed, although a table identifier such as APRIL can stand on its own when a table reference is passed to a procedure. The use of an index with a table identifier shall select a byte from the table regarded as a conventional array of 'BYTE' data with lower index bound zero. Thus the implied bounds of APRIL are 0:179. A table name is normally indexed only for the purpose of running through the table systematically, for example to set all data to zero, or to form a base for overlaying.

STORAGE ALLOCATION

Computer storage space for data is allocated automatically at compile time. One or more bytes (according to the number type) are allocated for each simple reference and for each array element, and as many bytes are allocated as are declared for each table-entry. In any one composite declaration, the compiler performs allocation serially.

EXAMPLE:

```
'BYTE' A,B,C;
'INTEGER' P,Q;
```

The locations of A, B and C become n, n+1 and n+2 respectively, and those of P and Q become m and m+1, m+2 and m+3 respectively, where m and n are undefined and unrelated.

An 'INTEGER' number has its less significant byte at the lower address.

In two-dimensional arrays, the second index is stepped first; the declaration

```
'ARRAY' A[1:2],B[1:2,1:2]
```

locates the elements

```
A[1],A[2],B[1,1],B[1,2],B[2,1],B[2,2]
```

in consecutive ascending locations.

PRESETTING

GENERAL

Some objects of data may be initalized when the program is loaded into store by the inclusion of a presetting clause in the data declaration. Presetting is not dynamic, and preset values that are altered by program are not restored unless the program is reloaded. If preset data and program instructions are ultimately stored in read-only memory it is not possible for values to be altered by program (although the compiler does not check this) and presetting should in such cases be reserved for constant data.

An object shall not be eligible for presetting if it is defined in an 'OVERLAY' declaration. Objects defined in the body of a 'RECURSIVE' procedure may be preset; note however that such objects are not replicated for each activation of the procedure and if the value of such an object is altered, it will appear altered to all activations of the procedure.

PRESETTING OF SIMPLE REFERENCES AND ARRAYS

The preset constants must be listed at the end of the declaration after an assignment symbol, and are allocated in the order already specified e.g.,

```
'INTEGER' A,B,C:=1,2,3;
'INTEGER' 'ARRAY' K[1:2,1:2]:=11,12,21,22;
```

If desired for legibility, round brackets may be used to group items of the presetlist, but such brackets are ignored by the compiler except for checking that they occur as matched pairs. The number of constants in the presetlist must not exceed, but may be less than, the number of references or array elements declared, and presetting ceases when the presetlist is exhausted. The preset assignment symbol may optionally be the only part of the presetlist present. The general syntax is given in Figure 3-7.

Presetlist	=	Assignmentsymbol Constantlist
Assignmentsymbol	=	:=
Constantlist	=	Group, Group, Constantlist
Group	=	Constant (Constantlist) Void

Figure 3-7 General Syntax for Preset Assignment.

NOTE. The main purpose of the final void may be seen by reference to table presetting that follows. For the expansion of constants see NUMBERS in Chapter 9.

PRESETTING OF TABLES

Two alternative mechanisms are available for presetting tables.

1. Either the internal structure of a table is completely disregarded and the table treated as an ordinary one-dimensional array of 'BYTE' data, and preset as such.

or,

2. All the table-elements are preset after their declaration list, as shown at Elementpresetlist in the syntax specified in Figure 3-1 e.g.,

```
'TABLE' GEARS [3,3]
          [TEETH1 'UNSIGNED'(6) 0,0;
           TEETH1 'UNSIGNED'(6) 0,6;
           RATIO' UNSIGNED'(11) 0,12;
           ARC 'UNSIGNED'(5) 0,12
           'PRESET' (57,19,3,),(50,25,2),(45,5,9,)]
```

For table-element presetting the word 'PRESET' shall be used instead of the assignment symbol specified in Figure 3-7. Each entry of the table must be preset in succession as a group of elements, taken in the order of their declaration. Voids in the list imply absence of any assignment; this may be necessary to avoid duplication when fields overlap, as do RATIO and ARC in the foregoing example. As specified in Figure 3-7 brackets used for grouping constants in the list of presets are ignored by the compiler. The general syntax is given in Figure 3-8.

```
Elementpresetlist = 'PRESET' Constantlist
                    Void
```

Figure 3-8 Table Element Presetting Syntax.

The previous example could, with equal effect but less convenience, be expressed in the form:

```
'TABLE' GEARS [3,3]
          [TEETH1 'UNSIGNED'(6) 0,0;
           TEETH1 'UNSIGNED'(6) 0,6;
           RATIO' UNSIGNED'(11) 0,12;
           ARC 'UNSIGNED'(5) 0,12]
          : =  'HEX'(F9),'HEX'(34),0,
               'HEX'(72),'HEX'(26),0,
               'HEX'(6D),'HEX'(91),0
```

PRESERVATION OF VALUES

Objects of data that have not been preset are not required to have existence outside the scope of their declarations.

The values to which local identifiers refer are in general assumed to be undefined when a block is first entered and whenever it is subsequently re-entered.

NOTE: This is consistent with the fact that a block structured language is designed for automatic overlaying of data. Local working space may therefore have been used for other purposes between one entry to a block and the next.

When a data declaration contains a presetlist as permitted by the Presetting requirements, the values of all the objects named in that declaration remain undisturbed between successive entries to the block or procedure body, like 'own' variables in ALGOL 60. Appearance of a preset assignment symbol, or in tables the word 'PRESET', suffice even though the list of preset constants is void.

OVERLAY DECLARATIONS

Overlaying may be found desirable to enable global data space to be reused whilst not required for its primary purpose, or to allow apparently different data references to refer simultaneously to the same objects of data, i.e., as alternative names and definitions of the same storage locations.

NOTE: Indiscriminate use of overlaying should be avoided as it can lead to confusion and obscurity.

To form an overlay declaration, an ordinary declaration must be preceded by a phrase of the form

'OVERLAY' Base 'WITH'

where Base is a data reference that has previously been covered by a declaration in the same 'COMMON' communicator or in the same program unit. The base shall be a simple reference, a one-dimensional array reference or a table reference treated as a one-dimensional array of 'BYTE' data. If the array or table identifier is not indexed, it must refer to the location of its zero element (which may be conceptual). Storage allocated by the overlay declaration shall start from the base, shall proceed serially (as already specified) and shall not be overlaid by succeeding declarations unless these are themselves overlay declarations. There is no requirement to re-order storage that is already allocated. The syntax of an overlay declaration is shown in Figure 3-9.

```
Overlaydec    =   'OVERLAY' Base 'WITH' Unsetdatadec
Base          =   Id
                  Id[Signedinteger]
```

Figure 3-9. Overlay Declaration Syntax.

Note that presetting is not allowed. If the storage overlaid contains preset objects then those preset values determine the initial values of the overlaying objects; otherwise the storage is variable (and re-usable) and no initial values can be assumed.

CHAPTER 4

PLACE REFERENCING

Place references refer to positions of program statements and the simplest marker is the label. A switch is a preset and unalterable array of labels with lower index bound one. These labels must be within scope at the switch declaration. Any use of the indexed switch name must refer to the corresponding label. For example the switch declaration

```
'SWITCH' S:= A,B,C
```

causes S[1] to refer to the label A, S[2] to B and S[3] to C.

The general syntax is shown in Figure 4-1.

```
Switchdec =    'SWITCH' Switch Assignmentsymbol Labellist
Labellist =    Label
               Label, Labellist
Switch    =    Id
Label     =    Id
```

Figure 4-1. Switch Syntax.

CHAPTER 5

EXPRESSIONS

The term 'expression' is reserved for arithmetic expressions. CORAL 66 has no designational expressions of ALGOL 60 type. As there are no Boolean variables and no bracketed Boolean expressions (see CONDITIONAL EXPRESSIONS later) the expressions after 'IF' are termed conditions. The general syntax for an expression is shown in Figure 5-1.

Expression	=	Unconditionalexpression Conditionalexpression
Unconditional expression	=	Simpleexpression String

Figure 5-1 General Expression Syntax.

NOTE: Requirements for strings are specified in Chapter 9.

SIMPLE EXPRESSIONS

Arithmetic is performed with the monadic and dyadic adding operators + and -, and with the dyadic multiplying operators * (multiply), / (divide) and 'MOD' (remainder of division). The plus and minus operators join terms. The multiplication, division and remainder operators join factors to form terms. There are no exponentiation operators. The general syntax for simple expression is shown in Figure 5-2.

Simpleexpression	=	Term Addoperator Term Simpleexpression Addoperator Term
Term	=	Factor Term Multoperator Factor
Addoperator	=	+ -
Multoperator	=	* / 'MOD'

Figure 5-2. Expressions Syntax.

The 'MOD' operator delivers the remainder of the first operand divided by the second. 'MOD' is not defined on 'FLOATING' data; apart from this restriction all operators may apply to all numeric types.

PRIMARIES

GENERAL

Primaries are the basic operands in expressions, e.g. in the analysis of the expression

X + Y * (A + B) - 4

there are three terms, the primary X, the term Y*(A+B) and the primary

4. The middle term is the product of two factors the primary Y and the primary (A+B). To complete the analysis, all expressions from within brackets are similarly analysed until no further reduction is possible and no expression brackets remain. When an expression contains no word-logical operators a factor shall be a primary, whether or not of a defined type. The syntax for a Primary operand is shown in Figure 5-3.

```
Factor    =  Primary
             Booleanword

Primary   =  Untypedprimary
             Typedprimary
```

Figure 5-3. Primary Operand Syntax.

UNTYPED PRIMARIES

Untyped primaries are those operands that cannot be classed as integer, floating-point or byte without reference to their context, e.g. the number 3.1416 may be represented with varying degree of accuracy in each of the number types. The same applies to an expression, whose type is determined by context. The syntax is:

```
Untyped primary  =  Real
                    (Expression)
```

Figure 5-4. Untyped Primary Syntax.

A 'real' (see NUMBERS in Chapter 9) is an unsigned numerical constant containing a decimal, octal or hexadecimal point or a tens exponent, or a decimal point and a tens exponent.

TYPED PRIMARIES

Typed primaries are classified as follows:

```
Typed primary  =  Reference
                  Partprimary
                  'LOCATION' (Reference)
                  Numbertype (Expression)
                  Procedurecall
                  Integer
```

Figure 5-5. Typed Primary Syntax.

REFERENCES

A simple reference, or a reference to an array element or whole numbertype table-element has a type defined in its declaration. Such references, termed References in the formal syntax, refer to data for which a whole number of computer bytes are set aside. A further kind of Reference, the anonymous reference, shall take the form

where the index is any expression evaluated as an integer to give the actual location of a computer byte. An anonymous reference possesses all the properties of an identified reference, except that it lacks an identifier. Just as a variable I, declared as 'INTEGER' I, may be used in an expression to refer to the contents of the two computer bytes allocated to I, so the use of an anonymous reference in an expression will refer to the contents of addresses defined by Index and (Index + 1). Such contents will be taken to be of numeric type 'INTEGER' , irrespective of any declaration associating that storage with some other type. (See also LOCATIONS later.) The syntax for a Reference is shown in Figure 5-6.

```
Reference  =  Id
              Id [Index]
              Id [Index, Index]
              [Index]
Index      =  Expression
```

Figure 5-6. Reference Syntax.

PART-PRIMARIES

Any single item of packed data may act as a typed primary. Such an item can be either:

1. a reference to a part-word table-element; or
2. a specified field of any typed primary.

In (1), the type is defined in the table declaration. in (2), the desired field is selected by a prefix of the form

'BITS' [Totalbits, Bitposition]

in front of the typed primary to be operated upon. The result of this operation is a positive integer value of width Totalbits and in units of the bit at 'Bitposition'. Total bits must not be set equal to the full size of 'INTEGER' (i.e. Totalbits must be less than 16). The syntax for a part-primary, which should be distinguished from that of a 'part-reference' is:

```
Partprimary  =  Id [Index]
                'BITS' [Totalbits, Bitposition] Typedprimary
```

Figure 5-7. Part Primary Syntax.

LOCATIONS

The computer location of any reference is obtainable by the location operator, which is written in the form

'LOCATION' (Reference)

and has a value of type 'INTEGER'.

NOTE: If I and J refer to integers, ['LOCATION'(I)] is equivalent to I, and 'LOCATION'([J]) is equivalent to J. The reasoning is as follows. 'LOCATION'(I) is the address of the 'INTEGER' I. Enclosure in square brackets forms an entity equivalent to an identifier standing for this address, which by hypothesis is I. Similarly [23] is equivalent to an identifier for the address 23 and 'LOCATION'([23]) is the address for which this fictitious identifier stands, which is 23 by hypothesis.

EXPLICIT TYPE-CHANGING

A typed primary may have its type changed, and an untyped primary may be typed, by enclosure within round brackets preceded by a specific Numbertype as specified in TYPED PRIMARIES earlier in this Chapter.

FUNCTIONS

The call of a typed procedure (see Chapter 7) may be treated as a function and used as a primary in any expression. (For the syntax of a procedure call, see Figure 6-4).

INTEGERS

An integer used in any expression (see Chapter 9) is assumed to have the numeric type 'INTEGER' before any necessary type-changes enforced by context. However a small integer, defined by the inclusive range -128 to +255 shall be assumed to have type 'BYTE' initially.

WORD-LOGIC

Five dyadic logical operators are defined for use between primaries. Three of these operators, concerned with operations on corresponding bits of their operands, are defined in the British Standard. The other two, 'SRL' and 'SLL' are extensions of the standard.

The standard operators combine corresponding bits of the operands as follows. The i th bit of the result is a given logical function of the i th bits of the two operands, and the result as a whole is a typed primary of numeric type 'INTEGER' except where both operands are of type 'BYTE' when the result shall be of type 'BYTE'.

The operators are:

'DIFFER'

	0	1
0	0	1
1	1	0

'UNION'

	0	1
0	0	1
1	1	1

'MASK'

	0	1
0	0	1
1	0	1

'DIFFER' is recognizable as 'not equivalent', 'UNION' as 'inclusive or' and 'MASK' as 'and'. The shift operators are 'SLL' (shift left logical) and 'SRL' (shift right logical). The result has the type of the first operand and a value equal to the bit pattern representing the first operand shifted in the specified direction by the number of bits specified in the second operand. The second operand must always be evaluated to type 'BYTE' and no defined result shall be given for negative values thereof.

The syntax, continued from the GENERAL Syntax given under PRIMARIES is:

```
Booleanword      =  Booleanword 2,
                    Booleanword 4 'DIFFER' Booleanword 5
Booleanword 2    =  Booleanword 3,
                    Booleanword 5 'UNION' Booleanword 6
Booleanword 3    =  Booleanword 6 'MASK' Shiftexpression
Booleanword 4    =  Booleanword
                    Shiftexpression
Booleanword 5    =  Booleanword 2
                    Shiftexpression
Booleanword 6    =  Booleanword 3
                    Shiftexpression
Shiftexpression  =  Primary
                    Shiftexpression
Shiftop          =  'SLL'
                    'SRL'
```

Figure 5-8. Word Logic Syntax.

EVALUATION OF EXPRESSIONS

Expressions are used in assignment statements, as value parameters of procedures and as integer indexes, all of which contexts determine the numeric type finally required. CORAL 66 expressions are automatically evaluated to this type, but in the process of calculation, data may be subjected by the compiler to various intermediate transformations.

For the operators 'SLL' and 'SRL' the second operand is always evaluated to type 'BYTE'. For all other dyadic operations the operands are evaluated to the same type.

All syntactically outermost terms in an expression are evaluated to the required numeric type before the adding operators are applied. If an expression is enclosed in round brackets, its terms are not 'outermost' and this requirement no longer applies.

The expressions on either side of a Comparator (see CONDITIONS later) will be automatically evaluated to the same type, and the type is 'FLOATING' if either expression is 'FLOATING', else 'INTEGER' if either expression is 'INTEGER' else 'BYTE'. However 'FLOATING' will not be used when 'FLOATING' point is not installed in the compiler or is suppressed by compile-time option.

The programmer may impose any desired system of evaluation by the use of Numbertype (Expression), which is a typed primary and any occurrence of which behaves like a variable, e.g. REF, declared as

```
Numbertype REF;
```

and assigned a value by

```
REF Assignmentsymbol Expression
```

before it is used. For example if I and J are 'INTEGER' references and X is a 'FLOATING' reference the assignment statement

```
X:=I-J
```

causes I and J to be converted to floating-point before subtraction, whilst

```
X:='INTEGER' (I-J)
```

causes subtraction of integers before conversion to floating-point. Although the order of evaluation of an expression is unspecified, the following requirement concerning functions shall apply. Value parameters are necessarily evaluated before the function itself is computed, so that, for example, the order of evaluation of SIN (COS (Expression)) is Expression, COS, SIN. Apart from this type of reversal, functions occuring in a simple expression are evaluated in the order in which they appear when the expression is read from left to right, regardless of brackets.

CONDITIONAL EXPRESSIONS

GENERAL

The general syntax for a conditional expression is:

```
Conditionalexpression = 'IF' Condition 'THEN' Expression
                                              'ELSE' Expression
```

Figure 5-9. Conditional Expression Syntax.

The expressions following 'THEN' and 'ELSE' are known as the consequent expression and the alternative expression respectively. The value of a conditional expression is the value of the consequent expression if the condition is true; it is the value of the alternative expression if the condition is false.

The numeric type used to evaluate the condition has no effect on the evaluation of the consequent or alternate expressions. Consequent and alternative expressions are not prevented from being regarded as syntactically outermost by their appearance in a conditional expression.

CONDITIONS

A condition consists of one or more arithmetic comparisons. Comparisons are connected by the Boolean operators 'OR' and 'AND', of which 'AND' takes precedence. The permissible arithmetic comparators are 'less than', 'less than or equal to', 'equal to', 'greater than or equal to', 'greater than', and 'not equal to'. The general syntax is shown in Figure 5-10.

```
Condition          =   Condition 'OR' Subcondition
                       Subcondition
Subcondition       =   Subcondition 'AND' Comparison
                       Comparison
Comparison         =   Expression Comparator Expression
Comparator         =   <
                       <=
                       =
                       >=
                       >
                       < >
```

Figure 5-10. Condition Syntax.

The Boolean operators have their usual meanings, the 'OR' being inclusive. Conditions and subconditions are evaluated from left to right only as far as necessary to determine truth or falsity. Comparisons are evaluated in the order in which they appear when a condition is read from left to right.

NOTE: No overflow detection is required by the definition of CORAL 66. Comparisons may give rise to overflow if the algebraic difference of the values compared exceeds the range of the particular number type even though the compared values are within the range. This range is particularly limited for 'BYTE' quantities.

CHAPTER 6

STATEMENTS

The general syntax of a statement is shown in Figure 6-1.

```
Statement          =  Label:Statement
                      Simplestatement
                      Conditionalstatement
Simplestatement    =  Assignmentstatement
                      Gotostatement
                      Procedurecall
                      Answerstatement
                      Codestatement
                      Compoundstatement
                      Block
                      Dummystatement
                      Forstatement
                      Returnstatement
                      Conditionalstatement2
```

Figure 6-1. General Statement Syntax.

Statements are executed in the order in which they are written, except that a goto statement may interrupt this sequence without return, and a conditional statement may cause certain statements to be skipped.

ASSIGNMENTS

The left-hand side of an assignment statement must always be a data reference, and the right-hand side must be an expression for procuring a numerical value. The result of the assignment is that the left-hand side refers to the new value until this is changed by further assignment, or until the value is lost because the reference goes out of scope (but see PRESERVATION OF VALUES in Chapter 3). The expression on the right hand side is evaluated to the numeric type of the reference, with automatic type conversion as necessary. Functions occurring in an assignment statement are evaluated in the order in which they are encountered when reading the text from left to right. The left-hand side may be a reference or a part reference i.e. a part-word table element or some selected field of a Reference. When assignment is made to a part reference, the remaining bits of the computer byte or bytes shall remain unaltered. As examples of assignment,

```
'INTEGER' I;
I:=4
```

has the effect of placing the integer 4 in the location allocated to I, and

```
'BITS'[2,6]X:=3
```

has the effect of placing the binary digits 11 in bits 7 and 6 of the first or only byte allocated to X. This last statement is treated in a similar manner to an assignment that has on its left-hand side an unsigned integer table-element. The statement

```
'BITS'[1,15]['LOCATION'(I)+2]:=1
```

has the effect of forcing the sign bit of the two bytes immediately following I to 'one'.

The general syntax of the assignment statement is shown in Figure 6-2.

```
Assignmentstatement =  Variable Assignmentsymbol Expression
Variable            =  Reference
                       Partreference
Partreference       =  Id [Index]
                       'BITS' [Totalbits, Bitposition] Reference
```

Figure 6-2. Assignment Statement Syntax.

There is no form of multiple assignment statement.

GOTO STATEMENTS

The goto statement causes the next statement for execution to be the one having a given label. The label may be written explicitly after 'GOTO' , or referenced by means of a switch whose index lies within the range 1 to n, where n is the number of labels specified in the switch declaration. The syntax is as shown in Figure 6-3.

```
Gotostatement =  'GOTO' Destination
Destination   =  Label
                 Switch [Index]
```

Figure 6-3. GOTO Statement Syntax.

NOTE: No range checking of a switch index is performed so that programs requiring such range checking must include explicit statements to validate the index.

PROCEDURE STATEMENTS

A procedure identifier, followed in parentheses by a list of actual parameters (if any) is known generally as a procedure call. If the procedure possesses a value, it may be used as a primary in an expression, but whether it possesses a value or not, a procedure call may also stand alone as a statement. The call of the procedure causes:

1. the formal parameters in the procedure declaration to be replaced by the actuals in a manner that depend on the formal parameter specifications (see PARAMETER SPECIFICATION in Chapter 7); the replacement is effected in the order in which the parameters are read when reading from left to right

2. the procedure body to be executed before the statement dynamically following the procedure statement is obeyed.

The syntax for a procedure call is shown in Figure 6-4.

```
Procedurecall  =  Id
                  Id(Actuallist)
Actuallist     =  Actual
                  Actual,Actuallist
Actual         =  Expression
                  Reference
                  Destination
                  Name
Name           =  Id
```

Figure 6-4. Procedure Call Syntax.

NOTE: The purpose of the four types of actual parameter is described in PARAMETER SPECIFICATION in Chapter 7.

ANSWER STATEMENTS

An answer statement is used only within a typed procedure body, and is the means by which a value is given to the procedure. It causes the expression in the answer statement to be evaluated to the numeric type of the procedure, followed by immediate exit from the procedure body. The syntax is:

```
Answerstatement  =  'ANSWER' Expression
```

Figure 6-5. Answer Statement Syntax.

CODE STATEMENTS

Any sequence of assembler source statements enclosed by 'CODE''BEGIN' and 'END' may be used as a CORAL 66 statement. For communication between code and other statements it is possible to use any CORAL 66 identifier of the program within the code statement, provided that the identifier is in scope. Such use of an identifier must be indicated by preceding it by a %. The identifier must also be followed by a printing delimiter (such as semicolon). The syntax for a code statement is:

```
Codestatement  =  'CODE''BEGIN' Codesequence 'END'
Codestatement  =  assembler instructions in which % name
                  refers to CORAL defined name
```

Figure 6-6. Code Statement Syntax.

NOTE: Further requirements for code statements, examples of code statements and details of the text substituted by the compiler for % identifier are given in the appropriate CORAL 66 Operating Guide.

COMPOUND STATEMENTS

A compound statement is a sequence of statements grouped to form a single statement, for use where the syntactic structure of the language demands. A compound statement is transparent to scopes and it is permissible to goto a label that is set inside a compound statement.

The syntax is shown in Figure 6-7.

```
Compoundstatement   =   'BEGIN' Statementlist 'END'
Statementlist       =   Statement
                        Statement;Statementlist
```

Figure 6-7. Compound Statement Syntax.

BLOCKS

See Chapter 2.

DUMMY STATEMENTS

A dummy statement is a void whose execution has no effect e.g. a dummy statement follows the colon in:

```
;label:'END'
```

The syntax is shown in Figure 6-8.

```
Dummystatement = Void
```

Figure 6-8. Dummy Statement Syntax.

CONDITIONAL STATEMENTS

The syntax of the conditional statement is:

```
Conditionalstatement    =   'IF' Condition 'THEN' Consequence
Consequence             =   Statement
Conditionalstatement2   =   'IF' Condition 'THEN' Consequence2
                            'ELSE' Alternative
Consequence2            =   Simplestatement
                            Label: Consequence2
Alternative             =   Statement
```

Figure 6-9. Conditional Statement Syntax.

NOTE: Conditionalstatement is a possible form of Statement while Conditionalstatement2 is a possible form of Simplestatement. In effect each 'ELSE' clause shall be matched with the most recent 'IF' for which there has not yet been an 'ELSE' clause.

FOR STATEMENTS

GENERAL

The for-statement comprises a means of repeatedly executing a given statement, the 'controlled statement', for different values of a chosen variable, the 'control variable', which may (or may not) occur within the controlled statement. The effect of jumps into the controlled statement are that the control variable exists but has an undefined value, and subsequent execution of the body may have detrimental and unpredictable effects.

One form of for-statement is

```
'FOR' I: =    1 'STEP' 1 'UNTIL' 4,
              6 'STEP' 2 'UNTIL' 10,
              15 'STEP' 5 'UNTIL' 30
              'DO' Statement
```

Other forms are exemplified by

```
'FOR' I: = 1,2,4,7,15 'DO' Statement
```

which is self-explanatory, and

```
'FOR' I: = I+1 'WHILE' X<Y 'DO' Statement
```

In the third example, the clause

```
I +1 'WHILE' X<Y
```

counts as a single for-element and could be used as one element in a list of for-elements (the 'for-list').

As each for-element is exhausted, the next element in the list is used. The syntax is shown in Figure 6-10.

```
Forstatement = 'FOR' Reference Assignmentsymbol Forlist 'DO' Statement
Forlist      = Forelement
               Forelement,Forlist
Forelement   = Expression
               Expression 'WHILE' Condition
               Expression 'STEP' Expression 'UNTIL' Expression
```

Figure 6-10. For-Element Syntax.

The control variable must be a Reference i.e. either an anonymous reference or a declared whole numbertype. The location of the control variable is evaluated once only, prior to evaluation of the for-list.

FOR-ELEMENTS WITH 'STEP'

EXAMPLE:

Consider the element is denoted by:

el 'STEP' e2 'UNTIL' e3

The expressions are evaluated once only. First their values are evaluated in the order in which they are met when reading from left to right. If these values be denoted by v1, v2 and v3 respectively. Then, in sequence:

1. v1 is assigned to the control variable
2. v1 is then compared with v3; if (v1-v3)*v2>0, the for-element shall be exhausted; otherwise
3. the controlled statement is executed;
4. the value v1 is set from the control variable, then incremented by v2 and the cycle repeated from step 1.

FOR-ELEMENTS WITH 'WHILE'

Consider that the element is denoted by:

el 'WHILE' Condition

Then, in sequence:

1. el is evaluated and assigned to the control variable
2. The condition is tested; if false the for-element is exhausted; otherwise
3. The controlled statement is executed and the cycle repeated from step 1.

Unlike the expressions considered in FOR ELEMENTS WITH STEP the expression el and those occurring in the condition are evaluated repeatedly.

RETURN STATEMENTS

A return statement shall be used only within an untyped procedure body, and shall be similar to an answer statement in that it shall cause an immediate exit from the procedure body. The syntax is shown in Figure 6-11.

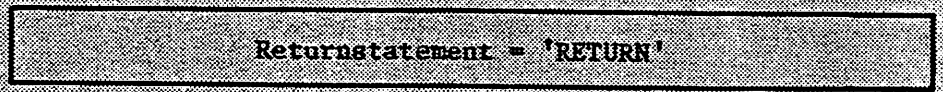

Figure 6-11. Return Statement Syntax.

CHAPTER 7

PROCEDURES

A procedure is a body of program, written once only, named with an identifier, and available for execution anywhere within the scope of the identifier. There are three possible methods of communication between a procedure and its program environment, as follows.

1. The body uses formal parameters, of types specified in the heading of the procedure declaration and represented by identifiers local to the body. When the procedure is called, the formal parameters are replaced by actual parameters, in one-to-one correspondence.

2. The body uses non-local identifiers whose scopes embrace the body. Such identifiers are also accessible outside the procedure.

NOTE:For the 8086 compiler this does not apply to non-local identifiers declared within a 'RECURSIVE' procedure which are not available within inner procedure bodies (irrespective of the specification of any such inner procedures).

3. An answer statement within the procedure body shall compute a single value for the procedure, making its call suitable for use as a function in an expression. A procedure that possesses a value shall be known as a typed procedure.

The syntax for a procedure declaration is shown in Figure 7-1.

```
Proceduredec  =  Answerspec 'PROCEDURE' Procedureheading;Statement
                 Answerspec 'RECURSIVE' Procedureheading;Statement
```

Figure 7-1. General Syntax for a Procedure Declaration.

The second of the foregoing syntax alternatives is the form of declaration for recursive procedures. If a procedure is defined in a manner that directly or indirectly calls itself at run-time, that procedure is said to 'recursive' and must be explicity defined as such. The statement following the procedure heading is the procedure body, which contains an answer statement unless the answer specification is void, and which shall be treated as a block whether or not it includes any local declarations.

ANSWER SPECIFICATION

The value of a typed procedure shall be given by one or more answer statements (see Chapter 6) in its body, and its numeric type must be specified at the front of the procedure declaration. An untyped procedure has no answer statement, possesses no value, and has a void answer specification before the word 'PROCEDURE' or 'RECURSIVE'. The syntax is shown in Figure 7-2.

```
Answerspec  =  Numbertype
               Void
```

Figure 7-2. Answer Specification Syntax.

PROCEDURE HEADING

The procedure heading shall give the procedure its name. It shall also describe and list any identifiers used as formal parameters in the body. On a call of the procedure, the compiler shall set up a correspondence between the actual parameters in the call and the formal parameters specified in the procedure heading. The syntax of the heading is shown in Figure 7-3.

```
Procedureheading    =   Id
                        Id(Parameterspeclist)
Parameterspeclist   =   Parameterspec
                        Parameterspec;Parameterspeclist
```

Figure 7-3. Procedure Heading Syntax.

PARAMETER SPECIFICATION

GENERAL

Any object in scope at the position of a procedure call may be passed to that procedure by means of a parameter, whether it is an object of data, a place in a program, or a procedure to be executed. For data there are two distinct levels of communication; numerical values (for input to the procedure) and data references (for input or output). Table 7-1 specifies all the types of object that may be passed, the syntactic form of specification, and the corresponding form of actual parameter that shall be supplied in the procedure call. The syntax is shown in Figure 7-4:

```
Parameterspec   =   Specifier Idlist
                    Tablespec
                    Procedurespec
Specifier       =   'VALUE' Numbertype
                    'LOCATION' Numbertype
                    Numbertype 'ARRAY'
                    'LABEL'
                    'SWITCH'
```

Figure 7-4. Parameter Specification Syntax.

Table 7-1. Parameters of Procedures

Object	Formal Specification	Actual Parameter
Numerical value	'VALUE' Numbertype Id[1]	Expression
Location of date	'LOCATION' Numbertype Id[1]	Reference
Name of array	Numbertype 'ARRAY' Id[1]	Id
Name of table	Tablespec[2]	Id
Place in program	'LABLE' Id[1]	Destination
Name of switch	'SWITCH' Id[1]	Id
Name of procedure	Procedurespec[3]	Id

1 - Composite specification has Idlist in place of Id

2 - See TABLE PARAMETERS in this Chapter

3 - See PROCEDURE PARAMETERS in this Chapter

VALUE PARAMETERS

The formal parameter is treated as though declared in the procedure body; upon entry to the procedure, the actual expression is evaluated to the type specified, and the value forthwith assigned to the formal parameter.

The formal parameter may be used subsequently for working space in the body; if the actual parameter is a variable, its value is unaffected by assignments to the formal parameter.

DATA REFERENCE PARAMETERS

Location, array and table parameters are all examples of data references. Upon entry to the procedure, these formals are made to refer to the same computer locations as those to which the actual parameters already refer. Operations upon such formal parameters within the procedure body must therefore be operations on the actual parameters. For example the values of the actual parameters may be altered by assignments within the procedure.

LOCATION PARAMETERS

The actual parameter must be a Reference i.e. a simple data reference, an array element, an indexed table identifier, a whole numbertype table-element or an anonymous reference. Index expressions are evaluated upon entry to the procedure as part of the process of obtaining the location of the actual parameter. The numeric type of the actual parameter agrees exactly with the formal specification. Part references shall not be used as location parameters.

An example of a procedure heading, and a possible call of the same procedure is

```
heading F ('VALUE''INTEGER'N;'LOCATION''INTEGER'M)
call    F (5*I+2,N[I])
```

where I is declared as 'INTEGER' and the N in the call is an 'INTEGER''ARRAY' of one dimension.

ARRAY PARAMETERS

As in an array declaration, the specified numeric type must apply to all the elements of the array named. The numeric type of the actual array name agrees with this formal specification. By indexing, within the body, the procedure may refer to any element of the actual array.

TABLE PARAMETERS

The specification of a table parameter must be identical in form to a table declaration except that presetting is not permitted. The syntax is shown in Figure 7-5:

```
Tablespec = 'TABLE' Id Tableform
Tableform = [Width,Length][Elementdeclist]
```

Figure 7-5. Table Parameters Syntax.

The element declaration list must include such fields as are used in the procedure body. Unused fields may be omitted.

PLACE PARAMETERS: LABEL PARAMETERS

The actual parameter must be a destination, i.e., a label or a switch element. In the latter case, the index is evaluated once upon entry to the procedure. The actual parameter shall be in scope at the call, even if it is out of scope where the formal parameter is used in the procedure body.

PLACE PARAMETERS: SWITCH PARAMETERS

The actual parameter must be a switch identifier. By indexing within the procedure body, the procedure is able to refer to any of the individual labels that form the elements of the switch.

PROCEDURE PARAMETERS

Within the body of a procedure, it may be necessary to execute an unknown procedure, i.e. a procedure whose name is to be supplied as an actual parameter. The features of the unknown procedure must be formally specified in the heading of the procedure within which it is called.

EXAMPLE:

Suppose that a procedure G has been declared as

```
'FLOATING''PROCEDURE'G('VALUE''INTEGER'I,J;'INTEGER''ARRAY'A);Statement
```

and further suppose that a procedure Q has a formal parameter F for which it may be required to substitute G. A declaration of Q, illustrating the necessary specification might be

```
'PROCEDURE'Q ('LABEL'B;'FLOATING''PROCEDURE'F
              ('VALUE''INTEGER','VALUE''INTEGER',
               'INTEGER''ARRAY')); Statement
```

A typical call of Q would be Q(LAB,G). At the inner level of parameter specification, no formal identifiers are required, no composite specifications are permitted (as for I and J in G) and the specifications must be separated by commas. To pursue the example to a deeper level of nesting, suppose that a procedure C66 has a parameter P for which it may be required to substitute Q. A declaration of C66 might then be

```
'PROCEDURE' C66 ('PROCEDURE'P('LABEL', 'FLOATING''PROCEDURE'),
                 'SWITCH' S); Statement
```

A typical call of C66 would be C66(Q,SW). At the level of specification shown underlined in the last example, no further parameter specifications shall be required.

The general syntax for a procedure specification is shown in Figure 7 - 6.

```
Procedurespec  =  Answerspec 'PROCEDURE' Procparamlist
Procparamlist  =  Procparameter
                  Procparameter,Procparamlist
Procparameter  =  Id
                  Id(Typelist)
Typelist       =  Type
                  Type,Typelist
Type           =  Specifier
                  'TABLE'
                  Answerspec 'PROCEDURE'
```

Figure 7 - 6 Procedure Specification Syntax.

NOTE:Procedure parameters are specified by the word 'PROCEDURE' whether or not the actual parameters to be substituted are the names of 'RECURSIVE' procedures. The name of a 'RECURSIVE' procedure may be used as an actual parameter.

NON STANDARD PARAMETER SPECIFICATION

The optional facilities described in the British Standard applicable to 'FIXED' numbers are not implemented.

THE PROCEDURE BODY

For purposes of scoping, a procedure declaration can be regarded as a block at the place where it appears in the program text. Everything except the body may be disregarded, and the formal parameters may be treated as though declared within the body, labels included. Identifiers that are non-local to the procedure body shall be those in scope at the place of the procedure declaration, subject to the restriction specified at the beginning of this Chapter. Actual parameters must be in scope at the procedure call. For example, the block

```
'BEGIN''INTEGER' I;
     'INTEGER''PROCEDURE'P;
          'ANSWER' I;
     I:=0;
     'BEGIN''INTEGER' I;
          I:=2;
          PRINT(P);
     'END'
'END'
```

has the effect of printing zero.

CHAPTER 8

COMMUNICATORS

The separately compiled units of a program may communicate with each other through 'COMMON' and with objects external to the program by means of the communicators 'EXTERNAL' and 'ABSOLUTE'. In this implementation 'LIBRARY' is used to include CORAL 66 text from a separate file and is therefore defined in Chapter 10.

'COMMON' COMMUNICATORS

Global objects declared within a program are communicated to all separately compiled units through a 'COMMON' communicator. This consists of a list of 'COMMON' items separated by semicolons all within round brackets following the word 'COMMON'. Such items shall be of three kinds, corresponding to the division of objects into data, places and procedures. A 'COMMON' data item is a declaration of the identifiers listed within it, exactly as specified in Chapter 3. Communication of places and procedures shall take the form of specification, as in the equivalent parameters of a procedure declaration (see PLACE PARAMETERS and PROCEDURE PARAMETERS). For each identifier specified in a 'COMMON' communicator there shall correspond an appropriate declaration (or for labels, a setting) in the outermost block of one and only one program unit. The syntax is as shown in Figure 8-1:

```
Commoncommunicator  =  'COMMON' (Commonitemlist)
Commonitemlist      =  Commonitem
                       Commonitem;Commonitemlist
Commonitem          =  Datadec
                       Overlaydec
                       Placespec
                       Procedurespec
                       Void
Placespec           =  'LABEL' Idlist
                       'SWITCH' Idlist
```

Figure 8-1. 'COMMON' Communicator Syntax.

The identity of each common item is determined by its overall position in the 'COMMON' communicators when all the 'COMMON' communicators present in the program unit are considered in the order in which they appear in the text. Association of its identifier with the actual object requires this position to be the same in the 'COMMON' communicators of all program units. The use of a 'LIBRARY' file containing 'COMMON' communicators by all program units is a means of ensuring that this is so.

'LIBRARY' COMMUNICATORS

See 'LIBRARY' CALLS in Chapter 10.

'EXTERNAL' COMMUNICATORS

Global objects defined in program units written in other languages must be defined through an 'EXTERNAL' communicator. Places and procedures specified in an 'EXTERNAL' communicator may also be written in CORAL 66; if present in the outermost block of a program unit they will be identified with the external object. 'EXTERNAL' data objects must not be defined in a CORAL 66 program unit; space for them must be allocated and associated with their names by other means.

The syntax for 'EXTERNAL' communicator is shown in Figure 8-2:

```
Externalcommunicator    =   'EXTERNAL'(Externalitemlist)
Externalitemlist        =   Externalitem
                            Externalitem;Externalitemlist
Externalitem            =   Unsetdatadec
                            Placespec
                            Procedurespec
                            Void
```

Figure 8-2. 'EXTERNAL' Communicator Syntax.

Names used for external objects are subject to restrictions of length and certain names of special significance to the assembler and linker may be invalid as external names. Unsetdatadec is used syntactically only in the above syntax; the communicator does not constitute a data declaration.

NOTE: External items are associated by name and there is no requirement that external items be given in the same order in each program unit, or that unused external names be included.

'ABSOLUTE' Communicators

CORAL 66 programs refer to an object having known addresses by the use of an 'ABSOLUTE' communicator which associate an identifier with a specification of the 'absolute' object, including its address. The form is similar to an external communicator with the addition of the address. The syntax is shown in Figure 8-3:

```
Absolutecommunicator    =   'ABSOLUTE'(Absoluteitemlist)
Absoluteitemlist        =   Absoluteitem
                            Absoluteitem;Absoluteitemlist
Absoluteitem            =   Absolutedatadec
                            Absoluteplacespec
                            Absoluteprocspec
                            Void
Absolutedatadec         =   Numbertype Absoluteidlist
                            Numbertype 'ARRAY' Absolutearraylist
                            'TABLE' Absoluteid Tableform
Absoluteidlist          =   Absoluteid
                            Absoluteid,Absoluteidlist
Absolutearraylist       =   Absolutearrayitem
                            Absolutearrayitem,Absolutearraylist
Absolutearrayitem       =   Absoltueidlist[Sizelist]
Absoluteid              =   Id/Absoluteaddress
Absoluteaddress         =   Integer
```

Figure 8-3. 'ABSOLUTE' Communicator Syntax.

An example of an 'ABSOLUTE' communicator is:

```
'ABSOLUTE' ('LABEL' RESTART/0;
            'BYTE''ARRAY'X/2000[10:100];
            'PROCEDURE'CO/'HEX'(F809)('VALUE''BYTE'))
```

For an array the absolute address must be the address of the lowest byte used; the array X in the last example occupies storage locations starting from address 2000 (decimal) where X[10] is stored.

CHAPTER 9

NAMES AND CONSTANTS

IDENTIFIERS

Identifiers are used for naming objects of data, labels and switches, procedures, macros and their formal parameters. An identifier consists of an arbitrary sequence of letters and digits, starting with a letter. When quotes are used to delimit language symbols any lower case letters are considered equal to and indistinguishable from the corresponding upper case letter. When language symbols are distinguished by upper case, identifiers must not contain upper case letters. An identifier carries no information in its form, e.g. single-letter identifiers are not reserved for special purposes. It may be of any length up to and including 255 characters, except that identifiers used to name external objects may be of restricted length. As layout characters are ignored, spaces may be used in identifiers without acting as terminators and without counting towards the length limit.

The syntax is shown in Figure 9-1:

```
Id = Letter
     Letter Letterdigitstring

Letterdigitstring = Letter Letterdigitstring
                    Digit Letterdigitstring
                    Void

Letter = A or B or C ... Z or a or b ... z

Digit = 0 or 1 or 2 or 3 ... 9
```

Figure 9-1. Identifier Syntax.

NUMBERS

Numerical constants specified elsewhere in this specification are of the following types:

* Constants for presetting, optionally signed;
* Integers and reals as primaries in expressions (a sign attached to a primary shall belong syntactically to the expression and not to the number);
* Integers and signed integers used in declarations or specifications, typically for defining bit-fields and array bounds;
* Integers giving addresses of 'ABSOLUTE' objects.

The syntax is shown in Figure 9-2:

```
Constant        =   Number
                    Addoperator Number

Number          =   Real
                    Integer

Signedinteger   =   Integer
                    Addoperator Integer

Real            =   Digitlist.Digitlist
                    Digitlist@Signedinteger
                    @Signedinteger
                    Digitlist.Digitlist@Signedinteger
                    'HEX'(Hexlist.Hexlist)
                    'OCTAL'(Octallist.Octallist)

Integer         =   Digitlist
                    'OCTAL'(Octallist)
                    'HEX' (Hexlist)
                    'LITERAL' (Printingcharacter)
                    #Octallist
                    #H Hexlist
                    #X Hexlist
                    #B binarylist
```

The further expansions are:

```
Digitlist       =   Digit
                    Digit Digitlist

Hexlist         =   Hexdigit
                    Hexdigit Hexlist

Octallist       =   Octaldigit
                    Octaldigit Octallist

Binarylist      =   Binarydigit
                    Binarydigit Binarylist

Hexdigit        =   Digit
                    A
                    B
                    C
                    D
                    E
                    F

Octaldigit      =   0
                    1
                    2
                    3
                    4
                    5
                    6
                    7

Binary digit    =   0
                    1
```

Figure 9-2. Number Syntax.

LITERAL CONSTANTS

A printing character shall have an integer value equal to its ASCII code with bit 7 (the most significant bit of an 8 bit value) set according to the compile-time option chosen. The integer value may be referred to within the program by the literal operator e.g.

```
'LITERAL'(a)
```

has an integer value uniquely representative of 'a'. The form is included within the syntax of Integer (see NUMBERS).

Layout characters may be specified as the argument of 'LITERAL' by one of the following special forms known as escape sequences.

Representation	Character
* B	Backspace
* C	Carriage Return
* L	Line Feed
* N	Line Feed
* P	Page Feed
* R	Rubout
* S	Space
* T	Horizontal Tab
* "	Double quote
* *	Asterisk

STRINGS

A string is any succession of up to 255 characters (printing or layout) enclosed in double quotation marks (string quotes). Double quotation marks required to be part of the string itself must be represented by an escape sequence as specified above and any of the other escape sequences shown above may be used. Further to this it is possible to interrupt the string for the purpose of spreading it over two or more lines of program text by ending a line with a single asterisk and continuing from an asterisk written on a subsequent line, provided that only layout characters appear between the closing and reopening asterisks. For example a string may be written

```
"THIS IS A VERY *
* LONG STRING"
```

A string can be classed as an unconditional expression and its value is its location but it shall not be used as a 'LOCATION' parameter.

The internal form of a string is that of a 'BYTE''ARRAY' of one dimension with lower bound zero. Element zero holds the (unsigned) length (0-255) characters and element one onwards holds the 'LITERAL' value for successive characters of the string.

Separate copies of equivalent strings are created for each occurrence of the string. For example in

```
'INTEGER' I;
I:="STRING";
'IF' I="STRING" 'THEN' Statement
```

the condition is false since the distinct locations of two copies of "STRING" are compared.

The syntax is as shown in Figre 9-3:

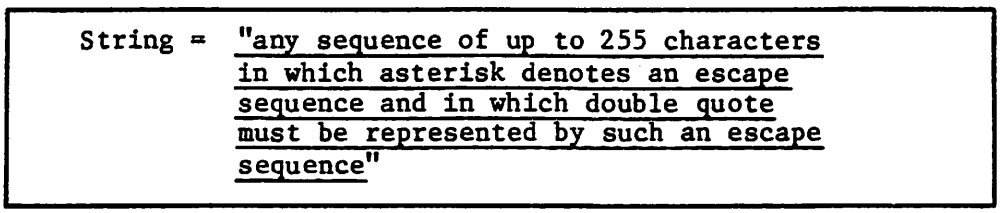

Figure 9-3. String Syntax.

CHAPTER 10

TEXT IN A PROGRAM

COMMENT

A program may be annotated by the insertion of textual matter and this comment is ignored by the compiler.

COMMENT SENTENCES

A comment sentence may be written within a program unit wherever a symbol or name can appear, except within the 'CODE' 'BEGIN' and 'END' of a code statement. A comment sentence consists of the word 'COMMENT' followed by text and terminated by a semicolon. For obvious reasons the text must not contain a semicolon. The entire comment sentence is ignored by the compiler.

BRACKETED COMMENT

Bracketed comment comprises any textual matter enclosed within round brackets immediately after a semicolon of the program or wherever a declaration or statement may appear.

The text may contain brackets provided that they are matched. Bracketed comment (including the brackets) are ignored by the compiler.

'END' COMMENT

Annotation may be inserted after the word 'END' provided that it takes the form of an identifier only. The 'identifier' is ignored by the compiler.

MACRO FACILITY

A CORAL 66 compiler embodies a macroprocessor, which may be regarded as a self-contained routine which processes the text of a CORAL 66 program before passing it to the compiler proper. Its function is to enable the programmer to define and use convenient macro names, in the form of identifiers, to stand in place of cumbersome or obscure portions of text, typically code statements. Once a macro name has been defined, the processor expands it in accordance with the definition wherever it is subsequently used, until the definition is altered or cancelled. However, the macro processor treats comments, constant character strings and the representations of numbers as indivisible entities, and does not expand any objects with the form of identifiers within these entities. No character that could form part of an identifier can be written adjacent to the use of a macro name or formal parameters of a macro, as this would inhibit the recognition of such names. A macro defintion may be written into the source program wherever a declaration or statement may appear and is removed from it by the action of the macro processor.

STRING REPLACEMENT

In the simplest use a macro name stands for a definite sequence of characters: the macro body.

EXAMPLE:

The (fictitous) code statement

```
'CODE''BEGIN' 123,45,6 'END'
```

might be given the name SHIFT6. The macro definition would be written

```
'DEFINE' SHIFT6 "'CODE''BEGIN' 123,45,6 'END'";
```

The expansion, or body, can be any sequence of up to 255 characters in which the rules of strings apply. For example, an asterisk within the body must be written as the asterisk asterisk escape sequence. Several macros may be defined with a single 'DEFINE' by writing 'DEFINE' MACROA "macroabody", MACROB "macrobbody", ...;

PARAMETERS OF MACROS

A macro may have parameters, as in the following example:

```
'DEFINE' SHIFT (N) "'CODE''BEGIN' 123,45,N 'END'";
```

Subsequent occurrences of SHIFT(6) would be expanded to the code statement in the STRING REPLACEMENT example. A formal parameter, such as N above, is written as an identifier. An actual parameter (e.g. 6) is any string of characters in which string quotes are matched, all round and square brackets are nested and matched, and all occurences of a comma lie between round or square brackets. This requirement enables commas to be used for separating actual parameters. The number of actual parameters must be the same as the number of formals, which must also be separated by commas.

NESTING OF MACROS

A macro definition may embody definitions or uses of other macros. When a macro is defined the body is kept but not expanded. When the macro is used it is as though the body were substituted into the program text, and it is during this substitution that any other macros encountered are processed. The use of a macro with parameters can be regarded as introducing virtual macro definitions for the formal parameters before the macro body is substituted. Thus, to continue from the previous example, the occurence of SHIFT(6) would be equivalent to

```
'DEFINE' N "6";
'CODE''BEGIN' 123,45,N 'END'
```

followed immediately by deletion of the virtual macro N.

Throughout the scope of the macro SHIFT, the formal parameter N must not be defined as a macro name. A formal parameter cannot be used in any inner nested macro definition; neither in its body nor as a macro name nor as a formal parameter. Furthermore no identifier in an actual parameter string, or its subsequent expansions, can be the same as any formal parameter of the calling macro.

DELETION AND REDEFINITION OF MACROS

The scope of a macro definition is from the point of definition until either the end of the program text is reached or the macro name is redefined or deleted. The scope of a macro is independent of the block structure of the program. To delete a macro the construct

```
'DELETE' Macroname;
```

is given wherever the requirements of this standard allow a declaration or statement to appear.

To delete more than one macro a list of macro names separated by commas may be given instead of Macroname.

The 'DELETE' construct is removed by the action of the macro processor. Alternatively, a macro name may be redefined. Macro definitions that have the same name are stacked and so processed that the most recent definition is deleted, and the previous one reinstated.

NOTE: 'Recent' and 'previous' refer to the sequence as processed by the Macro processor.

SYNTAX OF COMMENT AND MACROS

The syntax is shown in Figure 10-1:

```
Commentsentence          =  'COMMENT' any sequence of
                            characters not including a
                            semicolon;
Bracketedcomment         =  (any sequence of characters in
                            which round brackets are matched)
Endcomment               =  Id
Macrodefinition          =  'DEFINE' Macrodeflist;
Macrodeflist             =  Macrodef
                         =  Macrodef,Macrodeflist
Macrodef                 =  Macroname "Macrobody"
                            Macroname(Idlist)"Macrobody"
Macrobody                =  String
Macrodeletion            =  'DELETE' Macronamelist
Macronamelist            =  Macroname
                         =  Macroname,Macronamelist
Macroname                =  Id
Macrocall                =  Macroname
                            Macroname(Macrostringlist)
Macrostringlist          =  Macrostring
                            Macrostring,Macrostringlist
Macrostring              =  any sequence of up to 255
                            characters in which commas
                            are protected by round or square
                            brackets and in which such
                            brackets are properly matched
                            and nested
```

Figure 10-1. Syntax of Comment and Macros.

'LIBRARY' CALLS

A library call is valid as a symbol at any point in the CORAL 66 text of a program. The syntax is shown in Figure 10-2:

```
Librarycall = 'LIBRARY'"file specification"
```

Figure 10-2. Library Call Syntax

where file specification is operating system dependent and is further specified in the appropriate CORAL 66 Operating Guide.

The effect of a library call is that the entire contents of the file specified are inserted into the text at the point of the library call.

It is possible to nest 'LIBRARY' calls but the depth of such nesting may be limited by the input-output system.

A file inserted by a library call contains a whole number of CORAL 66 symbols; for example it is not possible to use a library file to provide the leading characters of an identifier and append further characters in the main text.

APPENDIX A

SYNTAX RULES

Some recourse to plain English occurs in the syntax rules and is underlined to avoid any possible confusion with formal class names and terminal symbols.

Rule			Figure Reference
Absoluteaddress	=	Integer	8-3
Absolutearrayitem	=	Absoluteidlist [Sizelist]	8-3
Absolutearraylist	=	Absolutearrayitem Absolutearrayitem, Absolutearraylist	8-3
Absolutecommunicator	=	'ABSOLUTE'(Absoluteitemlist)	8-3
Absoluteitemlist	=	Absoluteitem Absoluteitem;Absoluteitemlist	8-3
Absoluteitem	=	Absolutedatadec Absoluteplacespec Absoluteprocspec Void	8-3
Absolutedatadec	=	Numbertype Absoluteidlist Numbertype 'ARRAY' Absolutearraylist 'TABLE' Absoluteid Tableform	8-3
Absoluteid	=	Id/Absoluteaddress	8-3
Absoluteidlist	=	Absoluteid Absoluteid, Absoluteidlist	8-3
Actual	=	Expression Reference Destination Name	6-4
Actuallist	=	Actual Actual,Actuallist	6-4
Addoperator	=	+ -	5-2
Alternative	=	Statement	6-9
Answerspec	=	Numbertype Void	7-2

Answerstatement	=	'ANSWER' Expression	6-1
Arraydec	=	Unsetarraydec Presetlist	3-2
Arrayitem	=	Idlist [Sizelist]	3-2
Arraylist	=	Arrayitem Arrayitem,Arraylist	3-2
Assignmentstatement	=	Variable Assignmentsymbol Expression	6-2
Assignmentsymbol	=	:=	6-2
Base	=	Id Id [Signedinteger]	3-8
Binarydigit	=	0 1	9-2
Binarylist	=	Binarydigit Binarydigit Binarylist	9-2
Bitposition	=	Integer	3-4
Bitseparator	=	'BIT' ,	3-4
Block	=	'BEGIN' Declist;Statementlist 'END'	2-1
Booleanword	=	Booleanword2 Booleanword4 'DIFFER' Booleanword 5	5-3
Booleanword2	=	Booleanword3 Booleanword5 'UNION' Booleanword6	5-3
Booleanword3	=	Booleanword6 'MASK' Shiftexpression	5-3
Booleanword4	=	Booleanword Shiftexpression	5-3
Booleanword5	=	Booleanword2 Shiftexpression	5-3
Booleanword6	=	Booleanword3 Shiftexpression	5-3
Bracketedcomment	=	(any sequence of characters in which round brackets are matched)	10-1
Byteposition	=	Signedinteger	3-5
Codesequence	=	assembler instructions in which %name refers to CORAL defined name	6-6

Codestatement	=	'CODE''BEGIN' Codesequence 'END'	6-6
Commentsentence	=	'COMMENT' any sequence of characters not including semicolon;	10-1
Commoncommunicator	=	'COMMON' (Commonitemlist)	8-1
Commonitem	=	Datadec Overlaydec Placespec Procedurespec Void	8-1
Commonitemlist	=	Commonitem Commonitem;Commonitemlist	8-1
Communicator	=	Commoncommunicator Absolutecommunicator Externalcommunicator Void	8-1
Communicatorlist	=	Communicator Communicator;Communicatorlist	8-1
Comparator	=	< <= > >= < >	5-10
Comparison	=	Expression Comparator Expression	5-10
Compoundstatement	=	'BEGIN' Statementlist 'END'	6-1
Condition	=	Condition 'OR' Subcondition Subcondition	5-10
Conditionalexpression	=	'IF' Condition 'THEN' Expression 'ELSE' Expression	5-9
Conditionalstatement	=	'IF' Condition 'THEN' Consequence	6-1
Conditionalstatement2	=	'IF' Condition 'THEN' Consequence2 'ELSE' Alternative	6-1
Consequence	=	Statement	6-9
Consequence2	=	Simplestatement Label:Consequence2	6-9
Constant	=	Number Addoperator Number	9-2

Constantlist	=	Group Group, Constantlist	3-2
Datadec	=	Numberdec Arraydec Tabledec	2-1
Dec	=	Datadec Overlaydec Switchdec Proceduredec	2-1
Declist	=	Dec Dec; Declist	2-1
Destination	=	Label Switch[Index]	6-4
Digit	=	0 or 1 or 2 or 3 ... 9	9-1
Digitlist	=	Digit Digit Digitlist	9-2
Dimension	=	Lowerbound:Upperbound	3-2
Dummystatement	=	Void	6-1
Elementdec	=	Id Numbertype Byteposition Id Partwordtype Byteposition Bitseparator Bitposition	3-3
Elementdeclist	=	Elementdec Elementdec; Elementdeclist	
Elementpresetlist	=	'PRESET' Constantlist Void	3-3
Elementscale	=	(Totalbits)	3-6
Endcomment	=	Id	10-1
Expression	=	Unconditionalexpression Conditionalexpression	5-1
Externalcommunicator	=	'EXTERNAL' (Externalitemlist)	8-2
Externalitem	=	Unsetdatadec Placespec Procedurespec Void	8-2
Externalitemlist	=	Externalitem Externalitem;Externalitemlist	8-2
Factor	=	Primary Booleanword	5-3

Forelement	=	Expression Expression 'WHILE' Condition Expression 'STEP' Expression 'UNTIL' Expression	6-10
Forlist	=	Forelement Forelement,Forlist	6-10
Forstatement	=	'FOR' Reference Assignmentsymbol Forlist 'DO' Statement	6-10
Gotostatement	=	'GOTO' Destination	6-3
Group	=	Constantlist (Constantlist) Void	3-7
Hexdigit	=	Digit A B C D E F	9-2
Hexlist	=	Hexdigit Hexdigit Hexlist	9-2
Id	=	Letter Letterdigitstring	9-2
Idlist	=	Id Id, Idlist	3-1
Index	=	Expression	5-6
Integer	=	Digitlist 'OCTAL' (Octallist) 'HEX' (Hexlist) 'LITERAL' (printingcharacter) #Octallist #H Hexlist #B Binarylist #X Hexlist	9-2
Label	=	Id	4-1
Labellist	=	Label Label,Labellist	4-1
Length	=	Integer	3-3
Letter	=	A or B or C ... Z or a or b ... z	9-2

Letterdigitstring	=	Letter Letterdigitstring Digit Letterdigitstring Void	9-1
Librarycall	=	'LIBRARY' "file specification"	10-2
Lowerbound	=	Signedinteger	3-2
Macrobody	=	String	10-1
Macrocall	=	Macroname Macroname (Macrostringlist)	10-1 10-1
Macrodefintion	=	'DEFINE' Macrodeflist	10-1
Macrodeflist	=	Macrodef Macrodef, Macrodeflist	10-1
Macrodef	=	Macroname Macrobody Macroname (Idlist) Macrobody	10-1
Macrodeletion	=	'DELETE' Macronamelist	10-1
Macroname	=	Id	10-1
Macronamelist	=	Macroname Macroname, Macronamelist	10-1
Macrostring	=	any sequence of up to 255 characters in which commas are protected by round or square brackets and in which such brackets are properly matched and nested	10-1
Macrostringlist	=	Macrostring Macrostring, Macrostringlist	10-1
Multoperator	=	* / 'MOD'	5-2
Name	=	Id	6-4
Number	=	Real Integer	9-2
Numberdec	=	Unsetnumberdec Presetlist	3-1
Numbertype	=	'FLOATING' 'INTEGER' 'BYTE'	3-1

Octaldigit	=	0 1 2 3 4 5 6 7	9-2
Octallist	=	Octaldigit Octaldigit Octallist	9-2
Overlaydec	=	'OVERLAY' Base 'WITH' Unsetdatadec	3-9
Parameterspec	=	Specifier Idlist Tablespec Procedurespec	7-4
Parameterspeclist	=	Parameterspec Parameterspec; Parameterspeclist	7-3
Partprimary	=	Id[Index] 'BITS' [Totalbits, Bitposition] Typedprimary	5-5
Partreference	=	Id[Index] 'BITS' [Totalbits, Bitposition] Reference	6-2
Partwordtype	=	Elementscale 'UNSIGNED' Elementscale	3-4
Placespec	=	'LABEL' Idlist 'SWITCH' Idlist	8-1
Presetlist	=	Constantlist Void	3-7
Primary	=	Untypedprimary Typedprimary	5-3
Procedurecall	=	Id Id(Actuallist)	6-4
Proceduredec	=	Answerspec 'PROCEDURE' Procedureheading; Statement Answerspec 'RECURSIVE' Procedureheading; Statement	7-1
Procedureheading	=	Id Id (parameterspeclist)	7-3
Procedurespec	=	Answerspec 'PROCEDURE' Procparamlist	7-1
Procparameter	=	Id Id (Typelist)	7-6

Procparamlist	=	Procparameter Procparameter, Procparamlist	7-6
Programunit	=	'CORAL' Id Communicatorlist Block 'FINISH'	1-1
Real	=	Digitlist.Digitlist Digitlist@Signedinteger Digitlist.Digitlist@Signedinteger 'HEX'(Hexlist.Hexlist) 'OCTAL'(Octallist.Octallist)	9-2
Reference	=	Id Id [Index] Id [Index,Index] [Index]	5-6
Returnstatement	=	'RETURN'	6-11
Shiftexpression	=	Primary Shiftexpression Shiftop Primary	5-8
Shiftop	=	'SLL' 'SRL'	5-8
Signedinteger	=	Integer Addoperator Integer	9-2
Simpleexpression	=	Term Addoperator Term Simpleexpression Addoperator Term	5-2
Simplestatement	=	Assignmentstatement Gotostatement Procedurecall Answerstatement Codestatement Compoundstatement Block Dummystatement Forstatement Returnstatement Conditionalstatement2	6-1
Sizelist	=	Dimension Dimension,Dimension	3-2
Specifier	=	'VALUE' Numbertype 'LOCATION' Numbertype Numbertype 'ARRAY' 'LABEL' 'SWITCH'	7-4

Statement	=	Label:Statement Simplestatement Conditionalstatement	6-1
Statementlist	=	Statement Statement; Statementlist	6-7
String	=	"any sequence of zero to 255 characters in which asterisk denotes an escape sequence and in which double quote must be represented by such an escape sequence"	9-3 and 10-1
Subcondition	=	Subcondition 'AND' Comparison Comparison	5-10
Switch	=	Id	4-1
Switchdec	=	'SWITCH' Switch Assignmentsymbol Labellist	4-1
Tabledec	=	'TABLE' Id Tableform Presetlist 'TABLE' Id [Width,Length] [Elementdeclist Elementpresetlist]	3-3
Tableform	=	[Width,Length][Elementdeclist]	3-3
Tablespec	=	'TABLE' Id Tableform	7-5
Term	=	Factor Term Multoperator Factor	5-2
Totalbits	=	Integer	5-7
Type	=	Specifier 'TABLE' Answerspec 'PROCEDURE'	7-6
Typedprimary	=	Reference Partprimary 'LOCATION'(Reference) Numbertype(Expression) Procedurecall Integer	5-5
Typelist	=	Type Type,Typelist	7-6
Unconditionalexpression	=	Simpleexpression String	5-1

Unsetdatadec	=	Unsetnumberdec Unsetarraydec Tablespec	2-1
Unsetnumberdec	=	Numbertype Idlist	3-1
Unsetarraydec	=	Numbertype 'ARRAY' Arraylist	3-2
Untypedprimary	=	Real (Expression)	5-4
Upperbound	=	Signedinteger	3-2
Variable	=	Reference Partreference	6-2
Width	=	Integer	3-3

APPENDIX B

PROCEDURE PARAMETERS

Object	Formal Specification	Actual Parameter
Numerical value	'VALUE' Numbertype Id[1]	Expression
Location of data	'LOCATION' Numbertype Id[1]	Reference
Name of array	Numbertype 'ARRAY' Id	Id
Name of table	Tablespec[2]	Id
Place in program	'LABLE' Id[1]	Destination
Name of switch	'SWITCH' Id[1]	Id
Name of procedure	Procedurespec[3]	Id

1 - Composite specification has Idlist in place of Id
2 - See TABLE PARAMETERS in Chapter 7
3 - See PROCEDURE PARAMETERS in Chapter 7

APPENDIX C

LANGUAGE SYMBOLS

Language Symbol	Figure	Language Symbol	Figure
'ABSOLUTE'	8-3	'LABEL'	7-4 and 8-1
'AND'	5-10	'LIBRARY'	10-2
'ANSWER'	6-1	'LITERAL'	9-2
'ARRAY'	3-2 and 7-4	'LOCATION'	5-5 and 7-4
'BEGIN'	2-1, 6-1 and 6-6	'MASK'	5-3
'BIT'	3-4	'MOD'	5-2
'BITS'	5-6 and 6-2	'OCTAL'	9-2
'BYTE'	3-1	'OR'	5-10
'CODE'	6-6	'OVERLAY'	3-9
'COMMENT'	10-1	'PRESET'	3-3
'COMMON'	8-1	'PROCEDURE'	7-1
'CORAL'	1-1	'RECURSIVE'	7-1
'DEFINE'	10-1	'RETURN'	6-11
'DELETE'	10-1	'SLL'	5-8
'DIFFER'	5-3	'SRL'	5-8
'DO'	6-10	'STEP'	6-10
'ELSE'	5-9 and 6-1	'SWITCH'	4-1, 7-4 and 8-1
'END'	2-1 and 6-6	'TABLE'	3-3, 7-5 and 7-6
'EXTERNAL'	8-2	'THEN'	5-9 and 6-1
'FINISH'	1-1	'UNION'	5-3
'FLOATING'	3-1	'UNSIGNED'	3-6
'FOR'	6-10	'UNTIL'	6-10
'GOTO'	6-3	'VALUE'	7-4
'HEX'	9-2	'WHILE'	6-10
'IF'	5-9 and 6-1	'WITH'	3-9
'INTEGER'	3-1		

APPENDIX D

CHARACTER SET

Character	Description and Chapter Reference
0 1 2 3 4 5 6 7 8 9	digits, 9
a ... z A ... Z	letters, 9
+ -	adding operators, 5
* /	multiplying operators, 5 * in escape sequences, 9
< <= = >= > < >	comparators, 5
( )	expression brackets, 5 bracketed comments, 10
[ ]	index brackets, 3 and 4 table characters, 3
"	string quotes, 9 and 10
, ;	comma semicolon,separators for lists
:	colon separator for bounds, 3 terminator for label setting, 6
:= <-	assignment symbol, 3, 4 and 6
.	point, 9
@	'times ten to the power of', 9
#	binary octal and hexadecimal numbers, 9
%	trip character in code statements, 6
'	Language symbol quotes, 1

APPENDIX E

CORAL 66 CONSTRAINTS

1. Variable and procedure names may be of any length, but only the first 255 alphanumeric characters are significant.

2. The maximum size of a macro definition is 255 characters.

3. The maximum number of arguments any procedure can have is 255.

4. The maximum size of arguments a procedure can have is 255 bytes, and a recursive procedure 245.

5. The maximum size of arguments and local variables a recursive procedure can have is 255.

6. The maximum number of procedures that can be declared in any one segment is 100. Common and Absolute definitions of procedures are excluded from this figure.

7. Two dimensional arrays in ABSOLUTE and in EXTERNAL are allowed, but not supported.

APPENDIX F

SUMMARY OF IMPLEMENTATION SPECIFIC FEATURES

MAJOR FEATURES DEFINED AS OPTIONAL BY BRITISH STANDARD

'RECURSIVE' procedures

'RECURSIVE' procedures are implemented with the one restriction described in 3.7.1(b).

'TABLE' facilities

'TABLE' is implemented except that partword table elements may not contain fractional parts.

'FIXED' and 'FLOATING' point numbers

'FIXED' point numbers are not implemented. 'FLOATING' point numbers are implemented with some user choice of format and size.

> NOTE: The 8080 compiler is optionally supplied without 'FLOATING' point, and that 'FLOATING' point may also be suppressed by a compile-time option.

'OVERLAY' of data

'OVERLAY' is implemented except that no presetting is allowed in an overlay declaration.

'BITS'

'BITS' is implemented, and extended to operate on 'BYTE' data.

Set of dyadic logical operators

'DIFFER', 'UNION' and 'MASK'

These operators are implemented and are extended to operate on 'BYTE' data.

The communicators 'COMMON', 'LIBRARY', 'EXTERNAL' and 'ABSOLUTE'

'COMMON', 'ABSOLUTE' and 'EXTERNAL' are implemented for communication between CORAL modules, and between CORAL modules and objects defined in other languages.

'LIBRARY' is used for source statement inclusion.

'CODE' Statements

'CODE' statements may not possess a value in the way suggested by 6.6.6 of the British Standard. 'CODE' statements consist of assembler source lines for the appropriate computer and CORAL defined objects and constants may be referred to by %name or %number.

NON STANDARD PROCEDURE PARAMETER FEATURES

These facilities, described in 6.7.4.10 of the British Standard, applicable to 'FIXED' numbers, are not implemented.

EXTENSIONS TO STANDARD 'CORAL'

All of the following represent extensions to the British Standard:

* The 'RETURN' statement. (Chapter 6)
* The number type 'BYTE'. (Chapter 3)
* The operators 'SRL' and 'SLL'. (Chapter 5)
* The operator 'MOD'. (Chapter 5)
* Expressions compared in conditions may be themselves conditional expressions. (Chapter 5)
* Procedures may call other procedures declared at the same block irrespective of their order of declaration and irrespective of their inclusion in communicators. (Chapter 7)
* Data in inner blocks of a program, inner blocks of a procedure body and within the body of a 'RECURSIVE' procedure may be preset. (Chaper 3)
* Integer numbers may be written in four additional notations to those specified in the British Standard. (Chapter 9)
* Facilities for including layout characters in literal constants and strings. (Chapter 9)
* The rules for comment sentences are more liberal than the standard, comment sentences being allowed at any symbol or name in the program text. The rules for bracketed comments are also slightly more liberal than the standard. (Chapter 10)
* A list of macros may be defined or deleted in a single 'DEFINE' or 'DELETE' (Chapter 10)

ADDRESSING

Addressing is appropriate to byte-addressed computers. Consecutive 'BYTE' data have addresses in ascending order differing by one while consecutive 'INTEGER' data have addresses in ascending order differing by two. Values taken by the 'LOCATION' operator and 'LOCATION' parameters are machine addresses.