CAP 9200

CORAL

Book No. 309

Copy No. 10

Amendment No. 1

Marconi Aircraft Systems Division

This Book forms part of the MASD Software Library.

If there is a GREEN MASD LIBRARY stamp on this page, then this is a Recorded Copy of the Master Book, for which an UPDATING service is available; so it could be up-to-date.

If there is NO stamp or only a XEROXED stamp on this page, then this copy has NO updating service; and the reader uses it at his PERIL.

Marconi-Elliott Avionic Systems Limited
Airport Works, Rochester, Kent. ME1 2XX

A GEC-Marconi Electronics Company

Telephone: Medway (0634) 44400
Telegrams: Elliotauto Rochester
Telex: 96333/4

Compiled from Various Sources;
Issued by T.J.Froggatt.

Terry Froggatt 16/2/76

PREFACE.

This book describes the following 10 tapes:-

920C CORAL MACRO PASS,	CAP BOREHAMWOOD VERSION 3B;
920C CORAL PASS 1A,	CAP BOREHAMWOOD VERSION 3B;
920C CORAL PASS 1B,	CAP BOREHAMWOOD VERSION 3B;
920C CORAL PASS 2,	CAP BOREHAMWOOD VERSION 3B;
920C CORAL LOADER,	CAP VERSION 3, SEPTEMBER 1974;
920C CORAL DATA RETENSION,	CAP VERSION 2, SEPTEMBER 1974;
920C CORAL OBJECT DUMP,	CAP VERSION 2, SEPTEMBER 1974;
CAPQF, TJF VERSION 16/2/76,	R.L.B. Mode 3;
CORAL 8K EXTENDED LOADER,	Binary Mode 3;
CORAL 16K EXTENDED LOADER,	Binary Mode 3.

These tapes form a CORAL Compiling System for 900-Series 18-Bit Machines. Although written initially for a 920C their use is not confined to this machine.

The CORAL Language is defined and described in Book 302, CORAL INFORMATION.

This book is in two sections. The first and larger section, "CORAL COMPILER USERS' MANUAL" relates to the first 8 tapes. The second and smaller section, "EXTENDED LOADER", describes the changes to the "USERS' MANUAL" when one of the "EXTENDED LOADER" tapes is used rather than the "VERSION 3" loader tape.

The CORAL Compiling System's minimum compile-time requirements are a 920C or 905 computer (most passes will not run on a 920A, 920B or 903, or 920M computer) with a punch, reader, and preferably a teletype. Pages 2 & 64 of the Users' Manual herein state that 16K of store is needed; but for all except "practice" programs, 24K is needed (and the directives on page 70 will have to be used).

The system as described herein is a 5-pass paper-tape oriented system. 905 users with a Disk and at least 32K of store, available at compile time, are recommended to use the SODAR system described in Book 310, 905 SODAR.

The CORAL Compiling System (whether run under SODAR or not), can be used to produce object code to run on any 900-Series 18-bit machine except a 920A (including: 920B or 903, 920M, 920C or 905), and of any size (from 8K to 128K), with just a tape reader for program loading.

For efficiency the compiler uses the "absolute addressing" strategy, and the primary consequences of this are the following limitations on the user's programs:-

The data of the whole program must normally be placed in the first 8K of store; although non-preset data can be placed anywhere in store if accessed "anonymously".

On a 920B or 903, or a 920M, the program code itself must also all be placed in the first 8K of store, so the only use of store beyond 8K is for anonymous data.

It will be seen that 3 alternative loaders are available. All 3 can produce a Binary tape of the Object program. The advantages of the "VERSION 3" loader are:-

The Binary tape produced by it, for a given program, is shorter than that produced by the extended loaders.

The Binary tape is in the standard "A.C.D. 900-Series 18-Bit Binary Tape Format, 1/4/70", whereas that produced by the extended loaders is not.

The Version 3 loader may be used in "load-and-go" mode, unlike the extended loaders (which don't, in fact, load anything). This reduces the system from "5 passes + Binary Loading" to "5 passes including loading".

However the Version 3 loader restricts the user's program to locations 556 to 8165 and 8192 to 14707; limiting the total data space (other than anonymous reference) to 7609 locations, from 8157; and the total data+program space to 14124 locations, from 128K.

The advantages of the "EXTENDED LOADERS" are that:-

- These restrictions are removed.

The Extended Loader can be run on an 8K machine, (which may be a 920B or 903, 920M, or 920C or 905), whereas the Version 3 loader always needs a 16K store.

There are some compatability problems using these alternative loaders. The differences in the Binary tape lengths and formats imply that tapes made by the two routes, of the same program, will be different; and I would imagine that the Object codes, when loaded into core, will not be identical, either.

The implementation of the Multi-level environment, and of the Run-time sumcheck, in the Version 3 and extended loaders, are totally different: in a program using either of these facilities a change from the Version 3 loader to an extended loader, or back, would require a change to the CORAL source.

The standard "A.C.D. 900-Series 18-Bit Binary Tape Format, 1/4/70", produced by the version 3 loader, and referred to on page 77 of the User's Manual herein, is defined in Book 106, 903/905/920 USEFUL NOTES.

This CORAL Compiling System operates in ISO/ASCII Teleprinter Code, as referred to on page 59 of the User's Manual herein. In standard 900-Series terminology this is "903 Telecode". There is no CORAL compiler for the 900-Series which operates in the alternate "920 Telecode". 903 and 920 Telecodes are both defined in Book 106, 903/905/920 USEFUL NOTES.

The original "CAP 920C CORAL" was written, and updated in parts twice, by CAP (Reading) Ltd., for the Royal Aircraft Establishment at Farnborough. A further partial update was produced by CAP for RL, MEASL, Borehamwood. This sequence of events produced the first 7 of the tapes listed above, which are commonly known as "Issue 3Bii or 4" (!). The extended loader was written by CAP for MASD, MEASL, at Rochester: the 2 versions listed above are 8K & 16K versions of CAP's issue, modified to make maximum use of the first 8K of store and with one error corrected.

These programs are known to contain numerous errors, but as more are still being discovered I have seen little point in recording even those already known in a document updated as infrequently as I intend this to be. I shall endevor to keep all serious users of the compiling system informed of all errors discovered, and likewise would appreciate being informed of all errors found in the system, as I have issued it, with supporting documentation. I see little chance of these errors being corrected; particularly in view of the number of organisations involved.

The "CAPQF" tape supplied in this package is a complete re-write; it is an all-round improvement on the original issued by CAP and based on 903 QF; whilst maintaining full 900-Series compatability.

920C CORAL MACRO PASS, CAP BOREHAMWOOD VERSION 3B;

920C CORAL PASS 1A, CAP BOREHAMWOOD VERSION 3B;

920C CORAL PASS 1B, CAP BOREHAMWOOD VERSION 3B;

920C CORAL PASS 2, CAP BOREHAMWOOD VERSION 3B;

920C CORAL LOADER, CAP VERSION 3, SEPTEMBER 1974;

920C CORAL DATA RETENSION, CAP VERSION 2, SEPTEMBER 1974;

920C CORAL OBJECT DUMP, CAP VERSION 2, SEPTEMBER 1974;

CAPQF, TJF VERSION 16/2/76, R.I.B. Mode 3.

ACKNOWLEDGEMENTS

This manual provides user information on the CORAL 66 Compiler for the Elliott 920C computer written by Computer Analysts and Programmers Ltd.

We wish to acknowledge the support and advice which has been received during the production of the 920C CORAL Compiler from the Royal Aircraft Establishment (Farnborough), the Royal Radar Establishment (Malvern) and Marconi-Elliott Avionics Systems Limited (Rochester).

INTRODUCTION

A knowledge of the Official Definition of CORAL 66 (to which references are written QD a.b.c...) and the 920C order code is assumed.

The language implemented is full CORAL 66 excluding recursion and including the additional features of partword arrays and shift operators together with a method of producing multi-level object code.

The minimum configuration for compilation of a CORAL 66 program is an Elliott 920C(905) with 16K of core store, a paper tape reader, a paper tape punch and a teleprinter (if less than 8K of object code is produced this may be executed on a 920B (903) upwards compatible computer).

The following description summarises the contents of each chapter:

Chapter 1: A description of the implementation dependent features of the language together with an expansion of the relevant sections of the Official Definition of CORAL 66. (The full CORAL syntax is summarised in Appendix A).

A description of the additional features provided.

A list of all CORAL language symbols and external character codes.

Chapter 2: A description of the constituents of the 920C CORAL Compiling System.

A note on the object code compatibility for alternative machine configurations.

A description of the purpose, mode of operation and options provided by each program supplied.

A description of the general method of interface with the user.

Chapter 3: A description of the operating instructions for each program of the 920C CORAL Compiling System together with a summary of the options provided.

Chapter 4: A description of the error diagnostic messages produced by each compiler program together with any further diagnostic information produced to aid the user with program development.

Chapter 5: This chapter can normally be ignored since all necessary user information is provided by Chapter 1. However, if the structure of the object code is particularly important to the user it is hoped that the required information is provided.

A general description of the structure and runtime storage of the object code.

Notes on the optimisations performed by the Compiler and methods of producing efficient object code.

A description of the interrupt handling housekeeping code which can be generated by the Compiler.

Note: In order to simplify cross-referencing, chapters 2, 3 and 4, which provide a description, operating instructions and diagnostic information respectively, are identical in structure for each component of the 920C CORAL Compiling System.

The following representations are used within the manual:

↵ carriage return

∇ null

CONTENTS

A detailed index is provided at the head of each chapter.

CHAPTER 1

CORAL LANGUAGE DEFINITION

1.1 CLARIFICATION OF THE OFFICIAL DEFINITION

1 CORAL LANGUAGE DEFINITION

The general classification of the language implemented is full CORAL 66 excluding recursion, i.e. including:

- table handling
- bit manipulation
- data overlaying
- floating point arithmetic

plus the extra facilities:

- bit and byte arrays
- left and right shifts

The following description assumes a knowledge of the Official Definition of CORAL 66 (HMSO).

Section 1.1 describes the implementation dependent features of the language which includes an expansion of areas of the Official Definition where necessary.

Section 1.2 describes the extensions to the language provided by the 920C Compiler.

Section 1.3 defines the 920C CORAL language symbols and character codes.

Throughout the following description the bit numbering for a computer word is as in the Official Definition, i.e. least significant bit being bit 0 and the most significant bit being bit 17.

1.1 CLARIFICATION OF THE OFFICIAL DEFINITION OF CORAL 66

The following description of language features is simply a set of notes and should be read in conjunction with the Official Definition from which there is no deviation or addition unless otherwise stated.

The following points appear approximately in the order of Official Definition.

1.1.1 UNITS OF COMPILATION AND COMMUNICATORS (OD 2.2 and 9.1)

1.1.1.1 Units of Compilation

The 920C Compiler allows four distinct units of compilation, thereby allowing separate compilation of individual sections of a program which are link loaded by the CORAL Loader prior to execution.

A unit of compilation is structured:

'CORAL'

unit - see below

'FINISH'

and the paper tape must terminate with a halt code.

The unit of compilation can be arbitrarily split into several paper tapes where each tape except the last(which terminates with the 'FINISH' keyword) must terminate with a 'HALT' keyword - all tapes must have a haltcode as the last character following the 'HALT' or 'FINISH'.

The possible units of compilation are as follows:

1.1.1.1.1 A single program segment

An outermost block of a program may be compiled as a separate unit for link loading with the remainder of the program. Reference to Library procedures within the segment is indicated by a Library communicator (1.1.1.2.2) at the head of the segment. Communication with the outermost blocks of independently compiled segments is indicated by a common communicator (1.1.1.2.1) at the head of the segment.

The format of a single segment unit of compilation is:

```
'CORAL'
'PROGRAM' programname
Library communicator; (optional)
Common communicator;  (optional)
'SEGMENT' segment name
Outermost block constituting segment body
'FINISH'
```

1.1.1.1.1 (cont.) See also the example in Appendix E.

1.1.1.1.2 A set of program segments

A set of program segments, which may or may not comprise a complete program, may similarly be compiled as a separate unit. As above, reference to Library procedures is indicated by a Library communicator and communication with other independently compiled segments or with different segments in this unit is indicated by a Common communicator.

The format of a set of segments unit of compilation is:

```
'CORAL'
'PROGRAM' programname
Library communicator;(optional)
Common communicator; (optional)
'SEGMENT' seglname
Outermost block constituting segment body;
    .
    .
    .
    .
    .
'SEGMENT' segNname
Outermost block constituting segment body
'FINISH'
```

1.1.1.1.3 A Common segment

This merely contains the information contained in the Common communicator of a program and therefore compilation as a separate unit serves no real purpose since it does not contain any executable code. A possible use could be to determine the runtime size of the Common area of a program and the positions of items using the Object Map facility (4.1.4.2) as it may be loaded by itself.

1.1.1.1.3 (cont.) The format of the Common segment unit of compilation is:

'CORAL'
Common communicator
'FINISH'

1.1.1.1.4 A set of Library procedures

Library procedures are compiled as a special unit of compilation to be referenced from CORAL programs thus only requiring each Library procedure to be compiled once and not each time it is used. No Common communicator is allowed (communication is via the parameters of the procedures only) and the source is in the form of a number of procedure declarations not blocks. Several Library procedures may be compiled within one unit of compilation.

The format of a set of Library procedures unit of compilation is:

'CORAL'
'LIBRARY' Libraryname
Library communicator; (optional)
Procedure declaration of library procedure;
.
.
.
.
Procedure declaration of library procedure
'FINISH'

The inclusion of a Library communicator is necessary if reference is made from within this unit to any Library procedure contained in a different unit of compilation.

The declaration of a library procedure is identical to that of any other procedure as described in OD 8 except that the procedure name is of the form

name/no.

where the no. is any number between 10 and 2000 allocated by the user for identification of the

1.1.1.1.4 (cont.) procedure. The name alone is used to call the procedure within a program thereby allowing the user to have several versions of each Library procedure, as each is updated, and to link load the required version (the no. and not the name is used by the 920C CORAL Loader for linking purposes).

1.1.1.2 Communicators

As mentioned above there are two types of communicators.

1.1.1.2.1 Common communicators

This allows communication between the outermost blocks of a program and between separately compiled segments whereby objects can be global and accessible to each outermost block. The structure and contents of the Common communicator within individual units of compilation of a CORAL program must be identical.

The format of a Common communicator is:

'COMMON' Commonname
(Commonitemlist)

The Commonitemlist is as defined in OD 9.1.

The items in a common communicator are of two types, declarations and specifications. The first category includes all types of data, and the second includes all 'places' that is procedures, switches and labels.

1.1.1.2.1.1 Common Declarations

Items declared in common have the same semantic status in all segments as they would have if they had been declared in the outermost block of those segments, but the data space to which they refer is allocated in the common data area. Thus it is unnecessary to declare such items within a specific segment.

1.1.1.2.1.2. Common Specifications

A common specification contains only enough information about an item to enable it to be referenced correctly in all segments. It does not create that item, and it is necessary that a full declaration occur in the outermost block of one, and only one, segment of the program. The use of procedure and label specifications is adequately described in OD 8.3.3., OD 8.3.4., and OD 9.1 The use of common switches is described below.

1.1.1.2.1.1.1. Common Switches

A common switch item is a specification, and as such requires only the presence of the switch name, e.g.

'SWITCH' S1,S2;

specifies two common switches.

When one of these switches, say S2, is declared in the outermost block of one of the program segments, the declaration will be of the form.

'SWITCH' S2 := L1, L2, L3;

The use of the labels in this declaration is treated exactly the same, semantically, as any other use of them. Thus if the labels are not set in the outermost block of the segment containing the switch declaration, they must be specified in common. However, even in this case the common communicator does not specify any relationship between the label and the switch.

1.1.1.2.2. Library communicator

This allows reference to commonly used procedures, avoiding recompilation with each program.

The format of a Library communicator is:

'LIBRARY' Library procedure specifications;

.
.
.
.

'LIBRARY' Library procedure specifications

The specification of a Library procedure is identical to that of any other procedure specification as described in OD 8.3.4 except that the procedure name is of the form

Name/no.

for the reason described in 1.1.1.1.4.

The further communicators described in OD 9.3 and 9.4, i.e. EXTERNAL and ABSOLUTE respectively, do not form part of 920C CORAL because they serve no purpose since there is no general operating system and it is not possible to link load programs which have been produced independently from the Compiler, with 920C CORAL programs. (Note that [constant] allows access to a core location where 'constant' is the absolute address).

1.1.1.3 Transfer of Control between Segments

The start of a program is considered to be the start of the first segment loaded.

The compiler inserts a dynamic stop at the end of each segment. Therefore, in order to cause execution to pass from one segment to another, the first segment must contain a 'GOTO' statement to a following segment. The Compiler considers the 'segmentname' to label the first statement of a segment which can therefore be referenced as a normal label - for reference between segments it must be declared in Common.

e.g.

```
'PROGRAM' X
'COMMON' ('LABEL' SB, .....);
'SEGMENT' SA
'BEGIN'
        |
        |
        |
        |
    'GOTO' SB
'END'
'SEGMENT' SB
'BEGIN'
        |
        |
        |
'END'
'FINISH'
```

1.1.1.3 (cont.) If the CORAL source is to be loaded as a multilevel program (1.2.3.1) a further envelope is included around the set of segments for each level, which is described in (5.4.4).

1.1.1.4 Object Code Limits on Compilation Unit Sizes

Due to the object code strategy of absolute addressing of data, the following limits exist:

(1) The data area generated by 1 CORAL compilation unit must be ≤8K.

(2) The executable code generated by 1 CORAL compilation unit must be ≤8K.

(3) The data area generated by 1 CORAL program whether compiled as a whole or in separate units must be ≤8K, since it must lie within module 0. However, core locations outside module 0 may be accessed as data via indexed variables or anonymous references with large indices.

* (4) The executable code generated by 1 CORAL program must be ≤16K and therefore if >8K must be compiled in sections to adhere to (2).

* (5) The data area and executable code generated by 1 CORAL program must be ≤16K although any core locations above 16K may be accessed via indexed variables with large indices.

A full description of the runtime storage and object code strategy is provided in Chapter 5.

* But see EXTENDED LOADER manual, page 2.

1.1.2 DECLARATIONS

(1) The scoping rules implemented by the 920C Compiler are as defined in OD 3 - labels and variables are not allowed to have the same scope, i.e. a label and an identifier cannot have the same name within a block.

(2) The maximum number of declarations allowed in a data declaration list is 31.

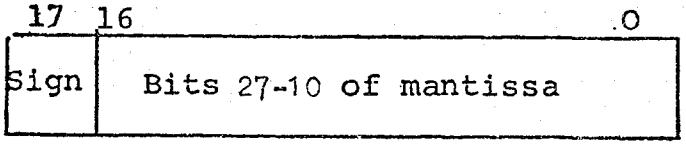
1.1.3 NUMERIC TYPES (OD 4.1)

There are three types of number, floating point, fixed point and integer, all of which are used as defined in OD 4.1. The representation of these numbers at runtime is described below (bit 0 is the most significant bit of a word).

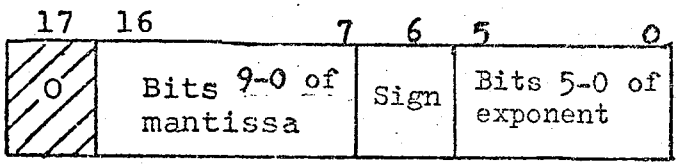
1.1.3.1 Floating Point

Floating point numbers are held in two words thereby adhering to the standard Elliott packed format. The mantissa is held in twenty-eight bits of the first and second words including the sign, and the exponent is held in seven bits of the second word:

First word



Second word



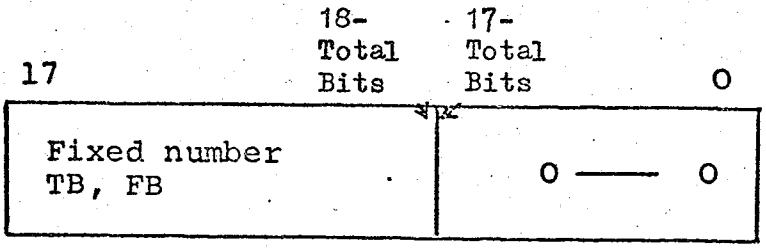
(Bit 17 is always zero)

The range of a floating point number is:

$$-9\cdot2*10^{18} \leq \text{no.} \leq +9\cdot2*10^{18}$$

1.1.3.2 Fixed Point

Fixed point numbers are held left justified including the sign, i.e. the position of the number is 17→18-TOTALBITS, redundant bits are held as zero:

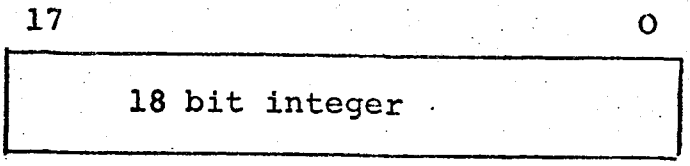


The range of a fixed point number is

$$2 \leq \text{Totalbits} \leq 18$$
$$-1023 \leq \text{Fractionbits} \leq +1023$$

1.1.3.3 Integer

Integer numbers occupy the full 18 bit word as normal, i.e. right justified.



Their range is $-131072 \leq I \leq 131071$.

N.B. An integer input in decimal must be in the range

$$-131071 \leq I \leq 131071$$

But an integer input in octal can be in the range

$$400000_8 \leq I \leq 377777_8$$

$$(-131072) \quad (131071)$$

1.1.4 PACKED DATA (OD 4.4)

1.1.4.1 Wordposition and Bitposition (OD 4.4.2)

The syntax for 'wordposition' allows negative representation

i.e. Wordposition = Signed integer

although the Official Definition states explicitly that Wordposition is to be numbered from zero upwards. However negative representation is allowed to increase flexibility in the use of tables.

1.1.4.2 Floating Table Elements

Floating table elements occupy exactly two words and the wordposition in the table element declaration refers to the first.

1.1.4.3 Results of Partword Table Element Access (OD 4.4.2.2)

The type of the result of a partword table element access is the type of the element specified in the table declaration.

1.1.4.3.1 Signed Integer Partword Table Element (i.e. Fractionbits not specified)

The result is an integer of width Totalbits and right justified with the sign extended. Since integers are held in a full word and right justified the result is actually an 18 bit integer with 'non-significant' bits set to the sign. Therefore the use of a signed integer partword table element in an expression is the use of the resulting 18 bit signed integer. For an example see 1.1.4.3.5.

1.1.4.3.2 Signed Fraction Partword Table Element (i.e. Fractionbits specified)

The result is a fixed number of the specified scale of width Totalbits left justified, including the sign, with non-significant bits set to zero. Therefore the use of a signed fraction partword table element in an expression is the use of the resulting fixed number as if it had been declared as a whole word element. For an example see 1.1.4.3.5.

1.1.4.3.3 Unsigned Integer Partword Table Element (i.e. 'UNSIGNED' and Fractionbits not specified)

The result is an integer of effectively width Totalbits, right justified, plus a zero extended sign. Since integers are held in a full word and right justified the result is actually an 18 bit positive integer. Therefore the use of an unsigned integer partword table element in an expression is the use of the resulting 18 bit positive integer. For an example see 1.1.4.3.5.

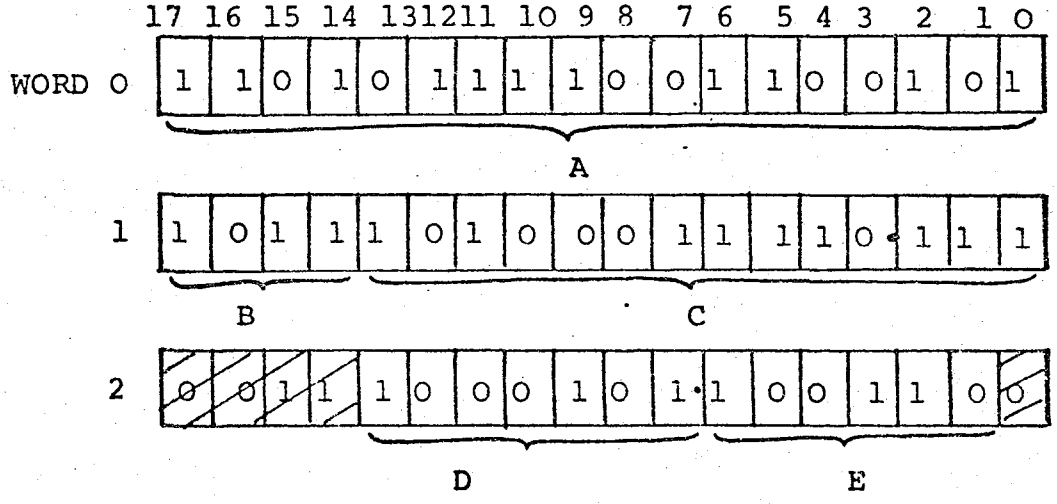
1.1.4.3.4 Unsigned Fraction Partword Table Element (i.e. 'UNSIGNED' and Fractionbits specified).

The result is a fixed number of the specified scale of width Totalbits+1 left justified up to a zero sign bit and with non-significant bits zero. Therefore the use of an unsigned fraction partword table element in an expression is the use of the resulting fixed number as if it had been declared as its equivalent whole word quantity of Totalbits + 1. For an example see 1.1.4.3.5.

1.1.4.3.5 Example of Table Declaration containing Partword Elements

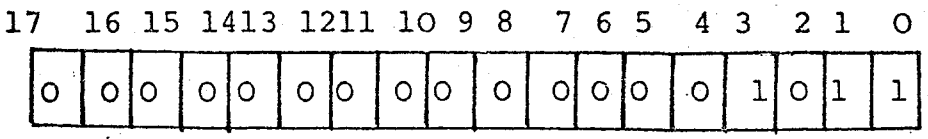
```
'TABLE'  X[1,3]
        [A 'INTEGER' 0;
         B 'UNSIGNED' (4) 1,14;
         C (14,3)1,0
         D 'UNSIGNED' (7,0)2,7;
         E (6)2,1 ] ;
```

1.1.4.3.5
(cont.) An example of a table entry is:

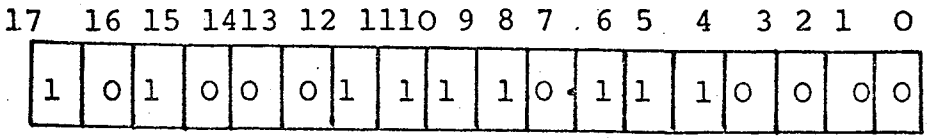


The use of each of the partwords B,C,D and E in an expression is the use of the following whole word:

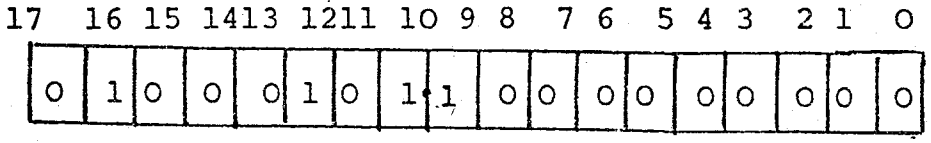
B : Unsigned integer



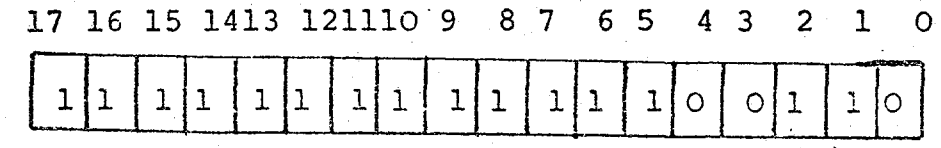
C : Signed fraction



D : Unsigned fraction



E : Signed integer



1.1.4.3.6 Restrictions on Partword Table Elements

(1) Signed

$2 \leq \text{Totalbits} \leq 18$ $0 \leq \text{Bitposition} \leq 16$

(2) Unsigned

$1 \leq \text{Totalbits} \leq 17$ $0 \leq \text{Bitposition} \leq 17$

(3) If either (1) or (2) have Fractionbits specified

$-1023 \leq \text{Fractionbits} \leq +1023$

1.1.5 OVERLAY DECLARATIONS (4.8)

1.1.5.1 Restrictions

Certain restrictions exist on the method and use of data overlaying due to the structure of the CORAL language.

(1) Data declared in a segment can only be overlayed by other data declared in the same segment, and data declared in COMMON can only be overlayed by an overlay declaration in COMMON. Data declared as an overlay can therefore be preset in accordance with the usual rules.

Internal procedure parameters may be overlayed in accordance with the usual rules but Common and Library procedure parameters cannot be overlayed.

(2) The 'Base' of an overlay declaration can be a formal value parameter of a procedure but for other types of parameter it has no meaning and is regarded as illegal.

(3) The 'Base' of an overlay declaration can be an unindexed array and it is the responsibility of the CORAL programmer to ensure that it is a meaningful declaration.

(4) Data declared as an overlay will not be overlayed by any succeeding declarations. Thus the declarations:

```
'INTEGER' I,J;
'OVERLAY' I 'WITH' 'INTEGER' A,B,C;
'INTEGER' K;
```

will not cause C to refer to the same location as any other variable.

1.1.5.2 Overlay of Floating Variables

The effect of overlaying a fixed or integer variable onto a floating variable is that the overlaying variable occupies the same store location as the first (most significant) word of the mantissa of the floating variable.

It must be noted that overlay of a floating variable onto a fixed variable could produce undesirable side effects in that the programmer cannot always know which variable the second word of the floating variable is overlaying and writing to the floating variable could corrupt the program in an undefined way.

1.1.6 PART-WORDS (OD 6.1.1.2.2)

1.1.6.1 Bitposition

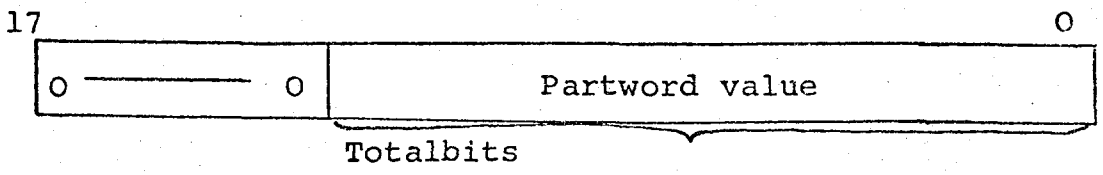
The bit numbering is as per the Official Definition.

1.1.6.2 Floating Variables

Partwords of floating variables are not allowed. The desired effect can be obtained by overlaying the floating variable with two integers and extracting the required partwords from the integers.

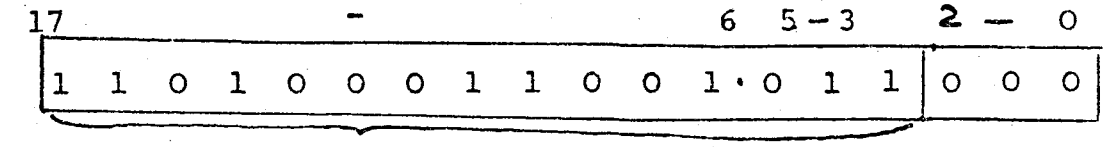
1.1.6.3 Result of Partword Access

As stated in OD 6.1.1.2.2, the result of a partword access is an integer of effectively width Totalbits, right justified, plus a zero extended sign. Since integers are held in a full word and right justified the result is actually an 18 bit positive integer.

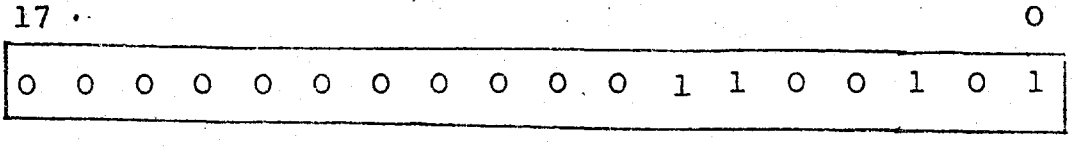


Therefore the use of the result of a partword in an expression is the use of the resulting 18 bit positive integer.

e.g. A 'FIXED' (15,3)X variable has the bit pattern:



'Bits'[6,5]X in an expression results in the use of the integer:



This is analagous to the use of an unsigned integer partword table element (1.1.4.3.3).

1.1.7 LOCATION EXPRESSION (OD 6.1.1.2.3)

The meaning and use of LOCATION expressions is as described in OD 6.1.1.2.3.

Since absolute as opposed to module relative addressing of data has been chosen as the object code strategy of the 920C CORAL Compiler, all data addresses are relative to the start of core. As a result, the interpretation of both LOCATION and ANONYMOUS REFERENCE are well defined irrespective of the parts of the program, and therefore the positions in core, in which they are used.

With regards to procedure parameters - LOCATION of a LOCATION parameter is LOCATION of the actual parameter and LOCATION of a VALUE parameter is LOCATION of the local workspace location within the procedure holding the value (5.2.3.2).

1.1.8 WORD-LOGIC (OD 6.1.2)

Word-logic operators operate on integer and fixed point typed primaries and the integer results of 'inner' word logic operations. Word-logic operations are not allowed on floating point items since they occupy more than one word.

1.1.9 EVALUATION OF EXPRESSIONS AND CONDITIONS (OD 6.1.3 and 6.2)

The following description uses the words 'term' and 'factor' as in the Official Definition:

A 'term' is an argument of + or -
A 'factor' is an argument of * or /

1.1.9.1 General Algorithm

The algorithm used in the evaluation of the following operations is not, in general, defined by the Official Definition for:

(1) Addition, Subtraction
(2) Multiplication, Division
(3) Conditional expressions
(4) Conditions
(5) Untyped primaries

The only defined algorithm is that for the evaluation of outermost terms where the required type and scale is known. In these cases the outermost terms are converted to the required resultant type before the addition or subtraction is carried out. The use of Numbertype in any context causes the enclosed expression to be treated as outermost.

An algorithm has been chosen to:

(1) Produce a consistent solution

(2) Utilise the 920C hardware operations

(3) Maintain maximum possible accuracy

(4) To overcome the fact that in the general case the types of the arguments of an operation and the resultant type may be undefined by context:

e.g. (a) The argument of + or - are untyped-primaries or the result is an untyped-primary (can be defined by context).

(b) The argument of * or / are untyped-primaries; the result may not have a defined type.

1.1.9.1 (cont.)

(c) Conditionalexpressions have untyped Unconditionalexpressions or Expressions. The required resultant type is always undefined.

(d) The Condition may consist of Untyped-primaries; the resultant type is always undefined.

The algorithm is as follows:

(1) If the type of an expression is unspecified and is not deducible from the context then it is evaluated as floating unless all its terms are of the same scale or all its factors are integer in which case it produces a result of the same scaling. Constants will be considered as having any scale, i.e. they will take the scale most suitable for their context.

This is applied to Conditionalexpressions, Conditions and Untypedprimaries.

(2) If the type of an expression is specified or deducible from context then it is evaluated to that type and care is taken to maintain maximum accuracy possible.

Thus for addition, subtraction the arguments are converted to the resultant type before the operation as for outermost terms. For multiplication and division the compiler automatically scales the arguments before or after the operation to obtain the required resultant type, if at all possible.

Example:

```
'BEGIN'
    'INTEGER' I,J ; (range -131072 to +131071)
    'FIXED' (18,17)A,B; (range -1.0 to 0.9999)
    'FIXED' (10,5)X,Y,Z; (range -16.0 to 15.96875)

    A:=0.25;                (initial values)
    I:=8;
    J:=32;
```

1.1.9.1
(cont.)

```
L1: X:= A+I;        (expected result = 8.25)

L2: B:= I/J;        (expected result = 0.25)

L3: Y:= X*B         (expected result = 8.25 *
                     0.25 = 2.0625)

L4: Z:= (A+I)*I/J;  (expected result = same
                     as Y)

'END'
```

An explanation of the evaluation is as follows:

L1: A and I are shifted to type fixed (10,5) and then added to give the expected result.

L2: I will be divided by J to immediately give a fixed (18,17) result as required.

L3: X will be multiplied by B to give a fixed (27,22) result. This will be truncated at the least significant end to provide a fixed (10,5) answer correctly. (Fixed numbers left justified).

L4: Since (A+I) is untyped and they are both not integer, each will be floated and a floating addition executed. It will then be floated and multiplied to the previous result and then J will be floated and a floating division executed. Finally, the result will be fixed to the scale (10,5) giving the expected result.

If the user wishes to avoid these floating operations he would have to write:-

```
Z:='FIXED' (10,5) (A+I) *  I/J;
```

1.1.9.2 Scaling

Rescaling operations are performed according to the algorithm in 1.1.9.1:

(1) For + and - the arguments are rescaled before the operation and transferred into one of the forms described in 1.1.3 with non-significant bits removed.

(2) For * the arguments are rescaled after the operation into one of the forms described in 1.1.3 with non-significant bits removed.

(3) For / the arguments are rescaled before the operation to a double precision intermediate form if necessary to prevent loss of significance or overflow on the division and, similarly to above, after the operation into one of the forms described in 1.1.3 with non-significant bits removed.

NOTE: Care must be taken in choosing the scales of arguments in expressions since the use of widely differing fixed scales will cause loss of significance, especially within multiplication and division operations.

Under the normal algorithm operations involving multiplication immediately followed by division, e.g. (a*b)/c, are performed as a*b scaled to single precision and then divided by c. However, in order to prevent loss of significance, since the result of the hardware * and the dividend of the hardware / could both be double precision, a special algorithm has been incorporated which retains a double precision intermediate result if a multiplication is immediately followed by a division and all arguments, e.g. a,b,c, are of the same scale. If the arguments are not of the same scale the normal algorithm is used and the intermediate result is truncated.

1.1.9.2.1 Scaling of Conditions

It should be noted that conditions have no predetermined scale and will therefore, according to the algorithm of 1.1.9.1, be floated whenever the two arguments of a relational operator are not of the same scale. This will often be the case when the arguments are not integers, and if floating-point evaluation is not required then explicit typing should be used.

There is no requirement that all operands in a multiple condition containing the boolean operators 'AND' and 'OR' should be of the same scale. Each relational expression is processed separately.

1.1.9.3 Compile Time Arithmetic

Operations (other than multiplication or division) on integer and fixed constants are performed at compile time and the results used in the object code:

e.g. (6+2) 'RIGHT'X

is computed as:

8 'RIGHT'X

Multiplication and division and floating constant operations are always performed at runtime.

1.1.9.4 Overflow Checking

Overflow checking is only performed on floating operations and the rescaling of a floating value to a fixed or integer value, since this can be effected without increasing the size of the object code due to the method of floating point processing (2.3).

For all operations on fixed and integer quantities and rescaling to floating it is the responsibility of the CORAL programmer to ensure that the values are in the correct range, otherwise overflow will occur without warning.

1.1.9.5 Rounding

1.1.9.5.1 On rescaling operations

In order to produce the most efficient object code and since the 920C hardware does not round, the object code generated by the Compiler does not include any special code for rounding.

Therefore results of rescaling from fixed to integer are truncated as follows:

$x \cdot y \rightarrow x$	e.g. $2.3 \rightarrow 2$, $2 \cdot 7 \rightarrow 2$
$-x \cdot y \rightarrow -(x+1)$	e.g. $-2.3 \rightarrow -3$, $-2 \cdot 7 \rightarrow -3$

1.1.9.5.2 On Division

The 920C hardware always causes the result of a division to contain bit 17 set to 1.

i.e. Let the correct result of $^{x}/_{y}$ be z

If result should be:	odd	even
$\frac{x}{y}$ →	z	z + 1
$\frac{-x}{y}$ →	-z	-(z-1)

However, the 920C Compiler has incorporated an algorithm to produce the correct result - the dividend is doubled and the quotient is halved.

It must be noted that in extreme cases this could mean the loss of the sign of the dividend and therefore produce corrupted results.

No such algorithm is applied to the division of fixed numbers since bit 17 is rarely significant.

1.1.9.6 Order of Evaluation

The tightness of binding of operations is as follows

LEFT and RIGHT (equal)
MASK, UNION and DIFFER
* and / (equal)
+ and - (equal)

As far as possible expressions are evaluated in the order which produces optimum object code.

However, as stated in OD 6.1.3 function calls are evaluated in the order in which they appear when the expression is read from left to right so that possible side effects caused by interaction between them can be determined.

1.1.9.6
(cont.) Therefore, it must be noted that where:

```
a+b*c
```

may be evaluated as:

```
b*c+a.
```

```
fa + fb * fc
```

is evaluated as:

```
fa into wsa
fb * fc + wsa
```

Obviously nested function calls are evaluated in the reverse order, i.e. fd(fe(ff(x))) causes evaluation in the order ff, fe and fd.

It must be noted that the function calls in expressions containing nested function calls are extracted and evaluated before the expressions to eliminate the overwriting of the parameter space if calls to the same function are nested.

e.g. fg(fh+1,fg(x,fi+2)+3)

```
fh is evaluated into wsh
fi is evaluated into wsi
x  is evaluated into parameters space of fg
wsi+2     "        "      "        "    " "
fq is evaluated into wsg
wsh is evaluated into parameter space of fg
wsg+3     "        "      "        "    " "
fg is evaluated
```

Conditions are also evaluated from left to right but only as far as is necessary to determine their truth or falsity.

1.1.10 CODE STATEMENTS (OD 7.5)

Instructions enclosed by 'CODE' 'BEGIN' and 'END' consist of a subset of SIR, the 920C SYMBOLIC INPUT ROUTINE, and a general knowledge of this is assumed. Code instructions are terminated by semicolons and their elements are separated by commas to conform with normal CORAL statements, otherwise there are no restrictions on the format. No CORAL declarations or statements are allowed within a CODE statement other than comments. A code statement can only be used in the position of a CORAL statement.

The general form of an instruction in a code statement is:

LABEL: /F, ADDRESS;

where:

LABEL: (optional) Normal Label which is accessed within the code statement or within the CORAL source enveloping it.

/ (optional) Modification Bit

F Standard 920C function code, i.e. 0-15

ADDRESS (1) Identifier - either
The name of an actual or formal by value variable which is declared within the CORAL source enveloping the statement. (It cannot be the name of a formal by location variable).

or

The name of a label declared within the code statement or within the CORAL source enveloping it.(F=7/8/9)

(2) Unsigned integer constant
The absolute address of the location to be referenced by this instruction.

1.1.10
(cont.)

(3) Signed constant
A constant to be referenced and held in the object code as:
(a) A normal INTEGER if written as an integer, e.g.+1234
(b) A FIXED(18,17) number if written as a fraction, e.g.+·1234
No other types of constant are allowed.

The validity of a code statement is the responsibility of the programmer - The Compiler provides only a limited number of error checks (3.1) to allow maximum flexibility. Special care must be taken over the use of the H-register which is assumed to be in absolute addressing mode on entry to and exit from a code statement.

The syntax of a code statement is presented in Appendix A.

1.1.11 FOR STATEMENT (OD 7.10)

1.1.11.1 For-elements with STEP (7.10.1)

The Official Definition was in fact in error when the 920C CORAL Compiler was written (although later issues of the O.D. have been corrected); so the following definition was used:

Let the element be denoted by

CV:= e1 'STEP' e2 'UNTIL' e3

In contrast with Algol 60, the expressions are evaluated once only.

The sequence of operations is as follows:

(i) The first expression e1 is evaluated and assigned to the control variable.

(ii) The second expression e2 is evaluated to the scale and type of the control variable and stored in the anonymous location v2.

(iii) The third expression e3 is evaluated to the scale and type of the control variable and stored in the anonymous location v3.

(iv) The value of the control variable cv is compared with the limit value v3, if (cv-v3)*v2>0 then the for element is exhausted, otherwise

(v) The controlled statement is executed.

(vi) The increment v2 is added to the control variable and the cycle repeated from (iv).

Note that if the control variable is subscripted then the subscript will have been evaluated and the LOCATION of the controlled variable held in V1.

The control variable is allowed to be INTEGER, FIXED or FLOATING, and to be a member of an array, but not to be a partword. It is recommended that only simple integer control variables be used.

1.1.11.2 Entry of DO Loop

The language allows a GOTO statement to transfer control into the controlled statement of a FOR statement. It must be recognised that this is dangerous since the control variable may be undefined.

1.1.12 PROCEDURES (OD 8)

Procedures are declared and called as defined in OD 8. The following points should be noted:

(1) Answer Statement (OD 8.1)

The answer statement of a typed procedure cannot be embedded in a nested procedure.

More than one answer statement may exist in a typed procedure and it is also possible to exit without executing an answer statement by jumping to the end.

(2) Dimensions of Formal Arrays (OD 8.3.2.2)

If the Compiler encounters uses of a formal array which attribute to that array conflicting dimensionality an error message will be output (no.131)

(3) Non-standard Parameter Specification (OD 8.3.5)

This facility is not allowed.

(4) Number of Parameters

The maximum of 30 procedure parameters are allowed.

(5) Scales of Fixed Parameters

The scales of fixed parameters are <u>not</u> checked for a match between the specification and declaration of the same procedure.

1.1.13 LITERALS AND STRINGS (OD 10.3 and 10.4)

1.1.13.1 Character Representation

In both literals and strings characters are held in 7 bit ISO code form, i.e. the external ASCII (1.3.2) code minus the parity bit. Any lower case character is held in its equivalent upper case form.

1.1.13.2 Literals

Literals may be any legal CORAL character (1.3.1). The method of obtaining other characters using the ! facility is described in 1.1.13.3.

1.1.13.3 Strings

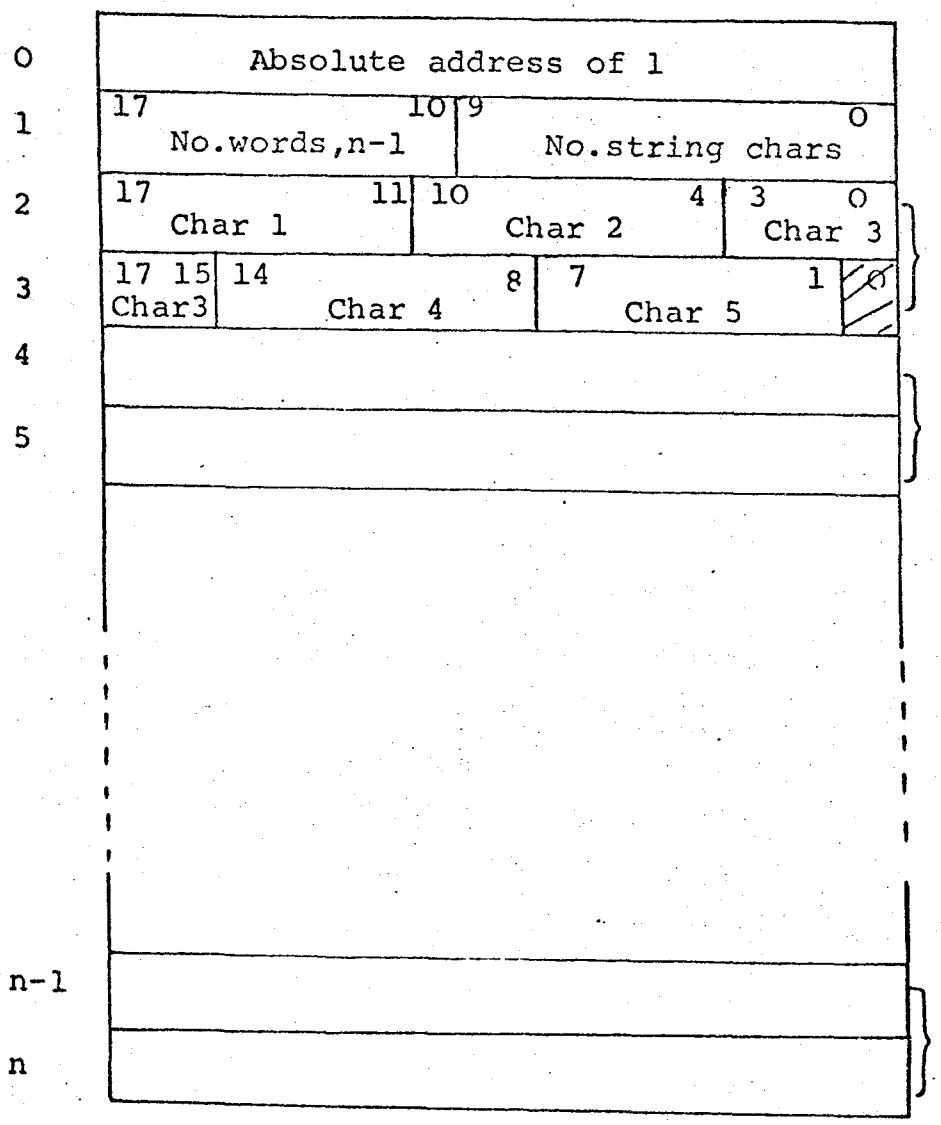
A string is delimited by the 'quotes' ↑< ' >↑ and can contain any ASCII printing and layout character. The quotes cannot be nested within a string (unless it is a macro definition - 1.1.15.1). Printing characters are stored within strings but layout characters are ignored. In order to get layout characters stored they can be represented by an identifier, optionally followed by an integer, within exclamation marks. For example, !L5! within a string will be interpreted as five consecutive linefeed characters. (If the integer is omitted then it is assumed to be 1). The full set of identifiers is:

S	space (Normal spaces between words are included without having to use the !Sx! facility)
C	carriage return
L	line feed
F	form feed
T	Horizontal Tab
H	haltcode

X followed by a decimal character representation indicates the external representation of a character, that is, the value of the tape code that would be output if the string were punched out. The maximum number of characters allowed in a string is 630.

As stated in the OD 10.4 a string is classed as an unconditional expression and its value is its address. Using this address the string characters may be accessed. The runtime format of a string is:

1.1.13.3
(cont.)



The contents of word 0 is accessed when a string is used, i.e. the value of a string is the address of word 1.

1.1.14 COMMENTS (OD 11.1)

1.1.14.1 Comment Sentences (OD 11.1.1)

A comment sentence may be written wherever it is convenient, i.e. between CORAL symbols, and not just wherever a declaration or statement can appear. Therefore comment sentences may precede the outermost block of a segment.

1.1.14.2 Concatenation of Comments

The concatenation of comments in the following way:

```
COMMENT      ;  (      )
```

is legal.

1.1.15 MACRO FACILITY (OD 11.2)

1.1.15.1 Macro Definitions

A macro definition is delimited by the quotes ↑< and >↑ which may be nested. Any ASCII character (1.3.2) may exist in a macro definition and layout characters are significant.

With the exception of the circumstances described in 1.1.15.4, macro definitions may occur in any suitable position in the CORAL source (like comment sentences). It must be noted that the scope of a macro definition does not follow the block structure but is always active until it is deleted.

1.1.15.2 Macro Deletions

With the exception of the circumstances described in 1.1.15.4, macro deletions may occur anywhere.

1.1.15.3 Macro Calls

OD 11.2.2 states that the actual parameters to a macro call should be treated as strings of characters, which are used to set up a 'virtual' macro body of which the corresponding formal parameter is the name. The analysis of an actual parameter as a string of characters, however, poses a number of problems, since the string delimitter (the comma or right parenthesis) can occur in a number of positions within the string as well. An example of such a situation is

```
CALLMAC ( (,↑< CHARS)>↑'COMMENT' THIS IS ONE,

OR MORE, PARAMETERS;  (POSSIBLY THREE),

                (CHARS  [ MORECHARS) ] )
```

It has therefore been decided that only legal CORAL symbols should be permitted as actual parameters, and that the rules for their use should be as follows:-

(1) Only legal symbols are permitted.

(2) Comment sentences and strings are regarded as single symbols and the occurrence of parameter delimitters within them is not recognised.

1.1.15.3 (Contd) (3) The construction 'LITERAL' (C), where C is any single character, is treated as a single symbol. It follows from this that at any point in the source 'LITERAL' (C) is also regarded as a single symbol, and that the entity between the brackets cannot be a macro call.

(4) Nested round and square brackets are not independent of each other. Thus [(CHARS]) is not regarded as a legal parameter.

(5) The characters comprising the symbols of actual parameters are stored as read. Identifiers that are macro names are not expanded, and macro directives ('DEFINE' and 'DELETE') are not recognised.

1.1.15.4 Macro Expansion

The process of setting up a macro expansion involves creating virtual macro definitions, which, for the duration of the expansion, can be regarded as having the same status as macros defined by the use of the 'DEFINE' directive. This gives rise to a number of restrictions on the use of parameters. The rules governing what may occur within a macro body are listed below. It should be noted that the error situations that can arise are trapped at the point of expansion, not definition.

(1) A formal parameter to the current macro may have the same name as a previously defined macro. While the current macro is being expanded the previous definition of the parameter is inaccessible, but it is restored when the expansion terminates.

(2) It is not possible to define or delete a name the current meaning of which is a macro parameter of a macro currently being expanded.

(3) It is not possible to delete or redefine a name which is the name of a macro currently being expanded.

1.1.15.4 (Contd) (4) A macro, A, may contain a definition of another macro, B. If so, the definition of B becomes active when A is called, and remains active after the expansion of A terminates.

(5) If one macro calls another, their formal parameters may not have the same names.

(6) If a macro, A, contains a definition of another macro, B, then the names of the formal parameters of A and B may be the same provided that A does not also contain a call of B.

1.1.15.5 Recursive Macro Calls

Recursive macro calls are not trapped where they occur because under certain circumstances it is permissible to have a macro undergoing more than one level of expansion at once. This situation occurs when a macro has as one of its parameters a call to itself. Since expansion of the parameter occurs at the point where it is used, and this is within the expansion of the outer macro, the situation is apparently identical to a genuine recursive macro call. However, a recursive call will rapidly exhaust the core available, and it is considered that this is a sufficient error indication.

1.1.15.6 Nested Macro Definitions

A nested macro definition, as for an outer macro definition, is valid from the point of definition until either the end of the program text is reached or the macro name is redefined, or deleted.

OD 11.2.4 states that if a redefined macro is deleted, it is the most recent definition which is deleted, and the previous one is reinstated, where 'recent' and 'previous' refer to the sequence of the written text of the program. However, if the redefinition of a macro is nested within a macro which is not called before the deletion of the initial macro, the terms 'recent' and 'previous' have the opposite meaning.

1.1.15.6 e.g. (Contd)

```
'DEFINE'   A  ↑<---------->↑;      (*)

'DEFINE'   B  ↑<----------'DEFINE'A-------->↑;

'DELETE'   A;          (Causes * to be deleted)
```

Note that for the most efficient use of Compiler data space macros should be deleted in reverse order to their definitions.

1.2 CORAL LANGUAGE EXTENSIONS

This section describes the extra language facilities allowed by the 920C CORAL Compiler.

1.2.1 SHIFT OPERATORS

The operators 'RIGHT' and 'LEFT' are provided to allow specification of right and left shift operations. The results of the shift operations are similar to the standard 920C shift instructions:

RIGHT: Arithmetic right shift (sign regeneration) without rounding.

LEFT : Logical left shift with non-significant bits cleared.

A shift operation is written:

x 'RIGHT' y

where

x is an integer or fixed point typed primary (shifts on floating items are not allowed).

y is an expression whose value at runtime must be $-36 \leq y \leq 36$ - outside this range the shifts will have undefined effects due to the hardware.

y is rescaled to type INTEGER if not already of that type but x is never rescaled before the operation. The result of a shift operation is the resulting bit pattern considered to have type INTEGER.

Shift operators have tighter binding power than the Boolean operators UNION, MASK and DIFFER.

The syntax of the shift operators is described in Appendix A.

Note: Since 'LEFT' includes a mask to clear redundant bits from the Q but 'RIGHT' does not (see 5.4.1.3)

```
        A 'LEFT' B
and     A 'RIGHT' C      where C = -B
```

will not necessarily produce the same result since redundant bits from Q will not be cleared in the latter case. It is recommended that only shifts by positive powers of two are performed or alternatively a 'MASK' operation is attached to the shift.

1.2.2 BIT AND BYTE ARRAYS

A facility is provided for defining arrays of elements of less than a full word in length - these elements may be either a single bit or a byte consisting of nine bits.

Bit and byte arrays can be used in the same way as any other data array (OD 4.3) where the word ARRAY is preceded by BIT or BYTE on the declaration.

The syntax is presented in Appendix A.

1.2.2.1 Storage Space

Consecutive BIT and BYTE ARRAY declarations are not closely packed and each array starts on a word boundary (see below).

(1) BIT ARRAY

Storage space for a BIT ARRAY begins on a word boundary with sixteen bits per word, right justified in bits 15-0 and it assumes element 0 lies in bit 15 of the first word (which may be conceptual).

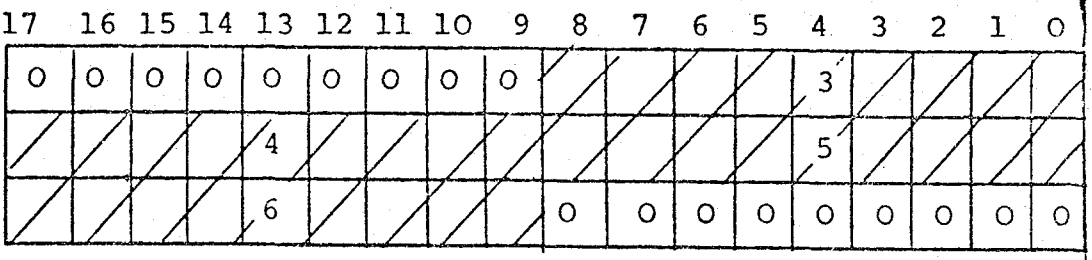
e.g. BIT ARRAY A[23:41] occupies two words:

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	23	24	25	26	27	28	29	30	31
0	0	32	33	34	35	36	37	38	39	40	41	0	0	0	0	0	0

1.2.2.1
(cont.) (2) BYTE ARRAY

Storage space for a BYTE ARRAY begins on a word boundary with two nine-bit bytes per word and it assumes element 0 lies in bits 17-9 of the first word (which may be conceptional).

e.g. BYTE ARRAY A[3:6] occupies three words:



1.2.2.2 Bit and Byte Array Access

The result of a bit or byte array element access is the positive INTEGER whole word with the bit/byte at the least significant end, i.e. bit 0 or bits 8-0 respectively. This corresponds to the part-word access of an equivalent number of bits (1.1.6.3). A bit/byte array reference can therefore be used as a partword reference (an integer variable on the left-hand side of an assignment) or as a typed primary with an integer value. The use of bit and byte arrays is inefficient compared with the use of whole word arrays (5.4.1.1).

1.2.2.3 Presetting of Bit and Byte Arrays

Both bit and byte arrays are preset by whole word integer constants which hold the values of the 16 bits or 2 bytes for the word which is being preset.

1.2.2.3
(cont.) e.g.'BIT''ARRAY'A[4:40] :='OCTAL'(005252),
'OCTAL'(125252),'OCTAL'(125200);

17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0
0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0
0	0	1	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0

1.2.3 RUNTIME FACILITIES

This section describes the runtime facilities provided by the 920C CORAL Compiler.

It must be noted that no runtime diagnostic checking or tracing facilities are provided although the conditional compilation facility (1.2.4) may be used to incorporate optional user diagnostics.

1.2.3.1 Multi-level Programs

1.2.3.1.1 A facility is provided for compiling multi-level programs where the interrupt handling housekeeping code is generated by the Compiler.

The operating instructions for loading and executing a multi-level program are provided in 3.1.5.1.

A CORAL program may be split into segments which run on different levels, a minimum of one segment per level, and all levels must be present. As for a normal program the same Common communicator must accompany each unit of compilation and is therefore shared between levels. The level upon which a segment resides is not fixed until loading, and each segment should be written as a normal segment - the segments of each level must be chained into a loop by making each segment terminate with a 'GOTO' nextsegname;' (1.1.1.3) and the last segment must 'GOTO' the first. ('GOTO' first segment name; causes a level terminate and on interrupt processing is resumed at the first segment. 'First' refers to the order of loading and for clarification of this with regards to compilation refer to 2.1.5.3)'. The Loader embeds the segments into the interrupt handling housekeeping code and no knowledge of this is required by the user. (For information it is described in 5.4.4). A multi-level program can therefore be written in pure CORAL with no provision for interrupt handling code other than segmenting the program according to its levels of execution.

CORAL Code, i.e. Common procedures, should not be shared between levels since 920C code is not re-entrant, and it is the responsibility of the user to ensure that this does not happen - no checks are performed by the Compiler. (The user need not maintain four copies of Library procedures, one for each level, since the Loader will regenerate them as required). Similarly, care

1.2.3.1.1 (contd) must be taken in updating Common data which is shared between levels.

1.2.3.1.2 The following information is provided for use in specialised circumstances - it has no effect on the object program and can be ignored if not required.

It is sometimes useful to distinguish between the reasons for entering a program at level 1:

(1) Initial start of a program after loading.

(2) Re-entry via AUTOSTART after power has been switched off and on: note that this does cause the program to be re-entered at its Start, NOT at where it was when the power went off; because 920C hardware does not provide the "power fail interrupt" needed to implement the latter.

(3) A top level interrupt.

For this purpose a flag, TOP LEVEL INDICATOR, has been supplied by the Compiler which will automatically contain the following values according to the above conditions:

+0 Initial start on level 1
-1 AUTOSTART
+1 Top level interrupt

The position of this flag within the object code is :

Module 0 lower bound + 4.

It is therefore necessary for the user to define the macro :

'DEFINE' TOPLEVELIND↑<absolute address of MCL + 4 >↑ ;

in order that the flag may be accessed mnemonically within the CORAL program. The module 0 lower bound is either the default value or the value specified by the user on loading (2.1.5.2).

For clarification of the routine storage of a CORAL program see 5.1.

1.2.3.2 Program Sumcheck

The 920C CORAL Loader calculates the runtime sumcheck of the executable code of a CORAL program and stores it in the object program for runtime sumchecking within the user program. It has no effect on the object program and can be ignored if not required. The sumcheck together with the program code bounds are provided in the six locations from the module 0 upper bound (2.1.2.2) downwards:

MOU	Module 1 code area upper bound (=M1U)
MOU-1	Module 1 code area lower bound
MOU-2	Module 0 code area upper bound (=MOU)
MOU-3	Module 0 code area lower bound
MOU-4	-1
MOU-5	-sumcheck

The sumcheck is the negated accumulation, ignoring overflow, of the contents of each word between the upper and lower code bounds in each module and it is accumulated as the program is loaded. The data area of a program is not included in the sumcheck since it contains an inseparable mixture of fixed preset and variable non-preset data.

The following program should be used for the runtime sumcheck:-

```
'BEGIN' 'INTEGER' SUM, ADDRESS POINTER, WORD POINTER;
SUM :=0;

'FOR' ADDRESS POINTER := HIGHEST,
ADDRESS POINTER - 2 'WHILE'[ADDRESS POINTER]  'GE' 0
'DO'

'FOR' WORD POINTER:=[ADDRESS POINTER - 1]
'STEP' 1 'UNTIL' [ADDRESS POINTER]
'DO'

SUM :=SUM + [WORD POINTER];

'IF' SUM = 0
'THEN' sumcheck ok ....
'ELSE' sumcheck fail ....;

'END'
```

1.2.3.2 (cont.) Where HIGHEST should be declared amongst the system macro :-

'DEFINE' HIGHEST↑< absolute address of MOU>↑ ;

For clarification of the runtime storage allocation of a CORAL program see 5.1.

Note that this Sumcheck facility has been designed in such a way that, should a loader become available which loaded above 16K or into more than 2 code areas, NO change would be needed to the Sumcheck Program.

1.2.3.3 Initialisation of Data Area

Before loading the CORAL Loader sets the data area to be occupied by the CORAL unit of compilation to zero. However the Extended Loader does not. It is therefore recommended that any areas requiring initial values are either preset or are explicitly assigned to at the start of the user's program. Remember that presets are only set, and the above initialisation is only performed, when the program is first loaded: to ensure that a program can be restarted without reloading it, locations used for variables should be given any initial values needed by assignments.

1.2.3.4 Self-Triggering and Autostart.

The 920C CORAL Loader provides the following two facilities which are usually required of a real-time program; especially when the program is to be used in an application environment, without the normal 920C Control Console:-

* Self-Triggering. A Loader option is provided, whereby, when the user's program has been successfully loaded, it starts immediately.

* Autostart. The multi-level housekeeping code (1.2.3.1.1), includes the code required to make a program restart automatically after a power failure & restoration.

Further, means are provided to detect & distinguish these events (1.2.3.1.2).

There are 2 limitations on the use of the Autostart facility:-

* To use the Autostart facility it is necessary to load even single-level programs in the multi-level mode of the Loader, so that the housekeeping code is inserted.

* The housekeeping code sets up the Autostart so that the program restarts if "subsequently" powered up, for example after a power failure, and is in the AUTO mode or has no control panel. However these instructions do not get executed unless the program is "initially" triggered by some other means, namely by using a control panel or by using a Self-Triggering Binary tape.

  Thus it is recommended that programs needing the Autostart facility and written in CORAL are link-loaded so as to also be Self-Triggering.

  In those instances where it is undesireable to run the program when it has just been loaded, for example because the program loading unit has to be disconnected and some application peripherals attached before the program will run correctly, the Self-Triggering facility should still be used and the Top-Level program should start with something like:-

```
'IF' TOP REASON = JUST LOADED 'THEN' STOP ; .
```

There is no intrinsic reason on the 920C why either a program using Autostart should be multi-level, or that the use of Self-Triggering tapes & the Autostart facility should be related: these 2 limitations are due to the 920C CORAL implementation.

Fortunately the majority of real-time programs will be multi-level using Self-Triggering and Autostart, and the majority of off-line programs will be single-level and use neither Self-Triggering or Autostart.

1.2.4 CONDITIONAL COMPILATION

A facility for allowing optional compilation of statements is provided.

Under conditional compilation mode of operation the Compiler will compile any statement in the source which appears between the characters % and ; as a normal statement. Under normal mode of operation the Compiler will treat such a statement as a comment. With this facility the user may insert trace directives in the source program to be used during development and ignored when the program is working, simply by altering its mode of compilation.

e.g.
```
A:=B;
%PRINTVALUE(A);
C:=D;
```

Conditional Compilation:
```
B is assigned to A
The value of A is printed
D is assigned to C
```

Normal Compilation:
```
B is assigned to A
D is assigned to C
```

1.3 CORAL SOURCE REPRESENTATION

This section describes the CORAL language symbols and physical character codes to be used in a CORAL program.

1.3.1 LANGUAGE SYMBOLS

The full list of language symbols available in 920C CORAL is given below. This list is substantially the same as that given in APPENDIX 2 of the Official Definition.

AND	FIXED	OR
ANSWER	FLOATING	OVERLAY
ARRAY	FOR	PRESET
BEGIN	GE	PROCEDURE
BIT	GOTO	PROGRAM
BITS	GT	RIGHT
BYTE	HALT	SEGMENT
CODE	IF	STEP
COMMENT	INTEGER	SWITCH
COMMON	LABEL	TABLE
CORAL	LE	THEN
DEFINE	LEFT	UNION
DELETE	LT	UNSIGNED
DIFFER	LIBRARY	UNTIL
DO	LITERAL	VALUE
END	LOCATION	WHILE
ELSE	MASK	WITH
EQ	NE	
FINISH	OCTAL	

digits 0 to 9

upper case letters A to Z

+ - * /	arithmetic operators
< ≤ = ≥ > ≠	comparators
( )	expression brackets
[ ]	index brackets
ǂ ǂ	string quotes
, ; : . % !	
←	assignment symbol
₁₀	subscript 10

Layout characters.

Language symbols that appear as words are delimited by acutes (single apostrophe), for example 'BEGIN', and they cannot be abbrieviated.

1.3.1
(cont.) The following representations for non-alphanumeric language symbols are used:

Official Definition	Representation	Alternative Representation
<	<	'LT'
≤	<=	'LE'
=	=	'EQ'
≥	>=	'GE'
>	>	'GT'
≠	<>	'NE'
←	←	:=
↑	↑<	
↓	>↑	
⊙	⊙	&

In processing CORAL source text, including code blocks, all layout characters are ignored and lower case letters are read as upper case. (Within the Macro Pass lower case letters are interpreted as upper case with regards to analysis but they are output as lower case in the expended source). Their use as literals and within a string is described in 1.1.13 and within macro definitions in 1.1.15.

1.3.2 CHARACTER CODES

The character code accepted by the Compiler is ISO or ASCII code with even parity in the eigth track. Reference must be made to 1.3.1, 1.1.13 and 1.1.15 to determine which characters are allowed in CORAL source symbols, literals and strings, and macro definitions respectively.

CHAPTER 2

920C CORAL COMPILING SYSTEM

2.1 COMPILER PROGRAMS

- 2.1.1 MACRO PREPROCESSOR
  - 2.1.1.1 Description
  - 2.1.1.2 Options
- 2.1.2 PASS 1A
  - 2.1.2.1 Description
  - 2.1.2.2 Options
- 2.1.3 PASS 1B
  - 2.1.3.1 Description
  - 2.1.3.2 Options
- 2.1.4 PASS 2
  - 2.1.4.1 Description
  - 2.1.4.2 Options
- 2.1.5 LOADER
  - 2.1.5.1 Description
  - 2.1.5.2 Options
  - 2.1.5.3 Order of Loading
  - 2.1.5.4 Library Procedure Loading

2.2 DIAGNOSTIC PROGRAMS

- 2.2.1 COMPILER DATA RETENSION
  - 2.2.1.1 Description
  - 2.2.1.2 Options
- 2.2.2 OBJECT DUMP
  - 2.2.2.1 Description
  - 2.2.2.2 Options
  - 2.2.2.3 Position in Core

2.3 FLOATING POINT LIBRARY PROCEDURE

2.4 INTERFACE WITH THE USER

- 2.4.1 COMMAND LANGUAGE
- 2.4.2 COMMAND FORMAT

2.5 MISCELLANEOUS NOTES

- 2.5.1 COMMON CHECKING

2 920C CORAL COMPILING SYSTEM

The minimum configuration required for compiling a CORAL program using the 920C CORAL Compiling System is an Elliott 920C with 16K words of core store, a paper tape reader, a paper tape punch and a teleprinter.

The standard minimum configuration for executing a CORAL program is an Elliott 920C with up to 16K words of core store where the data always resides in module 0 and the code resides in either or both modules 0 and 1 (core above 16K can be used as indirect data space, i.e. referenced by an indexed variable or an anonymous reference with a large index). However, since the object code executes in absolute addressing mode and does not contain any of the special 920C instructions, a program which will load completely into the lower 8K, i.e. less than approximately 7K, may be executed on a 920B upwards compatable computer. *

The 920C CORAL Compiling System consists of:

Five Compiler Programs:

- Macro Preprocessor
- Pass 1A
- Pass 1B
- Pass 2
- Linking Loader

Two Diagnostic Programs:

- Compiler Data Retention
- Object Dump

A Runtime Library Procedure:

- Floating Point Package

The structure and purpose of these programs is described below followed by a description of the standard method of interface with user.

All of the following programs run on level 1 and all execute in absolute addressing mode except for the Loader which contains some module relative code.

* Some of these limits are relaxed using the EXTENDED LOADER.

2.1 COMPILER PROGRAMS

The following description outlines the purpose and options provided by each compiler program. It assumes a knowledge of the remaining chapters of this manual to avoid repetition. A diagram of Compiler operation and a summary of Compiler input/output are provided in Appendices C and D respectively.

The action performed as a result of error detection within each pass of the Compiler is described in the relevant section below. However, there is a class of checking which produces a warning message indicating that the user may have misused the language but not seriously enough to affect the compilation. In this case the compilation continues as normal and it is the responsibility of the user to determine if this is sensible and continue or terminate compilation accordingly. Reference should be made to Chapter 4 for all error and warning situations.

It must be noted that Passes 1A and 1B were originally one pass, hence their names, but had to be split due to their total size. The passes were not, therefore, originally designed as separate entities and the split was made as simple as possible - unfortunately resulting in the production of a large intermediate code in proportion to the source, for transfer between the Passes.

2.1.1 MACRO PREPROCESSOR

2.1.1.1 Description

The Macro Preprocessor is a separate prepass to the Compiler which purely expands text.

It accepts CORAL source text containing macro definitions, calls and deletions and produces macro-free CORAL source text having expanded all the macro calls and from which all the 'DEFINE' and 'DELETE' directives have been removed (in so doing it performs syntax analysis to CORAL symbol level). All characters, excluding the 'DEFINE' and 'DELETE' and the ↑< and >↑ pairs enveloping the macro definitions occurring in the source are copied into the expanded source. Detection of an error or warning situation will not inhibit the production of expanded source and it is the responsibility of the user to determine on completion of processing whether the output is useful or not.

Execution of the Macro Preprocessor is optional and only necessary if the CORAL source to be compiled contains macros.

The rules governing the position and contents of macro definitions, calls and deletions (1.1.15) allow the Macro Preprocessor to be used as a general software tool since it allows libraries of macro definitions to be built up independently of the CORAL programs in which they are referenced.

MACRO PREPROCESSOR

2.1.1.2 Options

The following options are provided by the Macro Preprocessor using the standard user interface (2.4)

(1) Source Output Device.

The expanded source may be output on the paper tape punch, teletype or not at all (for checking purposes).

The default is the paper tape punch.

(2) Error Output Device.

The error and warning messages may be output on the paper tape punch, teletype or not at all. (It must be noted that with (1) and (2) the expanded source and error messages will be intermingled if the same output device is requested).

The default is the teletype.

(3) Conditional Compilation (1.2.4)

If the conditional compilation language feature is required it is necessary to specify this requirement on the use of the Macro Preprocessor in order that any macro definitions, calls or deletions may be recognised between the % and ;. If not specified such statements will be treated as comment sentences.

The default is that this option is not required.

(4) Source Checksum.

The option of reading each source tape twice to perform a checksum test on it is provided with the Macro Preprocessor.

The default is that this option is not required.

2.1.2 PASS 1A

2.1.2.1 Description

Pass 1A is the first main pass of the compiler, the purpose of which is to syntax check the CORAL source.

It accepts macro-free CORAL source text for a unit of compilation and produces an intermediate code form of the source for input to Pass 1B, having performed all the necessary syntax checking. Further information is transferred to Pass 1B from Pass 1A within compiler data tables. The detection of an error will not cause the output of intermediate code to be inhibited and processing will continue in order that the intermediate code may be submitted to Pass 1B for semantic checking.

Execution of Pass 1A need not be preceeded by execution of the Macro Preprocessor unless the CORAL source to be compiled contains macro definitions, calls and deletions.

PASS 1A

2.1.2.2 Options

The following options are provided by Pass 1A using the standard user interface (2.4)

(1) Intermediate Code Output Device.

The Pass 1A intermediate code may be output on the paper tape punch, or not at all (for checking purposes).

The default is the paper tape punch.

(2) Error Output Device.

The error and warning messages may be output on the paper tape punch, teletype or not at all. (It must be noted that (1) and (2) cannot be requested for the same device).

The default is the teletype.

(3) Conditional Compilation (1.2.4)

If the conditional compilation language feature is required it is necessary to specify this requirement on the use of Pass 1A.

The default is that this option is not required.

(4) Source Checksum.

The option of reading each source tape twice in order to perform a checksum test on it is provided with Pass 1A.

The default is that this option is not required.

(5) Object Map.

The option of providing a map of the object code from Pass 2 during production of the relocatable binary must be specified on the use of Pass 1A. A description of the format of this map is provided in 4.1.4.2. The map may be produced on the paper tape punch or the teletype.

The default is that this option is not required.

PASS 1A

2.1.2.2 (Contd) (6) Floating Indication.

The option of providing an indication whenever the compiler invokes a rescaling operation to floating point format within Pass 1B is available. (The algorithm for evaluating expressions (1.1.9) may cause floating point processing to be invoked whether there are any floating items in the CORAL source or not).

The default is that no such indication is required.

(7) Stack Positioning Commands.

Three commands are available to give the user the option of repositioning the compiler stacks if required. The default values are set to enable the compiler to run in 16K with the maximum data space, and are

Stack start address, SSS	14710
Stack length, SSL	1670
Stack size difference, SSD	400

In general, to run the compiler in more core the stack length option will be changed. The other two options are less likely to be used. The stack size difference is the amount by which the stacks must be contracted at the end of Pass 1 to make room for the code of Pass 2 to be loaded, and obviously depends on the value of the stack start address.

2.1.3 PASS 1B

2.1.3.1 Description

Pass 1B is the second main pass of the compiler, the purpose of which is to semantic check the CORAL source.

It accepts the intermediate code from Pass 1A of a unit of compilation and produces a further intermediate code form for input to Pass 2 having performed all the necessary semantic checking. Further information is transferred to Pass 2 within compiler data tables. The detection of an error during Pass 1B or previously in Pass 1A will inhibit the production of intermediate code from Pass 1B but will not stop the processing of the intermediate code from Pass 1A in order to detect as many errors as possible in one run.

Execution of Pass 1B must normally be preceeded by that of Pass 1A due to the passing of information in compiler tables.

PASS 1B

2.1.3.2 Options

There are no specific options provided by Pass 1B - the output of intermediate code and error and warning messages will be automatically produced on the devices specified for that of Pass 1A.

The floating warning message option is requested in Pass 1A (2.1.2.2).

2.1.4 PASS 2

2.1.4.1 Description

Pass 2 is the third main pass of the compiler the purpose of which is to generate object code.

It accepts the intermediate code from Pass 1B of a unit of compilation and produces relocatable binary of the object code for input to the Loader. Pass 2 also provides (optionally) a map of the object program (4.1.4.2). The detection of an error, of which there are very few, causes processing to halt since it will be irrecoverable from the Compiler's or User's point of view. If any error has been detected during Pass 1A or Pass 1B, Pass 2 should not be executed and will itself report an error if this is attempted.

Execution of Pass 2 must normally be preceeded by that of Pass 1B due to the passing of information in compiler tables.

Note that contrary to any other information transferred between compiler passes each relocatable binary tape must be input to the Loader backwards, i.e. the character produced last from Pass 2 must be input first to the Loader.

In practice this will usually be achieved by winding up the relocatable binary tape, produced by Pass 2, backwards, when it is first produced, then writing its name on the "outside" end; so that it can then be read by the loader (possibly frequently, as in the case of a 'LIBRARY' tape) without further complication.

PASS 2

2.1.4.2 Options

There are no specific options provided by Pass 2 - the output of relocatable binary and error and warning messages will be automatically produced on the devices specified for that of Pass 1A.

Pass 2 provides a map of the object program if requested during the use of Pass 1A(2.1.2.2).

2.1.5 LOADER

2.1.5.1 Description

The Loader links together independently compiled CORAL units of a program into an executable program in core. It is purpose-built and therefore does not allow CORAL units of compilation to be linked with any other type of program unit produced via another compiler or assembler.

The Loader accepts relocatable binary from Pass 2 of one or more units of compilation which it link loads producing an executable program in core. As the loading process is performed information on the utilisation of core is printed on the teletype (4.1.5.2)

During loading detection of an error which is not considered disastrous does not inhibit the loading process and execution of the object program is at the user's discretion. However, an irrecoverable error will cause the loader to halt. It must be noted that an incomplete program, i.e. a subset of the units of compilation comprising the whole program, may be executed similarly at the user's discretion.

Execution of the Loader need not immediately follow that of Pass 2 since no information is transferred other than within the relocatable binary.

LOADER

2.1.5.2 Options

The following options are provided by the Loader using the standard user interface (2.4).

(1) Small Loader Option

This facility became obsolete when the Extended Loader became available; and its existance is only mentionned here to avoid renumbering the paragraphs & editing any cross-references.

The default is that the reduced Loader is not required.

(2) Radix of Input/Output.

The radix of the numbers input by the user on the teletype during the specification of options, and those output on the teletype by the Loader, i.e. core map, entry point and error numbers, may be either octal or decimal.

The default is octal.

(3) Core Module Bounds.

It is possible to instruct the Loader to reserve areas of core at either end of each core module and thereby load the object program between them. If the Loader is requested to reduce the space it has automatically reserved by default it reverts to the default values.

LOADER

2.1.5.2 (Contd)

The defaults are:

Module Ø	Lower Bound:	1054_8	556_{10}
Module Ø	Upper Bound:	17745_8	8165_{10}
Module 1	Lower Bound:	20000_8	8192_{10}
Module 1	Upper Bound:	34563_8	14707_{10}

N.B. It is not possible to force the whole program into module 1 since data must reside in module 0 although code may reside in either module 0 or 1.

See 5.1 for the runtime storage allocation of a CORAL program.

(4) Absolute Binary Dump.

The option of producing an absolute binary dump of the object program is provided. The dump is in the standard A.C.D. 900 Series 18-Bit Binary Tape Format 1/4/70 for future loading using the hardware initial instructions.

The default is that this is not required.

(5) Program Level Number.

The Loader provides the option of loading multi-level programs (1.2.3.1) whereby it creates all the interrupt handling house-keeping code which envelopes the segment(s) on each level of the program. The order of loading the units of compilation of a multi-level program is described in 2.1.2.3.2 and the interrupt handling house-keeping code sequences are listed in 5.4.4.

The default for the level of a unit of compilation when loaded normally is level 1 and therefore a single level program will always run on level 1.

LOADER

2.1.5.2 (Contd)

(6) Self-triggering

The Loader provides the option of producing a Self-triggering object code program on the absolute binary dump.

The default is that this is not required.

(7) AutoStart

No specific option is provided in the Loader to request the Autostart facility: the code needed is automatically inserted in the housekeeping code of multi-level programs; for it to function correctly Self-triggering must also be selected.
See 1.2.3.4.

LOADER

2.1.5.3 Order of Loading

2.1.5.3.1 Normal (single level) program.

The relocatable binary paper tapes for the units of compilation may be loaded in any order excluding the library tape(s) which must be loaded last. The entry point of the program is assumed to be the first segment compiled of the first tape loaded. The address of this is specified by the loader when loading is complete

e.g.
```
Segment tape 1   ← Program entry point
Segment tape 2
      ⋮
Segment tape N
Library tape(s)
```

2.1.5.3.2 Multi-level program.

The relocatable binary paper tapes for the units of compilation of a multi-level program must be loaded together for each level but may be in any order within the level followed by the library tape(s) for that level. (The levels may also be loaded in any order). The same library tape(s) may be read for each level and the Loader will automatically create a copy of each relevant procedure per level upon which it is used. The entry point of a level is the first statement of the first segment compiled on the first tape loaded at that level; the entry point of the program is assumed to be the entry point of level 1 (specified by the Loader on completion of loading).

e.g.
```
Segment tape 1, level 1  ←level 1 entry point
      ⋮                          (= program entry point)
Segment tape N, level 1
Library tape(s)

Segment tape 1,level 2   ←level 2 entry point
      ⋮
Segment tape N, level 2
Library tape(s)
```

LOADER

2.1.5.3.2 (Contd)

```
Segment tape 1, level 3  ←level 3 entry point
      ¦
Segment tape N, level 3
Library tape(s)

Segment tape 1, level 4  ←level 4 entry point
      ¦
Segment tape N, level 4
Library tape(s)
```

LOADER

2.1.5.4 Library Procedure Loading

As described in 2.1.5.3 the Library tape(s) are loaded after all units of compilation for the current level. The content of a Library tape is as defined in 1.1.1.1.4. The Loader performs a scan of each Library tape and loads only those procedures which have been referenced by previously loaded units of compilation, i.e. having the same Library procedure number. Any number of Library tapes may be scanned until all references are satisfied. The Loader outputs a description of the Library Procedures loaded (4.1.5.2).

The following points should be noted:

(1) The Loader performs no check on duplicate Library numbers and simply loads the first procedure encountered with the required number (last compiled since loading backwards) - all subsequent procedures with the same number being ignored. This therefore allows the user to redefine Library procedures.

(2) Since communication with a Library procedure is via the Library procedure number and not the name, reference to different procedure names which have the same number will cause calls to the same procedure at runtime.

(3) The Library tape supplied with the 920C CORAL Compiling System, CAPQF, contains the Compiler Floating Point Library Procedure (2.3) which has the Library number 1. The user should therefore avoid the use of this number since redefinition of the procedure would no doubt have disastrous consequences.

(4) Only Library procedures which are called by previously loaded units of compilation are loaded - redundant specifications at the head of a unit of compilation do not cause the respective Library procedures to be loaded.

2.2 DIAGNOSTIC PROGRAMS

The following programs are off-line programs supplied as part of the 920C CORAL Compiling System to aid in the development of CORAL programs.

2.2.1 COMPILER DATA RETENTION

2.2.1.1 Description

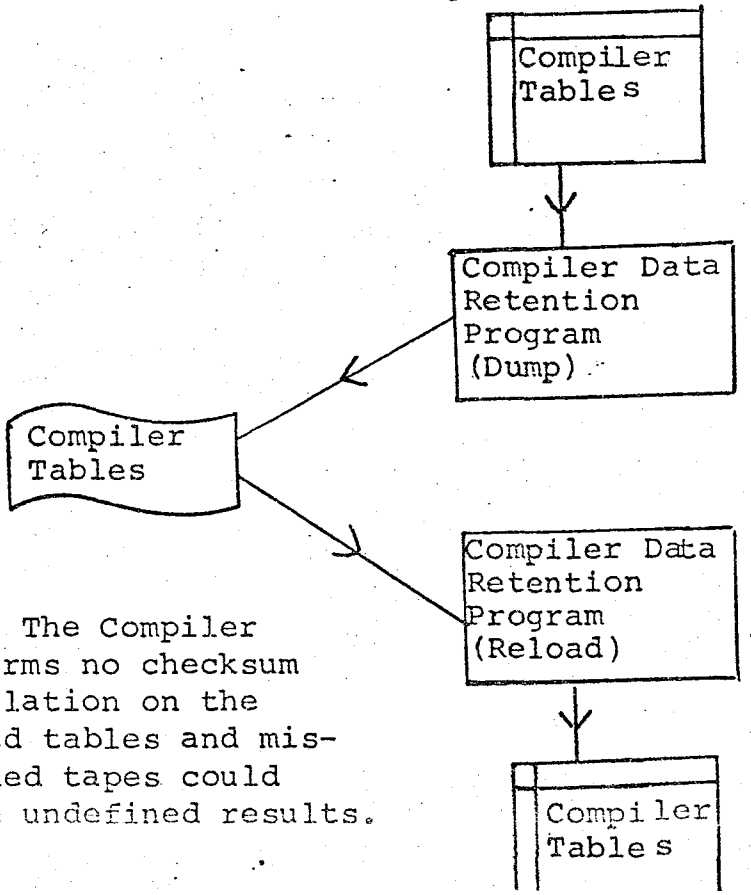
Since Pass 1A and Pass 1B transfer information in compiler data tables to Pass 1B and Pass 2 respectively, the three passes, Pass 1A, Pass 1B and Pass 2 must normally be run consecutively for the compilation of a CORAL unit without interruption. However, in order to provide the facility of running several units of compilation through a single pass, e.g. Pass 1A for syntax analysis, before proceeding to the next pass, the Compiler Data Retention Program is supplied which:

(1) Dumps the compiler data tables to the paper tape punch following compilation through a pass, to be retained by the user together with the relevant intermediate code.

(2) Reloads the dumped compiler tables from the paper tape reader before execution of the next pass.

Therefore, with the use of this program Passes 1A, 1B and 2 can be made independent of each other.

The following diagram summarises the operation of the Compiler Data Retention Program:



N.B. The Compiler performs no checksum calculation on the dumped tables and mis-punched tapes could cause undefined results.

COMPILER DATA RETENTION

2.2.1.2 Options

The only option provided by the Compiler Data Retention Program using the standard user interface (2.4) is whether the dump or restore of the compiler tables is required.

The default is that neither is required.

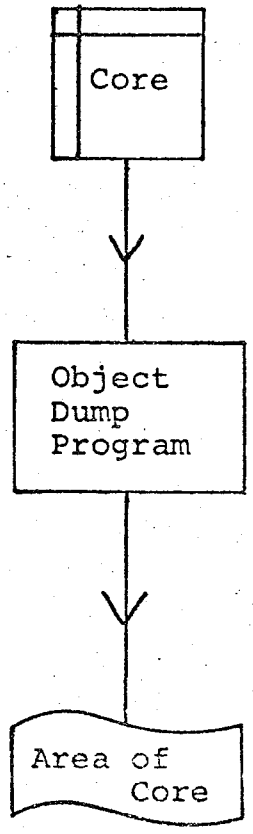
2.2.2 OBJECT DUMP

2.2.2.1 Description

In standard 900-Series terminology this program is not, as one might have thought, a Binary DUMP program, it is a STORE PRINT program giving Telecode output.

The Object Dump Program is a program completely independent of the rest of the 920C CORAL Compiling System. It is for use at run time to provide a readable dump of any requested areas of the 920C core store in octal, decimal and instruction word format. The format of the dump is described in 4.2.2. It can be used in conjunction with the object map produced by Pass 2 to determine the contents of the locations constituting a CORAL program.

The following diagram summarises the operation of the Object Dump Program:



OBJECT DUMP

2.2.2.2 Options

The following options are provided by the Object Dump Program using the standard user interface (2.4).

(1) Radix of Input

The radix of the numbers input by the user on the teletype during the specification of options may be either octal or decimal.

The default is decimal.

(2) Core Module Numbers.

The number of the core module to be analysed may be specified.

The default is module 0.

(4) Core bounds.

The bounds (inclusive) of the area to be dumped within the specified core module may be specified.

The defaults are -

Lower bound: 0
Upper bound: 0

(5) Dump Output Device.

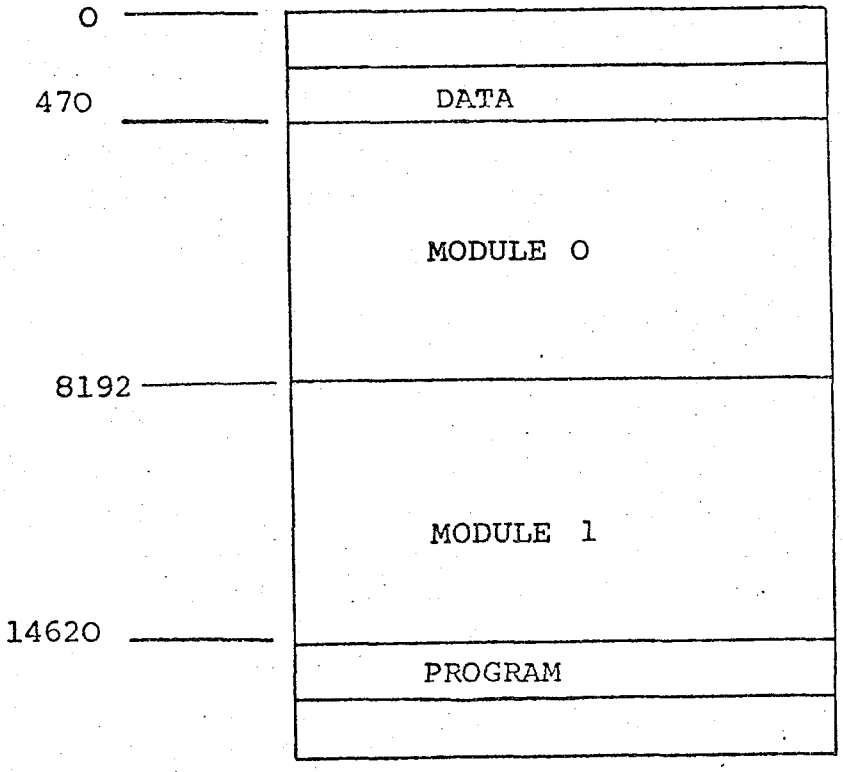
The dump may be output on the paper tape punch or teletype.

The default is the teletype.

Note: If the core module and core bounds combination specifies a non-existent core address the 920C computer will halt. There is no error check in order that this program can be used on different machine configurations.

OBJECT DUMP

2.2.2.3 Position in Core



Note: It is only recommended that this program is used for examining data after the object program has been run since unless module 0 upper bound is altered by the user it will overwrite part of the object program.

2.3 FLOATING POINT LIBRARY PROCEDURE

The floating point library procedure is the only library procedure issued as part of the 920C CORAL Compiling system.

Since the 920C has no floating point hardware the object code generated from CORAL source containing floating point operations has had to include software floating point processing. This is effected by using a procedure which interprets instructions as floating point operations when appropriate. This floating point procedure is therefore a library procedure and it is based on the standard 920C floating point package, QF, with modifications to suit the object code strategy of the Compiler. This floating point library procedure is inaccessible to the user since there is no mechanism for referencing it and it would serve no purpose - the invoking and use of this procedure is performed automatically and is transparent to the user. The procedure has a library number outside the range of numbers available to the user.

As far as the user is concerned a knowledge of the content and use of this package is unnecessary and the only action required is on loading, whereby if requested by the Loader the floating point procedure must be loaded along with any other user library procedures.

It must be noted that any error or warning message will always be output to the teletype. (4.3)

The Floating Point Library Procedure, (TJF 16/2/76 Version), occupies approximately 360 program locations and 45 data locations, on each level that uses it.

2.4 INTERFACE WITH THE USER

2.4.1 COMMAND LANGUAGE

Communication between the operator and a program of the 920C CORAL Compiling System for selection of compiling options and input/output devices is by means of commands typed in at the teletype in response to an invitation to type from the compiler program.

There are two types of commands:

(1) OPTION commands.

The effect of an option command is to set up information for use by the compiler program, e.g. the input device. No action is taken by the compiler program other than remembering this information and replying with a further invitation to type upon which another option command can be typed as appropriate.

(2) ACTIVATION command.

There is only one activation command, i.e. GO), whose effect is to set the compiler program processing according to the options previously set up or the default values if no options have been specified. A further invitation to type will not be given by the compiler program until the process has been performed and therefore, until this time, no further command can be issued by the user.

Option commands may be given in any order between activation commands.

If an incorrect option command is typed it can be altered by typing the correct command (there is no means of cancelling a command - other than by the use of an illegal character before)).

A description of command errors and the user action required is described in 4.4.1.

2.4.2 COMMAND FORMAT

The invitation to type a command issued by the compiler program is an * at the start of a new line.

The format of an option command issued by the user is:

(1) A three character alphanumeric group.

(2) If an option is to be specified an =. (Typing of an option command without an option is equivalent to OPT=YES).

(3) If an option is to be specified, an option parameter, which will be:

- (a) A number, or
- (b) A device specifier, or
- (c) The words YES or NO.

For example to specify that the output is to be from the paper tape punch the option command:

```
* OUT=PTP↵
```

must be typed, and to initiate execution of the compiler program the activation command:

```
* GO↵
```

must be typed.

A description of all the commands and their associated parameters for each compiler program is provided with the respective operating instructions in Chapter 3.

2.5 MISCELLANEOUS NOTES

2.5.1 COMMON CHECKING

The following checks are performed by the Loader on the Common communicator and its associated segment(s) for the units of compilation of a program:

(1) The size of the runtime Common area is the same for all units of compilation.

(2) A Common label is only declared once.

(3) A Common switch is only declared once.

(4) A Common procedure is only declared once and all are declared.

There are no other checks performed on Common and it is the responsibility of the user to ensure that the same Common communicator is used with each unit of compilation of a CORAL program and that Common procedures are not shared between interrupt levels. It must be noted that the Loader only loads one of the Common areas it encounters - all other Commons are simply checked as described above.

CHAPTER 3

OPERATING INSTRUCTIONS

3 OPERATING INSTRUCTIONS

All programs comprising the 920C CORAL Compiling System are issued as paper tapes in standard 920C absolute binary format.

A detailed knowledge of Chapter 2, which provides a description of all constituents of the 920C CORAL Compiling System and the general mechanism of interface with the user, is assumed. This chapter simply provides basic operating instructions.

The basic operating instructions for each program are:

(1) Load program using hardware initial instructions. (Trigger to entry point using hand-keys if using the Loader since it is not self-triggering).

(2) Input option commands.

(3) Type GO↵ .

An expansion of this for each program is given below.

The option commands should have the following format:

"COMMAND" = "PARAMETER".

3.1 COMPILER PROGRAMS

This section provides the operating instructions for compiling, loading and executing a CORAL program using the Compiler Programs.

For a description of the Compiler Programs for program production, see 2.1.

3.1.1 MACRO PREPROCESSOR

For a description of the Macro Preprocessor, see 2.1.1.1.

3.1.1.1 Operating Instructions

(1) Load the Macro Preprocessor binary paper tape using the hardware initial instructions.

It is self-triggering. (If you wish to re-enter the macro-pass then trigger to 17736_8)

An * will be printed on the teletype as in invitation to type.

(2) Type the option commands on the teletype according to the requirements (3.1.1.2)

An * will be printed on the teletype as an invitation to type following each command.

(3) Place the first source paper tape to be processed in the paper tape reader.

(4) Type the activation command GO)

(a) If no checksum option was specified the source tape will be read and processed the expanded source output.

or

(b) If the checksum option was specified the source tape will be read and the message 'RELOAD TAPE' will be printed in which case repeat (3) and (4) whereby the source tape will be read, checksummed and processed, and the expanded source output.

(5) Repeat (3) and (4) for each source tape to be processed. Each source tape must end with 'HALT' (1.1.1.1) unless it is the final tape which ends with 'FINISH' and all tapes must terminate with a halt code (1.1.1.1). Following the processing of each source tape which terminates with 'HALT' the message 'LOAD NEXT TAPE' is printed on the teletype followed by an * as in invitation to type in order to initiate the processing of each subsequent tape.

MACRO PREPROCESSOR

3.1.1.1 (Contd) ('HALT' keywords are not transferred to the expanded source). The Macro Preprocessor halts following the processing of the source tape which terminates with 'FINISH' since at that point all processing is complete.

(6) For re-use of the Macro-Preprocessor goto (1).

MACRO PREPROCESSOR

For a description of the options see 2.1.1.2.

3.1.1.2 Option Commands

COMMAND	PARAMETERS	DEFAULT	MEANING
OUT			Output device for expanded source:
	PTP	PTP	Paper tape punch
	TTY		Teletype
	NUL		No source output required
ERR			Output device for error messages:
	PTP		Paper tape punch
	TTY	TTY	Teletype
	NUL		No error output required
CON			Conditional compilation request:
	YES		Conditional compilation required
	NO	NO	Conditional compilation not required.
	∇		≡ YES
CKS			Source checksum request:
	YES		Source checksum required
	NO	NO	Source checksum not required
	∇		≡ YES

3.1.2 PASS 1A

For a description of Pass 1A see 2.1.2.1.

3.1.2.1 Operating Instructions

(1) Load the Pass 1A binary paper tape using the hardware initial instructions.

It is self-triggering. (If you wish to re-enter Pass 1A then trigger to 17735_8).

An * will be printed on the teletype as an invitation to type.

(2) Type the option commands on the teletype according to the requirements (3.1.2.2).

An * will be printed on the teletype as an invitation to type following each command.

(3) Place the first source paper tape to be processed in the paper tape reader.

(4) Type the activation command GO

(a) If no checksum option was specified the source tape will be read and processed and the Pass 1A intermediate code output. No intermediate code will be output if it is small enough to be contained within core for transfer to Pass1B.

or (b) If the checksum option was specified the source tape will be read and the message 'RELOAD TAPE' will be printed in which case repeat (3) and (4) whereby the source tape will be read, checksummed and processed and the Pass 1A intermediate code output. No intermediate code will be output if it is small enough to be contained within core for transfer to Pass 1B.

(5) Repeat (3) and (4) for each source tape to be processed. Each source tape must end with 'HALT' unless it is the final tape which ends with 'FINISH' and tapes must terminate with a halt-code (1.1.1.1). Following the processing of each source tape which terminates with 'HALT' the message 'LOAD NEXT TAPE' is printed on the teletype followed by * as an invitation to type in order to initiate the

3.1.2.1 (contd)

processing of each subsequent tape. Pass 1A halts following the processing of the source tape which terminates with 'FINISH' since at that point all processing is complete.

(6) Since Pass 1A builds core resident compiler tables for use by Pass 1B, Pass 1A must be immediately followed by Pass 1B for this unit of compilation.

PASS 1A

For a description of the options see 2.1.2.2.

3.1.2.2 Option Commands

COMMAND	PARAMETERS	DEFAULT	MEANING
OUT			Output for Pass 1A intermediate code:
	PTP	PTP	Paper tape punch
	NUL		No intermediate code required
ERR			Output device for error messages:
	PTP		Paper tape punch
	TTY	TTY	Teletype
	NUL		No error output required
CON			Conditional compilation request:
	YES		Conditional compilation required
	NO	NO	Conditional compilation not required
	∇		≡ YES
CKS			Source checksum request:
	YES		Source checksum required
	NO	NO	Source checksum not required
	∇		≡ YES
LST			Output device for Pass 2 object map:
	PTP		*Paper tape punch
	TTY		Teletype
	NUL	NUL	No object map required
FWR			Floating warning request from Pass 1b:
	YES		Floating indication required
	NO	NO	Floating indication not required
	∇		≡ YES
SSS	Decimal	14710	Stack start address
SSL	number	1670	Stack length
SSD		400	Stack size difference

* If the object map is requested via the paper tape punch no relocatable binary will be output by Pass 2.

PASS 1A

3.1.2.2 (cont.) Note that the following are not allowed:

OUT = TTY — i.e. Pass 1A intermediate code cannot be output on the teletype

or OUT = PTP, ERR = PTP — i.e. Pass 1A intermediate code and error messages cannot be output on the same device

or OUT = NUL, ERR = NUL

3.1.3 PASS 1B

For a description of Pass 1B see 2.1.3.1.

3.1.3.1 Operating Instructions

N.B. If execution of Pass 1B does not immediately follow execution of Pass 1A for the current unit of compilation, and therefore the Pass 1A intermediate code does not correspond with the core resident compiler tables, execution of 1B will be undefined.

(1) Load the Pass 1B binary paper tape using the hardware initial instructions.

It is self-triggering. (If you wish to re-enter Pass 1B then trigger to 17735_8).
An * will be printed on the teletype as an invitation

(2) Place the Pass 1A intermediate code paper tape to be processed in the paper tape reader unless the information has been passed in core (3.1.2.1(5)).

(3) Type the activation command GO↵ . (There are no option commands for Pass 1B - those applicable from Pass 1A are used).

The Pass 1A intermediate code tape will be read and processed and the Pass 1B intermediate code will be output.

3.1.4 PASS 2

For a description of Pass 2 see 2.1.4.1.

3.1.4.1 Operating Instructions

N.B. If execution of Pass 2 does not immediately follow execution of Pass 1B for the current unit of compilation, and therefore the Pass 1B intermediate code does not correspond with the core resident compiler tables, execution of Pass 2 will be undefined.

(1) Load the Pass 2 binary paper tape using the hardware initial instructions.

It is self-triggering. (If you wish to re-enter Pass 2 then trigger to 17735_8).
An * will be printed on the teletype as an invitation to type.

(2) Place the Pass 1B intermediate code paper tape to be processed in the paper tape reader.

(3) Type the activation command GO↵. (There are no option commands for Pass 2 - those applicable from Pass 1A are used).

The intermediate code tape will be read and processed and the relocatable binary tape for the unit of compilation output.

Note: Wind the output tape up BACKWARDS in preparation for input to the Loader.

3.1.5 LOADER

For a description of the Loader see 2.1.5.1.

3.1.5.1 Operating Instructions

3.1.5.1.1 Normal Loading

Normal Loading is the loading of a single level (level 1) program consisting of one or more units of compilation using the non-reduced Loader.

(1) Load the Loader binary paper tape using the hardware initial instructions.

(2) Trigger to 4096_{10} using the hand-keys. (It is not self-triggering).

(3) Type the option commands on the teletype according to the requirements (3.1.5.2).

An * will be printed on the teletype as an invitation to type (if the reduced loader option is specified see 3.1.5.1.2).

(4) Place the first relocatable binary paper tape of the program in the paper tape reader. (Relocatable binary tapes for the units of compilation of a program may be loaded in any order excluding the library tape(s) which must be loaded last. The entry point of the program is assumed to be the first statement of the first segment loaded).

(5) Type the activation command GO↵.

The relocatable binary tape will be read. A number of core utilisation messages will be printed as the tape is processed (4.1.5.2) followed by an * as an invitation to type. (If the tape was not placed in the reader backwards the message 'INVALID TAPE' is printed on the teletype and (4) and (5) must be repeated).

(6) Repeat (4) and (5) for each relocatable binary tape of the program to be loaded and again for each relevant library tape until all library procedure calls are satisfied.

(7) Type the option command END ↵.

An * will be printed on the teletype as an invitation to type.

LOADER

3.1.5.1.1 (Contd) (8) Type the activation command GO↵.

The processing will be terminated and the message 'PROGRAM ENTRY entry point' will be printed specifying the program entry point in the current radix. An absolute binary paper tape will then be output on the punch if the option was specified in (3). In either case the object program will be resident in core awaiting execution.

N.B. The Loader cannot be rerun without reloading it.

3.1.5.1.2 Normal Loading by the Reduced Loader

This mode of operation is now obsolete.

LOADER

3.1.5.1.3 Multi-level Loading

Multi-level Loading is the loading of a multi-level program consisting of four or more units of compilation using the non-reduced loader.

(1) Load the Loader binary paper tape using the hardware initial instructions.

(2) Trigger to 4096_{10} using the hand-keys. (It is not self-triggering).

(3) Type the option commands on the teletype according to the requirements (3.1.5.2).

An * will be printed on the teletype as an invitation to type (if the reduced Loader option is specified see 3.1.5.1.4).

(4) Type the option command LEV=Level no↵ specifying the level upon which the following segments are to be loaded.

An * will be output on the teletype as an invitation to type.

(5) Place the first relocatable binary paper tape for that level of the program in the paper tape reader. (Relocatable binary tapes for the units of compilation of the program must be loaded together for each level but may be in any order within the level followed by the library tape(s) for that level. The entry point of a level is assumed to be the first statement of the first segment loaded on that level; the entry point of the program is assumed to be the entry point of level 1).

(6) Type the activation command GO↵ .

The relocatable binary tape will be read, a number of core utilisation messages will be printed as the tape is processed (4.1.5.2) followed by an * as an invitation to type.

3.1.5.1.3 (cont.)

(7) Repeat (5) and (6) for each relocatable binary tape of the current program level to be loaded and again for each relevant library tape until all library procedure calls are satisfied.

(8) Repeat (4) - (7) for each level. (The same library tape(s) may be read for each level and the Loader will automatically create a copy of each relevant procedure per level upon which it is used).

(9) Type the option command END↵ .

An * will be output on the teletype as an invitation to type.

(10) Type the activation command GO↵ .

The processing will be terminated and the message 'PROGRAM ENTRY entry point' will be printed specifying the program entry point in the current radix. An absolute binary paper tape will then be output on the punch if the option was specified in (3). In either case the object program will be resident in core awaiting execution.

N.B. The Loader cannot be run without reloading it.

3.1.5.1.4 Multi-Level Loading by the Reduced Loader

This mode of operation is now obsolete.

LOADER

For a description of the options see 2.1.5.2.

3.1.5.2 Option Commands

COMMAND	PARAMETERS	DEFAULT	MEANING
RAD	8_{10} $12_8 (=10_{10})$	8_{10}	User interface Input/Output number radix Octal Decimal
MØL	0-17777	1054	Module Ø lower bound
MØU	0-17777	17745	Module Ø upper bound
M1L	20000-37777	20000	Module 1 lower bound
M1U	20000-37777	34563	Module 1 upper bound
DMP	YES NO ∇	NO	Absolute binary dump required? Yes No ≡ YES
LEV	1-4 Ø	Ø	Loading level specification 1 level program (all segments on level 1)
END	YES NO ∇	NO NO	Last tape loaded? Yes No ≡ YES
AUT	YES NO ∇	NO	Self-triggering on loading? Yes No ≡ YES

All unsubscripted figures are in octal.

The Autostart facility is provided automatically, in Multi-level programs only. Single-level programs needing it should be loaded using LEV=1 rather than LEV=0. For it to function correctly AUT=YES must also be selected: if Self-Triggering is NOT actually required see 1.2.3.4.

3.1.6 OBJECT PROGRAM

After loading using the 920C CORAL Loader, the object program is resident in core awaiting execution.

To execute the object program in core:

(1) Trigger to the entry point using the hand-keys.

To execute the absolute binary object program:

(1) Load the object code absolute binary paper tape using the hardware initial instructions.

(2) If the AUT option was specified, i.e. Self-triggering , execution of the object program occurs following (1), otherwise trigger to the entry point provided using the hand-keys.

3.2 DIAGNOSTIC PROGRAMS

This section provides the operating instructions for the use of the Diagnostic Programs supplied with the 920C CORAL Compiling System.

For a description of the Diagnostic Programs see 2.2.

3.2.1 COMPILER DATA RETENSION

For a description of the Compiler Data Retension program see 2.2.1.1.

3.2.1.1 Operating Instructions

(1) Load the Compiler Data Retension Program binary paper tape using the hardware initial instructions.

It is self-triggering.

An * will be output on the teletype as an invitation to type.

(2) Type the option command on the teletype according to the requirements (3.2.1.2).

An * will be printed on the teletype as an invitation to type.

(3) If the option was RST place the paper tape containing the dumped compiler tables in the paper tape reader.

(4) Type the activation command GO↵ .

(a) If the DMP option was specified a paper tape containing the dumped compiler tables will be output on the punch.

or (b) If the RST option was specified the paper tape containing the dumped compiler tables will be read and reset in their original positions in the 920C core.

COMPILER DATA RETENSION

For a description of the options see 2.2.1.2.

3.2.1.2 Option Commands

COMMAND	PARAMETERS	DEFAULT	MEANING
DMP	YES NO ∇	NO	Compiler Table Dump required: Yes No ≡ YES
RST	YES NO ∇	NO	Compiler Table Restore required: Yes No ≡ YES

Obviously DMP and RST cannot be specified together.

3.2.2 OBJECT DUMP

For a description of the Object Dump Program see 2.2.2.1.

3.2.2.1 Operating Instructions

(1) Load the Object Dump Program binary paper tape using the hardware initial instructions.

It is self-triggering.

An * will be output on the teletype as an invitation to type.

(2) Type the option commands on the teletype according to the requirements (3.2.2.2).

An * will be output on the teletype as an invitation to type following each command.

(3) Type the activation command GO ↵.

The contents of the specified area of core will be output.

For a description of the format of the output see 4.2.2.

(4) For re-use of the Object Dump Program go to (2).

OBJECT DUMP

For a description of the options see 2.2.2.2.

3.2.2.2 Option Commands

COMMAND	PARAMETERS	DEFAULT	MEANING
OUT			Output device for dump
	PTP		Paper tape Reader
	TTY	TTY	Teletype
RAD			User Interface Input number radix
	8_{10}		Octal
	$12_8 (=10_{10})$	10_{10}	Decimal
MOD	0 - 5	0	Core Module No.
STA	$0\text{-}8191_{10}$	0	Module relative start address of core to be dumped
FNA	$0\text{-}8191_{10}$	0	Module relative finish address of core to be dumped

3.3 MISCELLANEOUS NOTES

The following is a collection of useful information which does not logically fit in any of the above sections.

3.3.1 PAPER TAPE OUTPUT SEPARATION

If the paper tape punch runs out during the output from the programs of the 920C CORAL Compiling System it is normally acceptable, although perhaps undesirable, to stop the program,runout some blanks, reload the punch, and continue thus separating the output onto two paper tapes. The following list describes where it is acceptable to split tapes:

	Output:			Tape can be split:
(1)	Macro Pass	-	source	Yes, i.e. blanks are not significant
(2)	Pass 1A	-	intermediate code	Yes
(3)	Pass 1B	-	intermediate code	Yes
(4)	Pass 2	-	relocatable binary	Yes
(5)	Loader	-	absolute binary	No, i.e. blanks are significant
(6)	Compiler Data-Retension		compiler tables	Yes
(7)	Object Dump	-	core dump	Yes

3.3.2 COMPILER DATA SPACE OVERFLOW

If compilation halts with errors 20 or 100, the 920C core store has become full on compilation due to the unit of compilation being too large or too complicated. The following suggestions may help to overcome the problem:

(1) Split the unit of compilation into several segments and compile them individually or if it is a Library unit split the procedures into several units.

(2) Reduce identifiers to between 1 to 3 characters, particularly procedure names if the error occurred in Pass 2. (This may easily be effected by redefining the names with macro definitions).

(3) Split any declaration which has a large preset list into several declarations thereby reducing the number of presets per declaration. (The split declarations may be overlaid with the original declaration for access).

(4) Remove presets altogether and initialise at runtime.

NOTE: If the store space is exhausted during execution of the Macro Pass the source probably contains a recursive macro call.

3.3.3 COMPILER INPUT CHECKSUM CALCULATION

Although the input to each compiler program is line buffered the checksum accompanying the input is not and it exists as the checksum of the whole tape. Warning 2 is produced if on reading the tape the calculated checksum disagrees with the value on the tape. However, since this check is not performed until the whole tape is read a mispunch may cause the compiler program to fail in an undefined way.

CHAPTER 4

DIAGNOSTIC OUTPUT

4 DIAGNOSTIC OUTPUT

This chapter describes all the diagnostic output produced by the programs of the 920C CORAL Compiling System, e.g. error messages, core maps, etc. An explanation of the method of production and use of the information from each program is described by the equivalent sections of Chapter 2.

4.1 COMPILER PROGRAMS

The format of an error/warning message from the Macro Pass, Pass 1A and Pass 1B is:

E/W na AT (nb : nc) (nd : ne)

where

na	Error/warning number	
nb	Number of line in which error starts	
nc	Number of character within line na where error starts	
nd	Number of line in which error finishes	Not present if a character error.
ne	Number of character within line nd where error finishes	

It must be noted that the line numbers produced by the Macro Pass refer to the original source and those produced by Pass 1A and Pass 1B refer to the expanded source from the Macro Pass. It is assumed that there are offline facilities for listing these files.

The format of an error message from Pass 2 is simply:

E na

The format of any other messages are described in the relevant sections below.

4.1.1 MACRO PREPROCESSOR

4.1.1.1 Error Messages

NUMBER	MEANING	RESULT
1	Parity error on input	Character ignored. Processing continues.
4	Invalid keyword	Character ignored until next symbol. Processing continues.
5	No (in 'LITERAL'	"
7	No) in 'LITERAL'	"
8	No (in 'OCTAL'	"
9	String too long, i.e. >630 characters	"
10	Invalid use of ↑	"
11	Invalid character after ! in string	"
12	Invalid number following 'X' non-printing character in string, i.e. not between 0 and 127.	"
13	No ! after non-printing character in string	"
14	No) in 'OCTAL'	"
15	Invalid character after ! in literal	"
16	No ! after non-printing character in literal	"
20	920C core store full on compilation - unit of compilation too large	Compilation halts (See 3.3.2)
21	A macro call with an incorrect number of parameters, i.e. it does not correspond to the definition	The macro is not expanded & the next symbol that is analysed & output is the first symbol following the next semi-colon

4.1.1.1 (cont.)

NUMBER	MEANING	RESULT
22	'DEFINE' or 'DELETE' is not followed by an identifier	No action is taken & no further output is produced until the next semi-colon has been read
23	Two identical formal parameters in the same macro definition	The definition is not accepted and no output is produced until the next semi-colon has been read
24	A macro definition or deletion is not followed by a semi-colon	This has no effect on the operation of the directive but no analysis is performed & no output produced until the next semi-colon symbol has been read
25	Request to delete a non-existent macro, or a macro that cannot be deleted because it is either an active macro or the name of a formal parameter of an active macro.	No action is taken. Processing continues.
26	Error in the format of an actual macro parameter	The macro is not expanded & no output or analysis occurs until the next semi-colon is read.
27	Error in the syntax of a macro definition, either the parameter list is wrong or the macro body is not present.	The definition is not accepted & processing is not resumed until the next semi-colon has been read.
28	Attempt to define a macro with the same name as an active macro or a formal parameter of an active macro	The definition is not accepted & processing is not resumed until the next semi-colon has been read.

4.1.1.1 (cont.)

NUMBER	MEANING	RESULT
100	A compiler data area is full - unit of compilation is too large or too complex	Compilation halts (See 3.3.2)
≥500	Compiler consistency error.	Compilation halts

MACRO PREPROCESSOR

4.1.1.2 Warning Messages

NUMBER	MEANING	RESULT
1	Illegal CORAL character	Character ignored, processing continues.
*2	Checksum failure on input (if option specified)	Processing continues
3	Input buffer full, i.e. more than 120 characters on a line	Processing continues. (A character may be lost).

* Output of warning 2 is often caused by the output of other error or warning messages even if there is no checksum failure.

4.1.2 PASS 1A

4.1.2.1 Error Messages

NUMBER	MEANING	RESULT
1	Parity error on input	Character ignored. Processing continues.
4	Invalid keyword	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
5	No (in 'LITERAL'	"
7	No) in 'LITERAL'	"
8	No (in 'OCTAL'	"
9	String too long, i.e. >630 characters	"
10	Invalid use of ↑	"
11	Invalid character after ! in string	"
12	Invalid number following 'X' non-printing character in string, i.e. not between 0 and 127	"
13	No ! after non-printing character in string	"
14	No) in 'OCTAL'	"
15	Invalid character after ! in literal	"
16	No ! after non-printing character in literal	"
20	920C core store full on compilation (compilation halts because unit of compilation is too large)	Compilation halts (See 3.3.2)

4.1.2.1 (cont.)

NUMBER	MEANING	RESULT
21	'FIXED' not followed by (	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
22	Specification of total bits not an integer	"
23	Specification of total bits not followed by a comma	"
24	Specification of fraction bits not a signed integer	"
25	'FIXED' specification not concluded by).	"
26	Exponent not a signed integer	"
27	'BYTE' not followed by 'ARRAY'	"
28	'CODE' not followed by 'BEGIN'	"
29	'BITS' not followed by [	"
30	Field width specification in 'BITS' operation not an integer	"
31	Field width specification in 'BITS' operation not followed by a comma	"
32	Bit position specification not an integer	"

4.1.2.1 (cont.)

NUMBER	MEANING	RESULT
33	'BITS' specification not terminated by]	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
34	Invalid symbol, commonly an error in a number format	"
35	Integer too large i.e. $<-131071_{10}$ $>+131071_{10}$ $>777777_{8}$ (See 1.1.3.3)	"
36	Total bits in 'FIXED' specification greater than 18, or in 'BITS' or 'UNSIGNED' specification greater than 17.	"
37	Fractional bits in 'FIXED'specification outside range $\pm$1023	"
38	Real number too large to hold in two-word floating format	"
39	Invalid CORAL structure	"
40	Variable not followed by assignment symbol where expected	"
41	'PROCEDURE' not followed by an identifier	"
42	'GOTO' not followed by an identifier	"

4.1.2.1 (cont.)

NUMBER	MEANING	RESULT
43	Switch or array has too many subscripts	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
44	Condition not followed by 'THEN'	"
45	'OVERLAY' not followed by an identifier	"
46	Right hand bracket of one-dimensional array element missing in overlay declaration	"
47	Base of overlay has more than one subscript or overlay declaraction does not contain 'WITH'	"
48	'SWITCH' not followed by an identifier	
49	'SWITCH' ID not followed by an assignment symbol	"
50	'TABLE' not followed by an identifier	"
51	'TABLE' ID not followed by [	"
52	Error in table size specification	"
53	Table size specification not concluded by]	"
54	Table declaration not followed by [	"
55	Error in preset constant list	"

4.1.2.1 (cont.)

NUMBER	MEANING	RESULT
56	Identifier list not present in declaration	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
57	Integer expected and not found	"
58	Subscripts not terminated by]	"
59	Procedure actual parameters not separated by commas, or no closing bracket on procedure call or declaration	"
60	Assignment symbol not present in 'FOR' statement	"
61	'DO' not present in 'FOR' statement	"
62	'UNTIL' not present in 'FOR' statement	"
63	Conditional expression does not contain 'THEN' in correct position	"
64	No 'ELSE' in conditional expression	"
65	Array declaration with incorrect bounds specification	"
66	Unmatched parentheses in expression	"
67	Incorrect structure following 'LOCATION'	"

4.1.2.1 (cont.)

NUMBER	MEANING	RESULT
68	Typed expression not enclosed in brackets	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
69	Invalid primary	"
70	Open square bracket missing in anonymous reference, or invalid structure to word reference	"
71	] missing in anonymous reference	"
72	Illegal syntax in code statement	"
73	Number type specification in procedure parameter specification neither followed by 'ARRAY' or 'PROCEDURE' nor preceded by 'VALUE' or 'LOCATION'	"
74	Error in specification of procedure parameter	"
75	Error in table field specification	"
76	Total bits not specified in table field specification	"
77	Table preset list not terminated by]	"
78	Too many arguments to operator. Most operators have a general limit of 31 arguments	"

4.1.2.1 (cont.)

NUMBER	MEANING	RESULT
79	Illegal bit position in table element or 'BITS' operator	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
80	Compilation unit name not present	"
81	'COMMON' ID not followed by (	"
82	'CORAL' not found at start of program	"
83	Source tape not terminated by 'FINISH'	"
84	Unrecognised syntax structure to program element	"
85	Common list not terminated by semi-colon	"
86	Un-named segment within compilation unit	"
87	Segment is not a block	"
88	'OVERLAY' ID 'WITH' not followed by declarations	"
89	Invalid table field description	"
90	Incorrect termination to table field	"
91	Unrecognised construction in library specifications	"
92	Unrecognised construction in declarations	"

4.1.2.1 (cont.)

NUMBER	MEANING	RESULT
93	Semi-colon missing before procedure body, or procedure body statement missing	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
94	Attempt to preset table specification	"
95	Illegal separator between CORAL structures	"
96	On compilation of a single segment source, the two occurrences of the segment name are different	"
97	Code statement used outside code block	"
98	Invalid CORAL compilation unit	"
100	A Compiler data area is full - unit of compilation is too large or too complex	Compilation halts (See 3.3.2)
152	Library procedure number cannot be 1	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
≥ 500	Compiler consistency error	Compilation halts

PASS 1A

4.1.2.2 Warning Messages

NUMBER	MEANING	RESULT
1	Illegal CORAL character in source	Character ignored. Processing continues
*2	Checksum failure on input (If option specified)	Processing continues
3	Input buffer full, i.e more than 120 characters on a line	Processing continues. (A character may be lost).

* Warning 2 is often caused by the output of other error or warning messages even if there is no checksum failure.

4.1.3 PASS 1B

4.1.3.1 Error Messages

NUMBER	MEANING	RESULT
1	Parity error in input	Character ignored. Processing continues.
20	920C core store full on compilation - unit of compilation is too large	Compilation halts. (See 3.3.2)
85	Common list not terminated by semi-colon	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
86	Un-named segment within compilation unit	"
87	Segment is not a block	"
88	'OVERLAY' ID 'WITH' not followed by declarations	"
89	Invalid table field description	"
90	Incorrect termination to table field	"
91	Unrecognised construction in library specifications	"
92	Unrecognised construction in declarations	"

4.1.3.1 (cont.)

NUMBER	MEANING	RESULT
93	Semi-colon missing before procedure body, or procedure body statement missing	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
94	Attempt to preset a table specification	"
95	Illegal separator between program elements	"
96	On compilation of a single segment source, the two occurances of the segment name are different	"
97	Code statement used outside code block	"
98	Invalid compilation unit. The source is not recognised as a CORAL compilation unit	"
100	A Compiler data area is full - unit of compilation is too large or too complex	Compilation halts (See 3.3.2)
102	Label declared twice at same block level	Characters ignored until next terminator which is usually the next semi-colon. Processing continues.
103	Identifier used as both label and variable within the same block	"
104	Variable declared twice at the same block level	"

4.1.3.1 (cont.)

NUMBER	MEANING	RESULT
105	Variable used prior to its declaration at the same block level	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
106	Undeclared variable has same name as a label used or declared in an outer block	"
107	The block nesting has been carried to more than 63 levels, or more than 62 levels within a procedure	"
108	A declaration of a prespecified entity does not agree with the specification	"
109	An answer is not given in a procedure that requires one	"
110	Nesting of procedures has been carried to more than 31 levels	"
111	A procedure has more parameters on declaration than it had on specification	"
112	In an array declaration, the lower bound is greater than the upper bound	"
113	An attempt has been made to overlay in a segment a variable declared in common	"
114	Attempt to overlay a table element	"

4.1.3.1 (cont.)

NUMBER	MEANING	RESULT
115	Attempt to overlay a two-dimensional Array	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
116	Attempt to overlay a procedure	"
117	Attempt to overlay a non-value procedure parameter	"
118	Attempt to overlay an out-of-bounds array element	"
119	Too many items in table preset list	"
120	Too many items in preset list	"
121	Table element is out of bounds	"
122	Attempt to preset where this is illegal	"
123	Declared procedure parameter is not the same as specified procedure parameter	"
124	A procedure has more parameters on specification than it had on declaration	"
125	A procedure has more than 30 parameters	"
126	A fixed procedure does not have the same scale on declaration as it had on specification	"

4.1.3.1 (cont.)

NUMBER	MEANING	RESULT
127	A procedure does not have the same type on declaration that it had on specification	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
128	A procedure specified as library is declared in a segment	"
129	An attempt is made to overlay a switch	"
130	An expression contains non-data references	"
131	Conflicting number of dimensions between array or switch references	"
132	A data reference is not found where a word or partword reference is required	"
133	A partword reference is found where a word reference is required	"
134	An arithmetic expression does not have an arithmetic value, probably because it contains references to labels, arrays, untyped procedures, etc.	"
135	A typed primary is not found where one is required	"
136	A floating type primary is used as an argument to a word logic operator or on the left hand side of a shift operator	"

4.1.3.1 (cont.)

NUMBER	MEANING	RESULT
137	The argument to 'AND' and 'OR' operators are not conditions	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
138	A non-value actual parameter to a procedure does not match the corresponding formal in type and scale	"
139	The number of formal parameters at a procedure declaration is not the same as the number of actual parameters used at a call.	"
140	The argument to a 'GOTO' is not a label or a switch element	"
141	An answer is given for a procedure that does not require one, or 'ANSWER' occurs outside a procedure.	"
142	The address field of a code statement is out of range	"
143	Illegal function field of a code statement	"
144	'IF' not followed by a condition	"
145	An identifier used as a procedure is not a procedure	"

4.1.3.1 (cont.)

NUMBER	MEANING	RESULT
146	The address part of a code statement is not a constant address	Characters ignored until the next terminator which is usually the next semi-colon. Processing continues.
147	An overlay declaration the base of which is an unsubscripted array would result in a negative overlay address	"
148	Library procedure number not found where required or found on a non-library procedure	"
149	Library procedure number too large	"
150	A compilation unit contains references to unset labels	"
151	A word reference is invalid	"
153	Fixed scale of procedure parameter at declaration does not correspond to scale at specification.	"
≥ 500	Compiler consistency error	Compilation halts

PASS 1B

4.1.3.2 Warning Messages

NUMBER	MEANING	RESULT
1	Illegal CORAL character in source	Character ignored. Processing continues.
*2	Checksum failure on input	Processing continues.
3	Incorrect tape	Processing continues.
4	Rescaling operation to floating point format invoked by Compiler (if option specified)	Processing continues.
5	A procedure is called within itself or within a nested procedure.	Processing continues

* Output of warning 2 is often caused by the output of other error or warning messages even if there is no checksum failure.

4.1.4 PASS 2

4.1.4.1 Error Messages

NUMBER	MEANING	RESULT
1	Parity error on input	Compilation halts
5	A shift instruction of more than 36 places generated on a normal shift, rescaling, multiplication or division operation	"
6	Overflow or underflow on fixing a real constant (Note that there is no error check for overflow or underflow on floating a fixed constant but this is unlikely to happen)	"
20	920C core store full on compilation - unit of compilation is too large	" (See 3.3.2)
2	A Compiler data area is full - unit of compilation is too large or too complex	" (See 3.3.2)
200	Checksum failure on input	"
203	Input to Pass 2 when errors in Pass 1	"
202	Consistency error.	"
≥ 500	Compiler consistency error.	

4.1.4.2 Object Map

The object map is an option which provides a map of the object code as it is produced from the Compiler. All addresses of object code items are given as addresses relative to their relevant absolute bases which are provided by the Loader when the object program is loaded (4.1.5.3).

The following information is produced:

(1) The name of the compilation unit:

```
    PROGRAM   name

or  LIBRARY   name
```

(2) The name of each section within the compilation unit:

```
    COMMON    name
or  SEGMENT   name
or  LIBPROC   name/no.
```

(3) The relative address of each data declaration preceded by 'D':

```
D   decimal     octal     data
    address     address   name
```

The address is relative to the segment data or common data base according to the type of data declaration.

It must be noted that the address output for a table element is always the address of word 0 of the table.

(4) The relative address of each label declaration preceded by 'L':

```
L   decimal     octal     label
    address     address   name
```

The address is relative to the segment code base in which the label is declared.

4.1.4.2 (cont.)

(5) The relative address of each switch array declaration preceded by 'S':

S	decimal address	octal address	switch name

The address is relative to the segment switch base in which the switch is declared.

(6) The relative address of the link and entry point of each common or internal procedure preceded by 'D' and 'P' respectively.

D	decimal address	octal address	procedure name	(link)

P	decimal address	octal address	procedure name	(entry point)

The link address is relative to the segment data or common data base according to the type of the procedure.

The entry point address is relative to the segment code base in which the procedure is declared.

It must be noted that for internal procedures the link and entry addresses are output on the procedure declaration whereas for a common procedure the link address is output on the specification and the entry address is output on the declaration. No such information is output on compilation of a Library procedure since they are handled differently by the Compiler and loader. The necessary information can be obtained from the Loader core utilisation information (4.1.5.2).

An example CORAL program with its corresponding object map is supplied in Appendix E.

4.1.5 LOADER

4.1.5.1 Error and Warning Messages

The following messages are printed on the teletype on occurrence of the respective error conditions when the normal loader is in use.

NUMBER	MESSAGE	MEANING	RESULT
1	COMMAND ERROR	An invalid message has been typed. (If the message is output after typed GO, it indicates that one of the commands previously input is invalid in a way that cannot be checked earlier).	Awaits input of correct command. (& type GO)
2	CHECKSUM FAILURE	Checksum failure on input of paper tape	Loading halts
3	THIS LEVEL ALREADY LOADED	An attempt is being made to load programs on a level on which loading has already been terminated	Awaits input of next tape
4	LIBRARY LOADED ON THIS LEVEL	An attempt is being made to load a program tape after a library tape has been loaded on the current level	The tape is rejected. Awaits input of next tape
5	NOT ALL LEVELS LOADED	An attempt is being made to terminate the loading of a	A new tape is requested

4.1.5.1 (cont.)

NUMBER	MESSAGE	MEANING	RESULT
		multilevel program before all four levels have been loaded	
6	INVALID TAPE	Probably tape is being loaded forwards instead of backwards	The tape is rejected. Awaits input of correct tape.
7	Name DECLARED TWICE (where name is the name of a Common Procedure	Common procedure has been declared twice	Declarations after the first are ignored
8	NO PROGRAM TAPE LOADED	An attempt is being made to load a library tape before any program tape(s) on the current level	The tape is rejected. Awaits input of program tape
9	COMMON SWITCH DECLARED TWICE	Common switch has been declared twice	Declarations after the first are ignored
10	COMMON CHECK FAILURE	Separate units of compilation do not have the same Common	Loading halts.
11	CORE FULL	If data is being loaded module 0 is full. If code is being loaded modules 0 and 1 are full.	Loading halts. (see 2.5.1)
12	COMMON LABEL DECLARED TWICE	Common label has been declared twice	Declarations after the first are ignored.

4.1.5.1
(Cont.)

NUMBER	MESSAGE	MEANING	RESULT
13	PARITY FAILURE	Parity error on input	Loading halts
22	UNDECLARED LIBRARY PROCEDURES	References to Library Producedures not yet loaded	Loading continues
23	UNDECLARED COMMON PROCEDURES	References to Common procedures not yet loaded	Loading continues

LOADER

4.1.5.2 Core Utilisation Information

The following information is output by the normal loader as each unit is loaded:

CORE BOUNDS MOL MOU M1L M1U
UNIT NAME

For each Common and Library procedure declared:

Absolute address of ENTRY POINT | Absolute address of LINK | PROCEDURE NAME

The loading base names and their absolute addresses:

COMMON — Absolute address of BASE

DATA — "

CODE — "

SWITCH — "

The name of each section within the unit (in reverse order to compilation):

SEGMENT name N
:
SEGMENT name 1
COMMON name

At the end of loading each unit, any unsatisfied external procedure references are indicated by the respective messages:

UNDECLARED COMMON PROCEDURES
UNDECLARED LIBRARY PROCEDURES

Following the loading of all the units of compilation of the program, i.e. after typing the END directive, the program entry point is printed:

PROGRAM ENTRY absolute address
CORE BOUNDS MOL' MOU' M1L' M1U'

For an example see Appendix E.

4.2 DIAGNOSTIC PROGRAMS

4.2.1 COMPILER DATA RETENSION

There are no error or warning situations other than the standard error messages for incorrect user commands(4.4.1).

4.2.2 OBJECT DUMP

There are no error or warning situations other than the standard error messages for incorrect user commands(4.4.1).

The format of the output of the object dump program for the contents of each core location is:

Absolute address in decimal	Absolute address in octal	Contents in decimal	Contents in octal	Contents in instruction form

4.3 FLOATING POINT LIBRARY PROCEDURE

ERROR MESSAGE	MEANING	RESULT
QF2	Floating point underflow, i.e. computed exponent <-64	Execution continues with floating value of zero (smallest value)
QF3	Floating point overflow, i.e. computed exponent >63 (including division by 0)	Execution continues with floating value of $\pm 9 \cdot 2 * 10^{18}$ according to sign (largest value)
QF4	Overflow on fixing a floating number	Execution continues with the largest positive (01...1) or the smallest negative (10...0) according to the scale and the sign.
QF5	Underflow on fixing a floating number	Execution continues with the value of zero.

The above error messages are followed by the Octal Absolute address of the Floating Point interpreted instruction which gave rise to the error.

These error messages are always output to the teletype.

4.4 MISCELLANEOUS NOTES

4.4.1 COMMAND ERRORS

Within each program of the 920C CORAL Compiling System detection of an error in a user input command causes the following:

(1) Output of COMMAND ERROR if a syntatically incorrect command is input.

(2) Output of COMMAND ERROR after GO↵ is typed if a syntatically correct command has been input but the command is not applicable.

(3) Output of DEVICE SPECIFICATION ERROR after GO↵ is typed if the device specified with a command is not suitable.

In each case an invitation to type is re-issued from the Compiler Program for input of the correct command followed by GO↵ if applicable. Errors of type (2) and (3) cause repeated output of the respective message following each option command input until the correct command is input.

CHAPTER 5

OBJECT CODE STRATEGY

5 OBJECT CODE STRATEGY

The aim of the object code strategy is to provide tight, efficient code which works quickly.

This chapter may be ignored by the normal user since all necessary user information is provided in Chapter 1. However, if the actual structure of the object code is of importance to the user a general description is provided by this chapter.

It must be noted that although the object code is structured to run on a 920C or 905 it will also execute on any 920B, 920M or 903 such that the whole program resides in module 0, i.e. no 'set absolute' address mode setting instructions are generated unless the Loader is instructed to store programs above 8K and none of the instructions beyond 15 7168, which earlier machines would interpret as a terminate, or sequences assuming the preservation of the Q register after all jumps, are generated.

Any limits imposed by the object code strategy on the user are described in 1.1.1.4.

5.1 RUNTIME STORAGE ALLOCATION

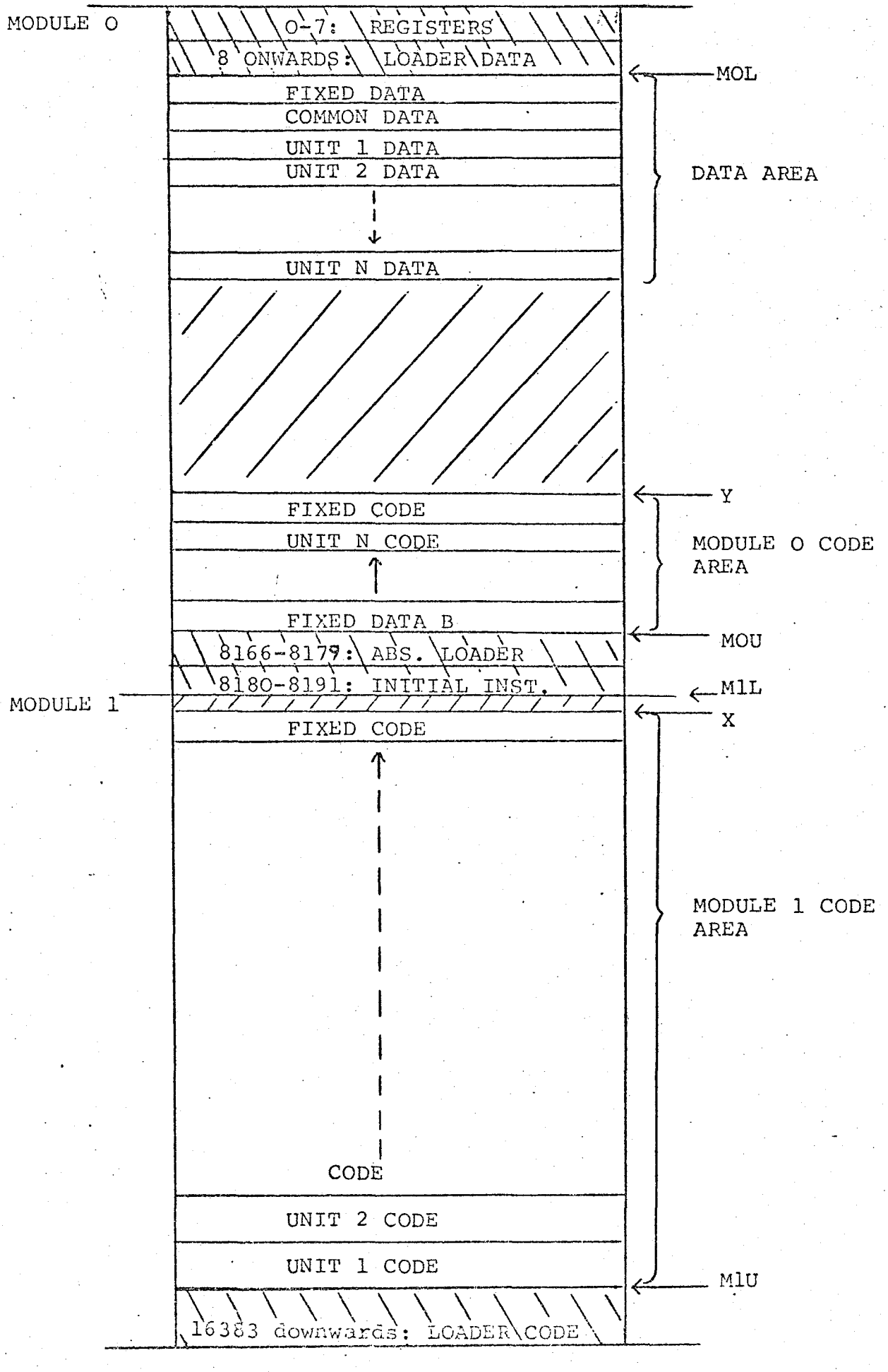
5.1.1 DESCRIPTION

The object code of a CORAL program contains data and executable code which are held statically at runtime in the 920C core store.

Data resides contiguously in core store in module 0 and is absolutely addressed and code resides in both modules 0 and 1.

Each unit of compilation comprising a program has a data and code area associated with it.

The following diagram describes the runtime storage allocation of a program comprising a number of units of compilation.


MODULE 0
0-7: REGISTERS
8 ONWARDS: LOADER DATA
MOL
FIXED DATA
COMMON DATA
UNIT 1 DATA
UNIT 2 DATA
DATA AREA
UNIT N DATA
Y
FIXED CODE
UNIT N CODE
MODULE 0 CODE AREA
FIXED DATA B
MOU
8166-8179: ABS. LOADER
8180-8191: INITIAL INST.
M1L
MODULE 1
X
FIXED CODE
MODULE 1 CODE AREA
CODE
UNIT 2 CODE
UNIT 1 CODE
M1U
16383 downwards: LOADER CODE

5.1.1
(cont.) The Loader loads the object program within the module bounds:

MOL	:	Module 0	Lower Bound, i.e. lowest available location in Module 0
MOU	:	Module 0	Upper Bound, i.e. highest available location in Module 0
M1L	:	Module 1	Lower bound
M1U	:	Module 1	Upper bound

For the default values of these and the method of specifying different values see 2.1.2.2 and 3.1.2.2.

The data area for each unit of compilation is loaded from MOL upwards - the Common data area loaded is that which accompanies the first unit of compilation being loaded and it is assumed that all subsequent Common data areas accompanying the following units of compilation are the same (a limited number of checks are performed by the Loader and are described in 2.5.1).

The code area for each unit of compilation is loaded from M1U downwards in module 1 until the remaining available space within the specified bounds is too small for the current unit whereupon it is loaded into module 0 - the remaining space in module 1 being used for a subsequent unit if possible. The Loader remembers the available space in each module and only when the code area of a unit of compilation will not fit in either module does it report that the core is full.

5.1.2 LOADER GENERATED INFORMATION

The fixed data areas are generated by the Loader above and below the module 0 lower and upper bounds respectively and contain information for use by the object program or the user, i.e.

FIXED DATA AREA A

MOL	Absolute address of level 1 entry	
MOL+1	Absolute address of level 2 entry	
MOL+2	Absolute address of level 3 entry	
MOL+3	Absolute address of level 4 entry	
MOL+4	Top level indicator (1.2.3.1)	Preset to 0
MOL+5	Low level indicator (of no use)	Preset to 0
MOL+6	Standard constants for use by the object program ↓	
MOL+N		

FIXED DATA AREA B

MOU	Module 1,upper code bound, M1U
MOU-1	Module 1,lower code bound, X
MOU-2	Module 0,upper code bound, MOU
MOU-3	Module 0,lower code bound, Y
MOU-4	-1
MOU-5	-Sumcheck (1.2.3.2)

The fixed code area is generated by the Loader below the executable object code in each core module if required by the object program and contains inter-module and multi-level housekeeping code. (It is not for access by the user).

5.2 DATA SPACE ALLOCATION

The data area of a unit of compilation contains CORAL data (declared data and procedure parameters) which is overlayed according to the block structure and compiler generated data (strings, constants, addresses, linkage information and workspace).

The data space allocation for each item of CORAL data is described below.

5.2.1 DATA DECLARATIONS, e.g. 'INTEGER' I;

5.2.1.1 Data Types

Internal Data : Internal data declaration space is held in the data area of the unit of compilation.

Common Data : Common data declaration space is held in the Common data area.

5.2.1.2 Space Allocation

DECLARATION	NO. WORDS	REFERENCE
INTEGER	1	(1.1.3.3)
FIXED	1	(1.1.3.2)
FLOATING	2	(1.1.3.1)

5.2.2 ARRAY/TABLE DECLARATIONS, e.g. 'FLOATING''ARRAY' A[1:10];

5.2.2.1 Array Types

Internal Array : Internal array space is held in the data area of the unit of compilation.

Common Array : Common array space is held in the Common data area.

5.2.2.2 Space Allocation

Space for the elements of an array is allocated in a contiguous area, the rows of a two-dimensional array following on immediately from one another and they therefore do not necessarily start on a word boundary for bit and byte arrays. The first element of all arrays always starts on a word boundary.

The following space is allocated for an element of each type of array:

DECLARATION	NO. WORDS	REFERENCE
INTEGER	1	(As in 5.2.1.2)
FIXED	1	(As in 5.2.1.2)
FLOATING	2	(As in 5.2.1.2)
BIT	1/16	(1.2.2)
BYTE	1/2	(1.2.2)
TABLE -		
INTEGER ELEMENT	1	(As in 5.2.1.2)
FIXED ELEMENT	1	(As in 5.2.1.2)
FLOATING ELEMENT	2	(As in 5.2.1.2)
PARTWORD ELEMENT	AS DECLARED	(1.1.4)

5.2.3 PROCEDURE DECLARATIONS, e.g. 'PROCEDURE'PROC (Parl, Par2,....ParN);

5.2.3.1 Procedure Types

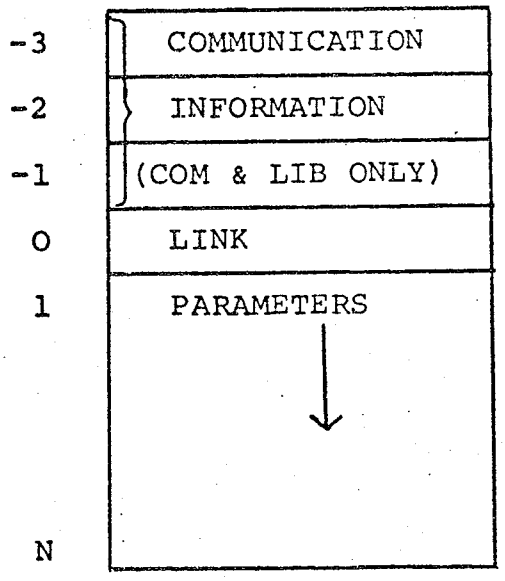
Internal Procedure : The link and parameter space is held in the data area of the unit of compilation.

Common Procedure : The link and parameter space, together with further information for external communication, is held in the Common data area.

Library Procedure : The link and parameter space, together with further information for external communication, is held in the data area of the unit of compilation in which the Library procedure is first referenced (first reference loaded).

5.2.3.2 Space Allocation

Space for the link, parameters and communication information if present is allocated as:



5.2.3.2 (cont.) The following space is allocated for individual parameters:

PARAMETER	NO. WORDS		CONTENTS
VALUE	As in 5.2.1.2		Value of parameter
LOCATION	1		Absolute address of parameter
ARRAY/TABLE	3	0	Object code Communication word
		1	Absolute address of zeroith element of array
		2	Row length of array if two dimensional
PROCEDURE	2	0	Absolute address of link
		1	Absolute address of entry point
LABEL	1		Absolute address of label

5.2.4 LABEL DECLARATIONS, e.g. LAB1:

Internal Label : No data space is allocated.

Common Label : A word containing the absolute address of the label is held in the Common data area.

5.2.5 STRINGS, e.g. ↑<STRING>↑

Although strings are not declared they are held in the data area of the unit of compilation in which they are used. For a description of their storage see 1.1.13.3.

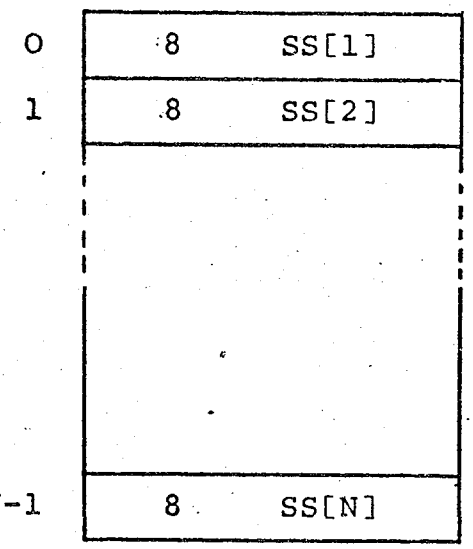
5.3 CODE SPACE ALLOCATION

The code area of a unit of compilation contains switch arrays and executable object code for the CORAL statements.

5.3.1 SWITCH DECLARATIONS, e.g. 'SWITCH' SS:=S1,S2,S3;

A switch array is held as a dispatch table of jump instructions to either the switch element, if within the same unit of compilation, or to a pair of instructions for transferring control to the switch element, if in a different unit of compilation:

i.e.



All switch arrays declared within a unit of compilation are held above (high address end) of the executable object code for that unit within the code area.

5.3.2 STATEMENTS

Each CORAL statement is made up of one or more operations each of which produces an executable object code sequence which occupies a finite number of words. Generally there is more than one sequence which can be generated for a particular operation according to its environment. A description of the structure of the executable object code is described in 5.4.

5.4 EXECUTABLE OBJECT CODE

5.4.1 GENERAL OBJECT CODE SEQUENCES

The following description lists the object code sequences generated for all CORAL operations. Only the general sequence for each operation is provided and it must be treated purely as a guide to the structure of the object code since in practice sequences are often modified according to the optimisation being performed (5.4.3) and the environment, e.g. it is sometimes necessary to evaluate a complicated argument into working space prior to its use. It must be emphasised that the aim has been to produce tight, efficient code (without subroutining) which works quickly, within the limitations of the hardware, although by only describing the general sequences this may not always appear so. However, it is beyond the scope of this manual to provide any more detailed information and it is hoped that the following description will be useful.

A knowledge of the 920C order code and a general knowledge of the 920C SIR assembly code is assumed since the sequences are described in this form.

A number of further symbols are used for clarity:

f : Function being performed, e.g. f=1 for addition

+X : Absolute address of X

ws : Workspace

wsX: Workspace to contain X

5.4.1.1 Data Reference

This section describes the sequences generated for each type of data access. For the use of each item there are normally two associated sequences:

(1) Accessing the VALUE of the data item for use in an expression.

(2) Accessing the ADDRESS of the data item for use in a 'LOCATION' expression and on the left-hand side of an assignment statement.

OPERATION	SEQUENCE
Access of an actual declared variable or a formal by value parameter:	
VALUE of V	f V
ADDRESS of V	f +V
Access of a formal by location parameter:	
VALUE of V	o V /f o
ADDRESS of V	f V
Access of an actual or formal whole word array or table element:	
VALUE of A[index]	Evaluate index into accumulator 1 +A[o] 5 B register /f o
ADDRESS of A[index]	Evaluate index into accumulator 1 +A[o]

OPERATION	SEQUENCE
Access of an actual or formal bit or byte array element: VALUE of A[index] (bit)	Evaluate index into accumulator 14 8188 1 +A[o] 5 B register 4 +o 14 4 5 ws /4 o o ws /14 8177 6 +1
VALUE of A[index] (byte)	Evaluate index into accumulator 14 8191 1 +A[o] 5 B register 4 +o 14 1 5 ws /4 o o ws /o shift modifier /14 8183 6 +511
ADDRESS of A[I]	Use in 'LOCATION' expression illegal. For use on left-hand side of assignment - see 5.4.1.2
Access of an actual or formal partword table element (For a description of the result of a partword table element access see 1.1.4.3): VALUE of A[index]	Evaluate index into accumulator 1 +A[o] 5 B register /4 o 14 shift value (14 shift value - only for sign regeneration) 6 mask value
ADDRESS of A[index]	Use in 'LOCATION' expression illegal. For use on left-hand side of assignment see 5.4.1.2.

OPERATION	SEQUENCE
Access of a partword. (For a description of the result of a partword access see 1.1.6.3).	
VALUE of 'BITS'[x,y]V	4 V 14 shift value 6 mask value
ADDRESS of 'BITS'[x,y]V	Use in 'LOCATION' expression illegal. For use on left-hand side of assignment see 5.4.1.2.
Access of an anonymous reference:	
VALUE of [V]	o V /f o
ADDRESS of [V]	f V
Access of a constant:	
VALUE of ± k	f ± k
ADDRESS of ± k	Illegal

5.4.1.2 Assignment Statements

The section describes the general sequences generated for each type of assignment statement.

OPERATION	SEQUENCE
Assignment to a whole word: V:=X	4 X 5 V
Assignment to a bit or byte array element: A[index]:= X	Evaluate index into accumulator 14 8188 - if bit 8191 - if byte 1 +A[o] 5 ws1 4 +o 14 4 - if bit 1 - if byte 5 ws2 o ws1 /4 o o ws2 /6 mask value 5 ws3 4 X o ws1 /o shift value /14 o o ws2 /6 mask value 1 ws3 o ws2 /5 o
Assignment to an actual or formal whole word array or table element A [index] := X	Evaluate index to accumulator 1 +A[O] 5 1 4 X /5 O

OPERATION	SEQUENCE
Assignment to a partword table element: A[index]:=X	Evaluate index into accumulator 1 +A[o] 5 ws1 o ws1 /4 o 6 mask value 5 ws2 4 X 14 shift value 6 mask value 1 ws2 o ws1 /5 o
Assignment to a partword: 'BITS'[x,y]V:=X	4 V 6 mask value 5 ws1 4 X 14 shift value 6 mask value 1 ws1 5 V

5.4.1.3 Dyadic Operators

This section describes the general sequences generated for the operations: +, -, *, /, MASK, UNION, DIFFER, LEFT AND RIGHT.

It must be noted that if any argument of an operator requires rescaling (1.1.9.2), other than those of * or / which automatically contain their rescaling operations, the rescaling is performed before the use of the argument within the sequence.

i.e. evaluate argument into accumulator
14 shift value
(6 mask value)

OPERATION	SEQUENCE
ADD: A1 + A2	4 A1 1 A2
SUBTRACT: A1 - A2	4 A2 2 A1
MULTIPLY: A1 * A2 (including any necessary rescaling).	4 A1 12 A2 14 shift value
DIVIDE: A1 / A2 (including any necessary rescaling)	4 A1 0 +0 14 shift value 13 A2 (14 8191) or 4 A2 0 +0 14 shift value 5 ws 4 A1 13 ws (14 8191)

OPERATION	SEQUENCE
MASK: A1 'MASK' A2	4 A1 6 A2
UNION: A1 'UNION' A2	4 A2 2 -1 5 ws 4 A1 2 -1 6 WS 2 -1
DIFFER: A1 'DIFFER' A2	4 A2 6 A1 14 1 6 777776_8 2 A2 1 A1
LEFT: A1 'LEFT' A2	0 A2 4 A1 2 +0 2 +0 } Clear Q /14 0
RIGHT: A1 'RIGHT' A2	4 A2 2 +0 5 B register 4 A1 /14 0

5.4.1.4 For Statements

This section describes the general sequences generated for each type of FOR statement.

OPERATION	SEQUENCE
Simple forelement: 'FOR' A1:=A2 'DO'...	4 A2 5 A1 'DO' statement
While forelement: 'FOR' A1:=A2 'WHILE'A3'DO'...	4 A2 5 A1 REP: evaluate condition A3 and jump to TRUE or FALSE accordingly TRUE: 'DO' statement 8 REP FALSE:
Step forelement: 'FOR' A1:=A2 'STEP'A3'UNTIL' A4'DO'...	4 A2 5 A1 4 A3 5 wsA3 4 A4 5 wsA4 REP: 4 A1 2 wsA4 12 wsA3 9 FALSE 'DO' statement 4 wsA3 1 A1 5 A1 8 REP FALSE:

5.4.1.5 Conditions

This section describes the general sequences generated for:

Relational operators - EQ,NOTEQ,GR,GE,LT,LE
Boolean operators - AND,OR
Conditional expressions and conditional statements

OPERATION		SEQUENCE	
EQ:	A1 = A2	4 2 7 8	A1 A2 TRUE FALSE
NOTEQ:	A1 ≠ A2	4 2 7 8	A1 A2 FALSE TRUE
GT:	A1 > A2	4 2 9 8	A1 A2 TRUE FALSE
GE:	A1 ≥ A2	4 2 9 8	A2 A1 FALSE TRUE
LT:	A1 < A2	4 2 9 8	A2 A1 TRUE FALSE
LE:	A1 ≤ A2	4 2 9 8	A1 A2 FALSE TRUE

OPERATION	SEQUENCE
AND: cond1 'AND' cond2 'AND' cond3 .... (can be combined with ORs)	evaluate cond 1 TRUE(1): evaluate cond 2 : TRUE(N-1):evaluate cond N TRUE(N): consequence 8 END FALSE(1→N): alternative END:
OR: cond1 'OR' cond2 'OR' cond3... (can be combined with ANDs)	evaluate cond 1 FALSE(1): evaluate cond 2 : FALSE(N-1):evaluate cond N TRUE(1→N): consequence 8 END FALSE(N): alternative END:
Conditional expression/ statement: 'IF' cond 'THEN' consequence 'ELSE' alternative	evaluate condition and jump to TRUE or FALSE TRUE: consequence 8 END FALSE: alternative END:

5.4.1.6 Procedure Handling

This section describes the general sequence generated for procedure declarations and calls.

OPERATION	SEQUENCE
Procedure call: PROC (Par1, Par2,...., ParN)	evaluate Par1 into acc 5 param space evaluate Par2 into acc 5 param space : evaluate ParN into acc 11 LINK } call 8 ENTRY } PROC
Procedure declaration: 'PROCEDURE'PROC (Par1,Par2,..., ParN)	5 last param space parameter housekeeping code procedure body : EXIT: 0 LINK /8 1
Answer statement: 'ANSWER' A	4 A 8 EXIT

5.4.1.7 Label and Switch Handling

This section describes the general sequences generated for label and switch access.

OPERATION	SEQUENCE
Label access: 'GOTO' L	8 L
Switch access: 'GOTO' SS|N|	4 N (cause 1 +SS[o] transfer to 8 x switch via a Compiler generated fixed sequence)

5.4.2 FLOATING POINT HANDLING

The standard Elliott 920C Floating Point package, QF, has been used as a basis for the object code floating point package (2.3).

Any floating operation produces object code which is interpreted by the Compiler Floating Point Library Package, CAPQF, (which is supplied in relocatable binary form).

The floating point object code sequences are basically similar to integer and fixed point sequences (5.4.1) with entries to, and exits from, the floating point package generated according to the following rules:

(1) Access of a floating item is preceded by an entry to QF it not currently in floating mode.

(2) Access of a non-floating item is preceded by an exit from QF if currently in floating mode.

(3) Instructions 7,8,9,11 and 15 cause automatic exit from floating mode prior to execution.

It must be noted that there has been no attempt made to especially optimise the object code for floating operations and therefore certain constructs are relatively inefficient.

5.4.3 OPTIMISATION AND THE PRODUCTION OF EFFICIENT CODE

This section describes the main features of the optimisation performed by the 920C CORAL Compiler, together with notes for the user on the production of efficient object code from a CORAL program.

It is not within the scope of this manual to provide an exhaustive description of the object code but it merely presents a guide to the structure of the code and notes on any specific important optimisations.

A detailed knowledge of 5.4.1 is assumed since the descriptions of the optimisations are presented as the effects on the general code sequences.

In the following description frequent reference is made to Chapter 1 where the structure of the object code which directly affects the user is described.

5.4.3.1 Evaluation of Expressions

(1) Re-ordering of Arguments

This is by far the most important optimisation performed by the Compiler in that it has the most significant effect.

In general the Compiler re-organises the arguments of operations to produce more efficient code.

e.g. A + B'MASK'C is evaluated as B'MASK'C + A

Unoptimised x		Optimised ✓	
4	A	4	B
5	ws	6	C
4	B	1	A
6	C		
1	ws		

This is mainly applied to expressions but is also applied to certain other types of operation. In the above example, if A,B and C are function calls the optimisation is not performed in order to adhere to the Official Definition (1.1.9.6 and OD 6.1.3).

It must be noted that the Compiler does not extract and evaluate sub-expressions which are used several times. It is the responsibility of the CORAL programmer to ensure that such expressions are only evaluated once into a temporary workspace and it is that which is used subsequently in place of the expression.

e.g.
```
A:= B*C + D;
E:= F/(B*C) + (B*C)/G;
```

should be written as:

```
X:= B*C;
A:= X+D;
E:= F/X + X/G;
```

Similarly, repeated use of the same partword should be avoided.

5.4.3.1
(cont.)

(2) Rescaling

Evaluation of expressions and assignment statements with arguments of different scales will automatically produce rescaling code sequences (1.1.9.2) which obviously degrade the efficiency of the object code. Integer or fixed point working is more efficient than floating point working and in general it is most efficient to use variables of the same scale.

(3) Compile Time Arithmetic

See 1.1.9.3.

(4) Multiplication and Division

The object code for division is relatively inefficient due to the structure of the hardware and the necessity to maintain accuracy (1.1.9.5).

An additional feature is provided which maintains a double precision intermediate result between a multiplication operation which is immediately followed by division with arguments of the same scale (1.1.9.2.).

Multiplication and division by constant values which are powers of two are not optimised into the equivalent shift instructions.

(5) Rounding

See 1.1.9.5.

(6) Function calls within expressions as value parameters

See 1.1.9.6.

5.4.3.2 Low Level Instruction Optimisation

(1) A and B Register Optimisation

The Compiler 'remembers' the contents of the hardware and floating point (software) accumulator and B register when they contain simple data or constant values and redundant instructions which reload these registers are usually avoided.

e.g.
```
A:=B;
C:=A + D;
```

Unoptimised x		Optimised ✓	
4	B	4	B
5	A	5	A
4	A	1	D
1	D	5	C
5	C		

It must be noted that the Compiler only remembers the 'latest' contents of the register.

e.g.
```
A:=B  will generate    4  B (Acc = B)
C :=A                  5  A (Acc = A)
                       5  C (Acc = C)
```

but
```
A:=B  will generate    4  B (Acc = B)
C:=B                   5  A (Acc = A)
                       4  B (Acc = B)
                       5  C (Acc = C)
```

A change to or from floating point mode will automatically cause the Compiler to 'forget' the contents of the hardware and floating point registers.

(2) Shift and Mask Optimisation

Redundant shift and mask instructions within rescaling, mulitplication, division and partword handling operations are not generated.

Similarly, consecutive shift and mask instructions are combined (other than those generated from a user 'RIGHT' operation which is generated as specified since it may be written for sign regeneration).

5.4.3.2
(cont.)

This is particularly important when both the left and right-hand sides of an assignment statement are partwords.

e.g. 'BITS'[3, 1]A:= 'BITS'[3,15]B

Unoptimised x			Optimised ✓	
4	A		4	A
6	$+777761_8$		6	$+777761_8$
5	ws		5	ws
4	B	Access part-word	4	B
14	8177		14	8178
6	+7		6	$+16_8$
14	1	Assignment to part-word	1	ws
6	$+16_8$		5	A
1	ws			
5	A			

(3) Jump Optimisation

A jump instruction to a compiler generated label which labels a jump instruction to a source label causes the latter not to be generated and the former to be generated as a jump instruction to the source label.

e.g. 'IF' A=B 'THEN' 'GOTO' X 'ELSE' 'GOTO' Y

Unoptimised x			Optimised ✓	
	4	B	4	B
	2	A	2	A
	7	T	7	X
	8	F	8	Y
T:	8	X		
	8	E		
F:	8	Y		
E:				

5.4.3.2
(cont.)

Also, a jump instruction to a compiler generated label on the next instruction is removed.

e.g.
```
'PROCEDURE' PROC;
    'BEGIN'
       PROC BODY
          ¦
          ¦
       'ANSWER' A
    'END';
```

Unoptimised x

```
  PROC body
     ¦
     ¦
     ¦
     4     A
     8     EXIT
(EXIT:) 0  LINK
    /8     1
```

Optimised ✓

```
  PROC body
     ¦
     ¦
     ¦
     4     A
(EXIT:) 0  LINK
    /8     1
```

5.4.3.3 Data Access

(1) Data Overlaying

Data is overlayed according to the block structure of the CORAL language (OD 3) However, it is recommended that the 'OVERLAY' declaration facility be used to economise even further on data space or to add clarity to a program (1.1.5 and OD 4.8).

(2) Array Access

Access of an array element (other than bit or byte) with a constant index between 0 and 8191 causes the generation of an optimised code sequence.

e.g. Access of A[6]

Unoptimised x		Optimised /	
4	+6	0	+A[0]
1	+A[0]	/f	6
5	B register		
/f	0		

This is the only optimisation performed on array access and it must be recognised by the user that array access with a variable index, particularly two-dimensional, is inefficient.

Bit/byte array access is necessarily inefficient and unless a large quantity of such data is required it is recommended that it is held in a different form, i.e. within a whole word array or a table. Alternatively in some cases it may be possible to overlay the array with a table with elements of equivalent structure to increase efficiency - however this may well not be convenient.

(3) Anonymous Reference

As with array access the use of an anonymous reference with a constant index between 0 and 8191 causes the generation of an optimised code sequence.

5.4.3.3
(cont.)

e.g. Access of [6]

Unoptimised x	Optimised ✓
0 +6	ŧ 6
/f 0	

(4) Partword Access ('BITS' or partword table element)

As for bit and byte array access but to a lesser degree partword access is necessarily inefficient and should be avoided as far as possible. Unless a large quantity of such data is required it is recommended that it is held in a whole word form. Alternatively a set of partword information should be unpacked into whole word working space before use and repacked into the partword form after use - both operations being performed by general purpose procedures.

As stated previously the Compiler will not generate redundant shift or mask instructions.

5.4.3.4 Miscellaneous Optimisations

(1) Assignment Statement

The 920C increment instruction is used in a simple assignment statement if applicable.

e.g. A:= A + 1;

Unoptimised x		Optimised ✓	
4	A	10	A
1	+1		
5	A		

(2) For Statements

If there is only one FOR element the 'DO' statement is obeyed inline otherwise it is made into a subroutine which is generated following the code for the FOR statement (a call to the subroutine being generated in the position of the 'DO' statement within the FOR statement code).

The following notes apply to a FOR statement with a STEP element:

'FOR' A1:=A2 'STEP' A3 'UNTIL A4 'DO' ...

(a) If A3 and A4 are constants they are not evaluated into working storage but are used directly.

(b) If A3 is a constant the sequence for determining exhaustion of the loop is optimised to a simple subtraction since only the sign is required.

(c) If A3 is the constant +1 the 920C increment instruction, 10, is used for the updating of A1.

(3) Procedure Parameters

On a procedure call parameters are set up in the procedure parameter space before transferring control to the procedure except for the last parameter which is passed via the accumulator. Single parameter procedures are therefore relatively efficient and care should be taken to ensure that further parameters are necessary - Common data space being an alternative.

5.4.3.5 Access of External Information

The structure of the object code has been designed such that access of external information, i.e. outside the current unit of compilation, usually incurs very little overhead to that of internal information. The method of external access is described below together with access of formal procedure parameter information to which it is analagous.

(1) Data Access

All data lies within the lower 8K and is always directly addressed.

Common data access is therefore identical to internal data access.

Formal data access is similarly identical except for formal by 'LOCATION' parameter access (5.4.1.1).

(2) Procedure Access

If a Common procedure is loaded into the same core module as its call the normal procedure call instruction pair is generated:

```
11    LINK
 8    ENTRY
```

Therefore by careful program loading no overhead will be incurred.

However if a Common procedure is loaded in a different core module from its call there is still only a two word call generated:

```
11    a
 8    b
```

which cause transfer to the procedure via a further pair of compiler generated instructions for that procedure and a compiler generated linkage procedure. The overhead is therefore

5.4.3.5
(cont.)

two instructions per procedure declaration (not call) of such a procedure plus a single general purpose linkage procedure (approx 15 words) which is used for all inter-module procedure calls from the current module.

A formal procedure call always causes the generation of three instructions:

```
 0     c
11     d
 8     e
```

which causes transfer to the procedure via the above compiler generated linkage procedure (since the position in core of any corresponding actual procedure is unknown).

(3) Label Access

A pair of instructions are generated at the head of a unit of compilation for a Common label and they transfer control to the label via a three word general purpose module relativising sequence which only occurs once per module:

```
L':   4    absolute address of label L
      8    module relativising sequence
```

Therefore any reference to the label within the unit of compilation only generates a single jump instruction which transfers control to the label via this two word pair:

```
8     L'
```

A similar pair of instructions are generated at the head of a procedure for a formal label such that all references to the label are generated as single jump instructions.

5.4.3.5
(cont.) (4) Switch Access

In order that all switch element accesses are only three instructions in length, internal, Common and Formal switch element accesses are identical:

```
4     Index
1     + SS[0]
8     module relativising sequence
                          (as in (3) )
```

(This has a slight time overhead if the switch is internal).

5.4.4 MULTI-LEVEL HOUSEKEEPING CODE

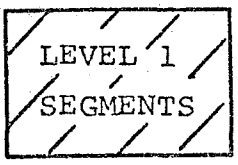
A knowledge of 1.2.3.1 is assumed. The following interrupt handling housekeeping code is generated by the Loader as the segments of a multi-level program are loaded.

The following code is generated by the Loader in the fixed code area of module 0. The PROGRAM ENTRY POINT supplied by the Loader (4.1.5.2) is at START.

```
RESTART   4   -1
          5   TOP LEVEL INDICATOR (-1→ AUTOSTART)

START     4   =8 RESTART          (Set up AUTOSTART)
          5    8177

                                  (Set up low level
          4   +L2S                SCRs)
          5    2
          4   +L3S
          5    4
          4   +L4S
          5    6

         15   7177                (14 0 if program
                                   loaded in module 0)

          0   +L1S
         /8   0
```

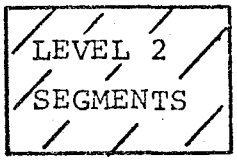
The following code is generated enveloping the segments of each level and may therefore reside in any module.

SEGLEV1



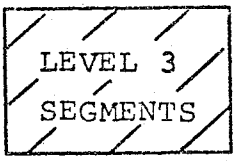

```
          2   L1Q       (reset  A & Q)
          4   L1A
         15   7168
L1S      15   7177        (14 0 if program loaded
          5   L1A          in module 0)
         14   18
          5   L1Q
          4   +1
          5   TOP LEVEL INDICATOR (+1  INTERRUPT)
          8   SEGLEV1
```

SEGLEV2



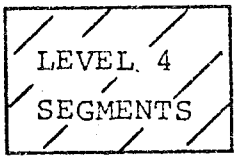

```
          2   L2Q
          4   L2A
         15   7168
L2S      15   7177
          5   L2A
         14   18
          5   L2Q
          4   +1
          5   LOWER LEVEL INDICATOR
          8   SEGLEV2
```

SEGLEV3




```
          2   L3Q
          4   L3A
         15   7168
L3S      15   7177
          5   L3A
         14   18
          5   L3Q
          4   +1
          5   LOWER LEVEL INDICATOR
          8   SEGLEV3
```

SEGLEV4/L4S




```
          8   SEGLEV4
```

NOTE: The 8 SEGLEVn instruction is generated as 8 ; +0 unless the last segment of a level ends with 'GOTO' first segment name;.

APPENDIX A

920C CORAL SYNTAX

As described in the Official Definition of CORAL 66 together with the implementation dependent features and enhancements.

```
Actual = Expression
         Wordreference
         Destination
         Name
Actualist = Actual
            Actual, Actualist
Addoperator = +
              -
Address = Signedinteger
          Addoperator Fraction
          Id
Alternative = Statement
Answerspec = Numbertype
             Void
Answerstatement = ANSWER Expression
Arraydec = Numbertype ARRAY Arraylist Presetlist
           BIT ARRAY Arraylist Presetlist
           BYTE ARRAY Arraylist Presetlist
Arrayitem = Idlist [Sizelist]
Arraylist = Arrayitem
            Arrayitem, Arraylist
Assignmentstatement = Variable ← Expression

Base = Id
       Id [Signedinteger]
Bitposition = Integer
Block = BEGIN Declist ; statementlist END
Booleanword = Booleanword2
              Booleanword4 DIFFER Booleanword5
Booleanword2 = Booleanword3
               Booleanword5 UNION Booleanword6
Booleanword3 = Shiftword
               Booleanword6 MASK Shiftword2
Booleanword4 = Booleanword
               Typedprimary
Booleanword5 = Booleanword2
               Typedprimary
Booleanword6 = Booleanword3
               Typedprimary
Bracketedcomment = (any sequence of characters in which
                    round brackets are matched)

Codeinstruction = Label: Codeinstruction
                  Simplecodeinstruction
Codesequence = Codeinstruction
               Codeinstruction; Codesequence
Codestatement = CODE BEGIN Codesequence END
Common = Commoncommunicator;
         Void
```

```
Commentsentence = COMMENT any sequence of characters
                          not including a semi-colon;
Commoncommunicator = COMMON Id (Commonitemlist)
Commonitem = Datadec
             Overlaydec
             Placespec
             Procedurespec
             Void
Commonitemlist = Commonitem
                 Commonitem ; Commonitemlist
Comparator = < or ≤ or = or ≥ or > or ≠
Comparison = Simpleexpression Comparator Simpleexpression
Compileunit = Program
              Commoncommunicator
              Library
Compoundstatement = BEGIN Statementlist END
Condition = Condition OR Subcondition
            Subcondition
Conditionalexpression = IF Condition
                            THEN Unconditionalexpression
                            ELSE Expression
Conditionalstatement = IF Condition THEN Consequence
                       IF Condition THEN Consequence
                           ELSE Alternative
Consequence = Simplestatement
              Label : Consequence
Constant = Number
           Addoperator Number
Constantlist = Group
               Group, Constantlist
Coralunit = CORAL Compileunit FINISH

Datadec = Numberdec
          Arraydec
          Tabledec
Dec = Datadec
      Overlaydec
      Switchdec
      Proceduredec
Declist = Dec
          Dec ; Declist
Destination = Label
              Switch [Index]
Digit = 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7 or 8 or 9
Digitlist = Digit
            Digit Digitlist
Dimension = Lowerbound : Upperbound
Dummystatement = Void
```

```
Elementdec = Id Numbertype Wordposition
             Id Partwordtype Wordposition, Bitposition
             Id Partwordtype Wordposition BIT
                 Bitposition
Elementdeclist = Elementdec
                 Elementdec ; Elementdeclist
Elementpresetlist = PRESET Constantlist
                    Void
Elementscale = (Totalbits, Fractionbits)
               (Totalbits)
Endcomment = Id
Expression = Unconditionalexpression
             Conditionalexpression


Factor = Primary
         Booleanword
Forelement = Expression
             Expression WHILE Condition
             Expression STEP Expression UNTIL Expression
Forlist = Forelement
          Forelement, Forlist
Forstatement = FOR Wordreference ← Forlist DO Statement
Fraction = .Digitlist
Fractionbits = Signedinteger
Function = Integer

Gotostatement = GOTO Destination
Group = Constant
        (Constantlist)
        Void

Id = Letter Letterdigitstring
Idlist = Id
         Id, Idlist
Index = Expression
Integer = Digitlist
          OCTAL (Octallist)
          LITERAL (printing character)

Label = Id
Labellist = Label
            Label, Labellist
Length = Integer
Letter = a or b or c or .... or z
Letterdigitstring = Letter Letterdigitstring
                    Digit Letterdigitstring
                    Void
```

```
Liblist = Libspec
          Libspec Liblist
          Void
Libproceduredec = Answerspec PROCEDURE Libprocedurehead-
                  ing; statement
Libproceduredeclist = Libproceduredec
                      Libproceduredec; Libproceduredec-
                      list
Libprocedureheading = Id/Digitlist
                      Id/Digitlist (Parameterspeclist)
Libprocedurespec = Answerspec PROCEDURE Libprocparamlist
Libprocparameter = Id/Digitlist
                   Id/Digitlist (Typelist)
Libprocparamlist = Libprocparameter
                   Libprocparameter, Libprocparamlist
Library = LIBRARY Id Liblist Libproceduredeclist
Libspec = LIBRARY Libprocedurespec;
Lowerbound = Signedinteger

Macrobody = any sequence of characters in which string
            quotes are matched
Macrocall = Macroname
            Macroname (Macrostringlist)
Macrodefinition = DEFINE Macroname { Macrobody } ;
                  DEFINE Macroname (Idlist) { Macrobody } ;
Macrodeletion = DELETE Macroname ;
Macroname = Id
Macrostring = any sequence of characters in which commas
              are protected by round or square brackets
              and in which such brackets are properly
              matched and nested
Macrostring = Macrostring
              Macrostring, Macrostringlist
Multoperator = *
               /

Name = Id
Number = Real
         Integer
Numberdec = Numbertype Idlist Presetlist
Numbertype = FLOATING
             FIXED Scale
             INTEGER

Octaldigit = 0 or 1 or 2 or 3 or 4 or 5 or 6 or 7
Octallist = Octaldigit
            Octaldigit Octallist
Overlaydec = OVERLAY Base WITH Datadec

Parameterspec = Specifier Idlist
                Tablespec
                Procedurespec
```

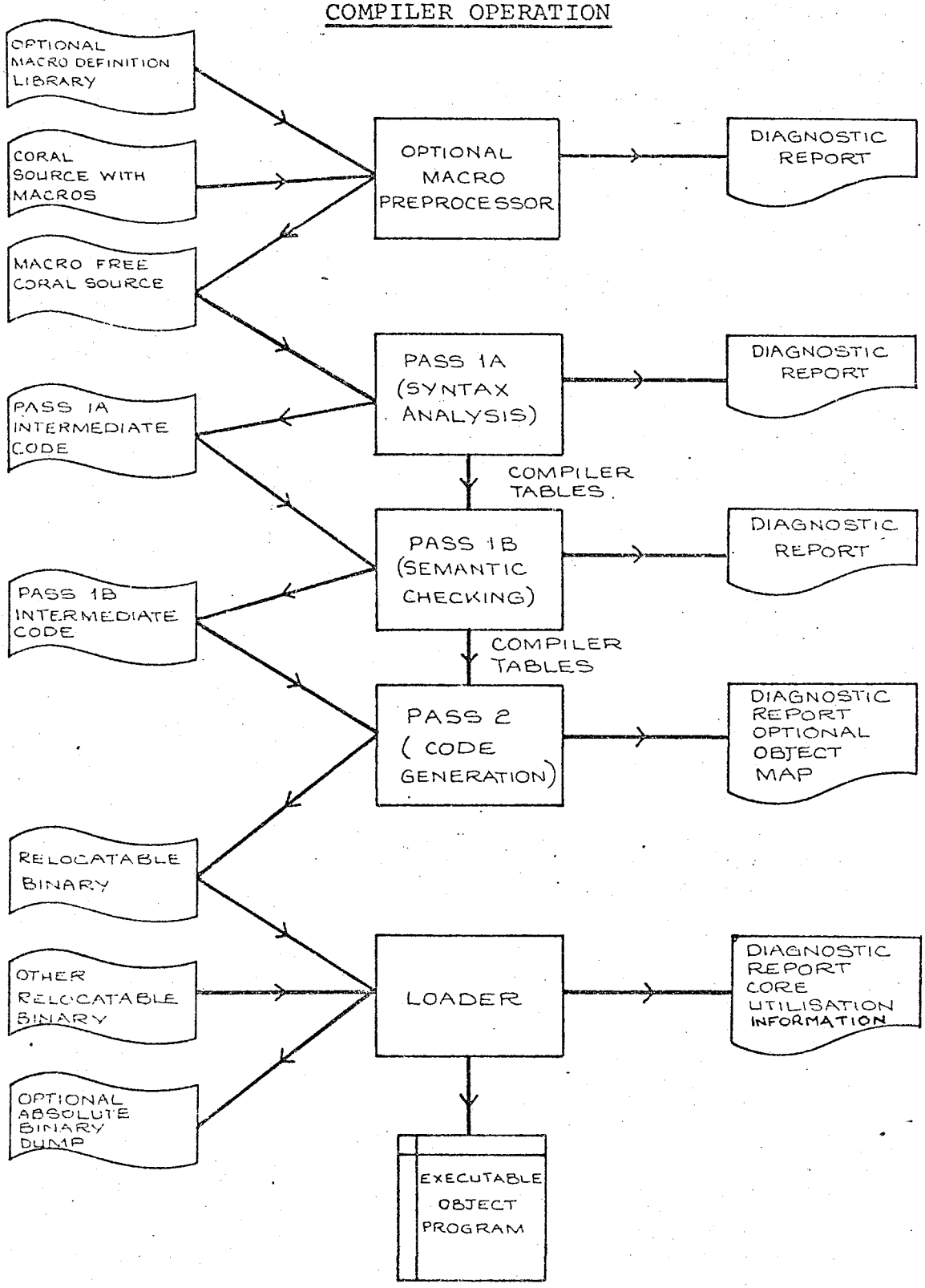
```
Parameterspeclist = Parameterspec
                    Parameterspec ; Parameterspeclist
Partword = Id [Index]
           BITS [Totalbits, Bitposition] Typedprimary
Partwordreference = Id [Index]
                    BITS [Totalbits, Bitposition]
                                  Wordreference
Partwordtype = Elementscale
               UNSIGNED Elementscale
Placespec = LABEL Idlist
            SWITCH Idlist
Presetlist = = Constantlist
             Void
Primary = Untypedprimary
          Typedprimary
Procedurecall = Id
                Id (Actuallist)
Proceduredec = Answerspec PROCEDURE Procedureheading;
               Statement
Procedureheading = Id
                   Id (Parameterspeclist)
Procedurespec = Answerspec PROCEDURE Procparamlist
Procparameter = Id
                Id (Typelist)
Procparamlist = Procparameter
                Procparameter, Procparamlist
Program = PROGRAM Id Liblist Common Segmentlist

Real = Digitlist.Digitlist
       Digitlist 10 Signedinteger
         Signedinteger
       Digitlist. Digitlist 10 Signedinteger
       OCTAL (Octallist. Octallist)

Scale = (Totalbits, Fractionbits)
Segment = SEGMENT Id Block
Segmentlist = Segment
              Segment; Segmentlist
Shiftoperator = LEFT
                RIGHT
Shiftword = Shiftword2 Shiftoperator Typedprimary
Shiftword2 = Shiftword
             Typedprimary
Signedinteger = Integer
                Addoperator Integer
Simplecodeinstruction = / Function, Address
                        Function, Address
                        Void

Simpleexpression = Term
                   Addoperator Term
                   Simpleexpression Addoperator Term
```

```
Simplestatement = Assignmentstatement
                  Gotostatement
                  Procedurecall
                  Answerstatement
                  Codestatement
                  Compoundstatement
                  Block
                  Dummystatement
Sizelist = Dimension
           Dimension, Dimension
Specifier = VALUE Numbertype
            LOCATION Numbertype
            Numbertype ARRAY
            BIT ARRAY
            BYTE ARRAY
            LABEL
            SWITCH
Statement = Label : Statement
            Simplestatement
            Conditionalstatement
            Forstatement
Statementlist = Statement
                Statement ; Statementlist
String = ↑< sequence of characters >↑
Subcondition = Subcondition AND Comparison
               Comparison
Switch = Id
Switchdec = SWITCH Switch ← Labellist

Tabledec = TABLE Id [Width, Length]
                    [Elementdeclist Elementpresetlist]
                     Presetlist
Tablespec = TABLE Id [Width, Length][Elementdeclist]
Term = Factor
       Term Multoperator Factor
Totalbits = Integer
Type = Specifier
       TABLE
       Answerspec PROCEDURE
Typedprimary= Wordreference
              Partword
              LOCATION (Wordreference)
              Numbertype (Expression)
              Procedurecall
              Integer
Typelist = Type
           Type, Typelist
```

```
Unconditionalexpression = Simpleexpression
                          String
Untypedprimary = Real
                 (Expression)
Upperbound = Signedinteger

Variable = Wordreference
           Partwordreference
Width = Integer
Wordposition = Signedinteger
Wordreference = Id
                Id [Index]
                Id [Index, Index]
                [Index]
```

APPENDIX C

COMPILER OPERATION



APPENDIX D

COMPILER INPUT/OUTPUT

COMPILER PROGRAM	INPUT		OUTPUT		
	PERIPHERAL	CORE	PERIPHERAL	CORE	DIAGNOSTICS
MACRO PASS	CORAL source with macros		Macro-free CORAL source		Error and Warning Messages
PASS 1A	Macro-free CORAL cource		Pass 1A intermediate code	Pass 1A compiler tables	Error and Warning Messages
PASS 1B	Pass 1A intermediate code	Pass 1A compiler tables	Pass 1B intermediate code	Pass 1B compiler tables	Error and Warning Messages
PASS 2	Pass 1B intermediate code	Pass 1B compiler tables	Relocatable binary		Error Messages Object Map
LOADER	Relocatable binary		Absolute binary	Program in core	Error and Warning Messages Core utilisation information

APPENDIX E

EXAMPLE PROGRAM

The Program consists of three units of compilation:

Two segments
One segment
Two library procedures

The following information is supplied for the compilation of each unit:

Source
Teletype log containing user commands for each Pass
Object Map

The following information is supplied for the link loading of the units:

Teletype log containing user commands for the Loader
Core utilisation information from the Loader

UNIT 1 : SOURCE

```
'CORAL'
'PROGRAM'EXAMPLE CORAL PROGRAM
'COMMENT'
        THIS PROGRAM HAS BEEN WRITTEN TO DEMONSTRATE THE
        USE OF AND INFORMATION SUPPLIED BY THE 920C CORAL
        COMPILER AND LOADER ;


'LIBRARY' 'PROCEDURE'
          ADDTHEM      / 2 ('VALUE''INTEGER','LOCATION''INTEGER') ,
          SUBTRACTTHEM / 4 ('VALUE''INTEGER','LOCATION''INTEGER') ;


'COMMON' PLACE(
 'PROCEDURE' DIVIDETHEM ('VALUE''INTEGER',
                         'LOCATION''INTEGER') ;
 'LABEL' DEMON 2,DEMON 3 ;
 'INTEGER' INT ; ) ;


'SEGMENT' DEMON 1
'COMMENT'
         SEGMENT ONE CALLS A LIBRARY PROCEDURE ;
'BEGIN'
       'INTEGER' A ;
       A:=6 ;
       ADDTHEM (A,INT) ;
       'GOTO' DEMON 2 ;
'END' ;


'SEGMENT' DEMON 2
'COMMENT'
         SEGMENT TWO DECLARES THE COMMON PROCEDURE
         AND CALLS A LIBRARY PROCEDURE ;
'BEGIN'
       'INTEGER' A ;
       'PROCEDURE' DIVIDETHEM ('VALUE''INTEGER' B ;
                               'LOCATION''INTEGER' C) ;
                   C:=B/6 ;

       A:=7 ;
       SUBTRACTTHEM (A,INT) ;
       'GOTO' DEMON 3 ;


'END' ;
'FINISH'
```

UNIT 2 : SOURCE

```
'CORAL'
'PROGRAM' EXAMPLE CORAL PROGRAM

'COMMENT'
         THIS PROGRAM HAS BEEN WRITTEN TO DEMONSTRATE THE
         USE OF AND THE INFORMATION SUPPLIED BY THE 920C CORAL
         COMPILER AND LOADER ;


'COMMON' PLACE (
 'PROCEDURE' DIVIDETHEM ('VALUE''INTEGER',
                         'LOCATION''INTEGER') ;
 'LABEL' DEMON 2, DEMON 3 ;
 'INTEGER' INT ; ) ;


'SEGMENT' DEMON 3
'COMMENT'
         SEGMENT THREE USES THE COMMON PROCEDURE ;
'BEGIN'
       'INTEGER' A ;
       A:=6 ;
       DIVIDETHEM (A,INT) ;
'END' ;
'FINISH'
```

UNIT 3 : SOURCE

```
'CORAL'
'LIBRARY' EXAMPLE CORAL PROGRAM

'COMMENT'
        THIS LIBRARY HAS BEEN WRITTEN TO DEMONSTRATE THE
        USE OF AND THE INFORMATION SUPPLIED BY THE 920C CORAL
        COMPILER AND LOADER ;


'PROCEDURE' ADDTHEM          / 2 ('VALUE''INTEGER' W ;
                                  'LOCATION''INTEGER' X) ;
            X:=W+3 ;


'PROCEDURE' SUBTRACTTHEM     / 4 ('VALUE''INTEGER' Y ;
                                  'LOCATION''INTEGER' Z) ;
            Z:=Y-3 ;
'FINISH'
```

UNIT 1 - COMPILATION LOG

```
*GO             (Pass 1A)
*GO             (Pass 1B)
*GO             (Pass 2 )
```

UNIT 1 - OBJECT MAP

No object map requested

UNIT 2 - COMPILATION LOG

```
*OUT=TTY                        (Incorrect command - should have
*GO                              typed LST)
DEVICE SPECIFICATION ERROR      (Output cannot go to TTY)

*OUT=PTP                        (Reset output to PTP)
*LST=TTY                        (Object map request)
*CKS                            (Source checksum request)
*GO                             (Pass 1A)
RELOAD TAPE

*GO                             (Pass 1A - repeat input of
                                 tape for checking)
*GO                             (Pass 1B)
*GO                             (Pass 2 )
```

UNIT 2 - OBJECT MAP

```
PROGRAM EXAMPLECORAL

COMMON PLACE

D      O OOOOOO DIVIDETHEM
D      8 OOOOlO INT

SEGMENT DEMON 3

L      4 OOOOO4 DEMON 3
D      O OOOOOO A
```

UNIT 3 (LIBRARY) - COMPILATION LOG

```
*LST=TTY                    (Object map request)
*GO                         (Pass 1A)

*GO                         (Pass 1B)

*GO                         (Pass 2 )
```

UNIT 3 (LIBRARY) - OBJECT MAP

```
LIBRARY EXAMPLECORAL

LIBPROC ADDTHEM          / 2

LIBPROC SUBTRACTTHEM     / 4
```

LOADING INFORMATION

```
*DMP                              (Absolute binary dump request)
*RAD=8                            (Radix=octal)
*MOU=1600                         (Object program to lie in module 0
                                   below location 1601)
*M1U=+20000
*COMMAND ERROR
*M1U=20000
*GO                               (Load Unit 1)

CORE BOUNDS      1052  1600  20000  20000

PROGRAM EXAMPLECORAL
1547  1274  DIVIDETHEM

COMMON    1271
DATA      1302
CODE      1531
SWITCH    1571

SEGMENT   DEMON 2
SEGMENT   DEMON 1
COMMON    PLACE
UNDECLARED LIBRARY PROCEDURES

*GO                               (Load Unit 2)
PROGRAM EXAMPLECORAL

COMMON    1271
DATA      1321
CODE      1512
SWITCH    1526

SEGMENT   DEMON 3
COMMON    PLACE

UNDECLARED LIBRARY PROCEDURES

*GO                               (Load Unit 3)
LIBRARY EXAMPLECORAL
LIBPROC SUBTRACTTHEM
1503  1316  SUBTRACTTHEM

COMMON    1271
DATA      1323
CODE      1502
SWITCH    1512
```

LOADING INFORMATION

(Cont'd)

```
LIBPROC ADDTHEM
1473  1310  ADDTHEM

COMMON    1271
DATA      1323
CODE      1472
SWITCH    1502

*END
*GO.

PROGRAM ENTRY  1571
```

APPENDIX G

SUMMARY OF OPERATING INSTRUCTIONS

MACRO PASS	PASS 1A	PASS 1B	PASS 2	LOADER
. Load Macro Pass binary tape. . Type options. . Source tape in reader. . Type GO↵ . . Repeat 3&4 for each continuation tape.	1. Load Pass 1A binary tape. 2. Type options. 3. Source tape in reader. 4. Type GO↵ . 5. Repeat 3&4 for each continuation tape.	1. Load Pass 1B binary tape. 2. Pass 1A tape in reader. 3. Type GO↵ .	1. Load Pass 2 binary tape. 2. Pass 1B tape in reader. 3. Type GO↵ . WIND THE RESULTING TAPE UP BACKWARDS	1. Load Loader binary tape. 2. Trigger to 4096_{10}. 3. Type options. 4. RLB tape in reader. 5. Type GO↵ . 6. Repeat 4&5 for each program tape. 7. Repeat 4&5 for each Library tape. 8. Type END↵ . 9. Type GO↵ .

CORAL 8K EXTENDED LOADER, Binary Mode 3;

CORAL 16K EXTENDED LOADER, Binary Mode 3.

CONTENTS

INTRODUCTION

This document is for use in conjunction with the CAP Royal Aircraft Establishment 920C CORAL COMPILER Users Manual to describe the use of the Extended Loader, and it is assumed that the user is familar with that document. All section numbering is followed by a reference to the equivalent section in that Manual. Any other references to the User Manual are prefixed U.M.

The minimum configuration for producing a loadable CORAL 66 program compiled by the CORAL compiler is an Elliott 920B (903) with 8K of core store, a paper tape reader, a paper tape punch and a teletype. The program may then be run on an object machine with up to 128K of core store, as specified at load time.

The following description summarises the contents of each chapter:

Chapter 1 : A description of the ways in which the Extended Loader affects the usage of the CORAL language.

Chapter 2 : A description of the purpose of the Loader, with a description of the use of the various options and commands available. An explanation of the order of loading units of compilation. A description of the general method of interface with the user.

Chapter 3 : A detailed description of the operating instructions for using the Extended Loader. A description of all options available to the user. Operating instructions for use with the Object Program.

Chapter 4 : A list, with reasons, of all possible Loader error messages. A description of the core map produced by the Loader.

Chapter 5 : A description of the way in which the Extended Loader locates and produces the object program in Absolute Binary form.

1. CORAL LANGUAGE

The implementation of the CORAL syntax is as described in the 920C CORAL compiler User's Manual. The following sections serve to amplify and supplement the features relevant to the Extended Loader.

1.1(1.1.1) UNITS OF COMPILATION

As indicated, there are four types of units of compilation which may be compiled and loaded by the Extended Loader.

These are: a single program segment
a set of program segments
a Common segment
a set of Library procedures

It should be emphasised that a Common sgement unit of compilation may only be loaded as a separate unit and not at the same time as other types of units of compilation which will by definition contain the common segment.

1.2 (1.1.1.4) OBJECT CODE LIMITS ON COMPILATION UNIT SIZES

Due to the object code strategy of absolute addressing of data, the following limits exist:

(1) The data area generated by 1 CORAL compilation unit must be ≤ 8K.

(2) The executable code generated by 1 CORAL compilation unit must be ≤ 8K.

(3) The data area generated by 1 CORAL program whether compiled as a whole or in separate units must be ≤ 8K, since it must lie within module 0. However core locations outside module 0 may be accessed as data via indexed variables or anonymous references with large indices.

(4) The executable code generated by 1 CORAL program must be ≤ 128K and therefore if > 8K must be compiled in sections to adhere to (2).

(5) The data area and executable code generated by 1 CORAL program must be ≤ 128K although any core locations above 128K may be accessed via indexed variables with large indices.

A full description of the runtime storage and object code strategy is provided in Chapter 5.

1.3(1.2.3) RUNTIME FACILITIES

1.3.1(1.2.3.1) Multi-level programs

The interrupt handling housekeeping code is not generated by the Extended Loader, the responsibility for writing and including this is with the CORAL programmer.

The operating instructions for loading and executing a multi-level program are provided in section 3.1.

A CORAL program may be split into segments which run on different levels, a minimum of one segment per level, and all levels must be present. As for a normal program the same Common communicator must accompany each unit of compilation and is therefore shared between levels. The level upon which segments reside is determined by the input of the relevant Loader option before loading those segments.

CORAL code, i.e. Common procedures, should not be shared between levels since 900 code is not reentrant, and it is the responsibility of the user to ensure that this does not happen - no checks are performed by the Loader. Similarly, care must be taken in updating Common data which is shared between levels.

The user need not maintain four copies of Library procedures, one for each level. The same Library tape may be loaded at each level; the Loader treats the Library procedures as different on each level.

1.3.2(1.2.3.2) Program sumcheck

The Extended Loader calculates the runtime sumcheck of the executable code of a CORAL program and prints it out in octal at the end of loading.

The sumcheck is the sum of the contents of all locations included in the program's Unit Code Bounds including the entry sequence (ignoring overflow). None of the data areas are included since data, unlike code, may be variable.

1.3.3(1.2.3.3) Data Area Initialisation

Note that the Loader does not include any clear-store facility. It is up to the user to ensure that any areas requiring initial values are either preset or are explicitly assigned to at the start of the user's program. Remember that presets are only set when the program is first loaded: to ensure that a program can be restarted without reloading it, locations used for variables should be given any initial values needed by assignments.

2(2.1.5) LOADER DESCRIPTION

2.1(2.1.5) DESCRIPTION

The Extended Loader links together independently compiled CORAL units of a program into an executable program, producing the latter in the form of a sumchecked binary tape (or tapes).

It is purpose built and therefore does not allow CORAL units of compilation to be linked with any other program unit produced via another compiler or assembler.

The Loader accepts relocatable binary from Pass 2 of the CORAL compiler.

Note that the relocatable binary tapes produced by the compiler must be input to the Loader backwards, i.e. the character produced last by the Compiler must be input first to the Loader.

As well as producing the absolute binary of the program, a utilisation of core map is produced on the teletype (unless suppressed by the user.), see 4.2.

During loading, detection of an error which is not considered disastrous does not inhibit the loading process and execution of the object program is at the user's discretion. However, an irrecoverable error will cause the loader to halt. The Loader may be re-entered to recommence the loading process, without reloading into core.

It should be noted that an incomplete program, i.e. a subset of the units of compialtion comprising the whole program, may be loaded and executed similarly at the user's discretion.

2.2(2.1.5.2) OPTIONS

The following options are provided by the Loader using the standard Extended Loader user interface, see 2.5.

2.2.1 Initialisation Options

The options included in this section are all concerned with initialisation. They cannot be used once a Load-Time option has been input.

2.2.1.1 Self-triggering program (AUT)

The user can specify that he requires his object program to be self-triggering, on being read into core.

The default is that this is not required.

2.2.1.2 Object Machine Core Size (COR)

The minimum number of modules in the object machine can be set using this option. Note that even if the actual core is not present in the object machine the value input should be the module number of the highest module to be used, plus one.

This option also prints out the object machine configuration.

The default is that the object machine consists of not less than 4 modules i.e. modules 0-3.

2.2.1.3 Core Utilisation Map Suppression (MAP)

The Extended Loader normally provides information on the object program's utilisation of core, as units of compilation are loaded. However, there is an option to suppress this information. All error and warning messages are still printed on the teletype along with the entry point, level and checksum information. Input is still via the teletype.

2.2.1.4 Module Bounds (MUB, MLB)

The upper and lower bounds of any module may be changed to fit any object machine constraints.

The defaults for each module are the normal extremities of a module, namely 8191 and 0 respectively, except for module 0 where the lower bound is 8, and the upper 8166.

2.2.1.5 Radix of Input/Output (RAD)

The radix of the numbers input by the user on the teletype during the specification of options, and those output on the teletype by the Loader i.e. core map, entry point may be either octal or decimal. The exceptions are error numbers which are always decimal, and the code checksum which is always octal.

The default is decimal.

2.2.1.6 Tape size (TAP)

The object program can be output on more than one tape if the user does not want a tape to be unwieldy. He can do this by specifying the maximum number of words to be output on any one tape. Each tape is preceded by a tape number in legible tape code, which must not be read. The object program tapes are checked to ensure loading of the tapes is in sequence and that each tape is read correctly.

The default is a maximum size of 16384 (decimal) words.

2.2.2 Load-Time Instructions

After all the required initialisation options have been input, the following may be input before loading an RLB tape.

2.2.2.1 Unit of Compilation Code address (ADD)

The user may select the object machine base address of the code in a unit of compilation. He otherwise may choose the module in which it is to be located. Previously loaded code can not be overwritten.

The default is to place the unit's code in the locations chosen by the Optimum Location of Units of Compilation Algorithm, see 5.3.2.

2.2.2.2 End of Loading (END)

Termination of the loading sequence, incorporating the resolving of procedure references, must be terminated by the user.

The default is to continue accepting more units of compilation.

2.2.2.3 Entry address (ENT)

The object program's entry point is the first statement in the first (last) segment compiled (loaded) in a specified unit of compilation.

The default entry point is the first statement in the first segment compiled in the first unit of compilation loaded.

2.2.2.4 Process Relocatable Binary Tape (GO)

On receipt of this command the Extended Loader reads and processes the next unit.

2.2.2.5 Program level (LEV)

The precise operating instructions for loading multi-level programs are given in 3.1. The loader runs in either single-level mode or multi-level mode.

The default for the level of a unit of compilation when loaded normally is level 1 and therefore a single level program will always run on level 1.

2.2.1.6 Relocatable binary version (RLB)

The Extended Loader can accept relocatable binary tapes from current or obsolete versions of the Compiler.
The version can be specified by the user.

The default is automatic recognition by the Loader.

2.2.2.6 Undeclared procedures list (UND)

If, after processing a unit of compilation, there are still references to undeclared procedures a message warns that there are undeclared Library and/or Common procedures. The user may request a list of these.

2.3(2.1.5.3) ORDER OF LOADING

2.3.1(2.1.5.3.1) Single level program

The relocatable binary paper tapes for the units of compilation may be loaded in any order excluding the library tape(s) which must be loaded last.

The entry point of the program is assumed to be the first statement of the first segment compiled unless changed by the user, see 2.2.2.3.

e.g.
segment tape 1
segment tape 2
⋮
segment tape N
Library tape 1
Library tape 2
⋮
Library tape M

2.3.2(2.1.5.3.2) Multi-level program

The relocatable binary paper tapes for the units of compilation of a multi-level program must be loaded together for each level but may be in any order within the level followed by the library tape(s) for that level. (The levels may also be loaded in any order). The same library tape(s) may be read for each level and the Loader will create a copy of each relevant procedure per level upon which it is used. The entry point of a level is assumed to be the first statement of the first unit loaded on that level, except in the case when the ENTry option is used. In this case, the entry point for that level will be the entry point of the program.

e.g.
```
segment tape 1, level 1
        :
segment tape N1, level 1
Library tape 1
Library tape M

segment tape 1,  level 2
        :
segment tape N2, level 2
library tape 1
        :
library tape M

segment tape 1,  level 3
        :
segment tape N3, level 3
library tape 1
        :
library tape M

segment tape 1,  level 4
        :
segment tape N4, level 4
Library tape 1
        :
library tape M
```

The concept of hardware levels can also be carried across to software levels. More than one set of units of compilation and library tape(s) can be loaded on one hardware level, so that more than one copy of a library procedure may be generated on that hardware level. Before each independent set is loaded the LEVel option must be used to specify that the following set is to be loaded on the same level. As specified above the level entry point is either the default (the first statement of the first segment compiled in this set) or the first statement of the first segment in the unit of compilation specified by the ENTry option; but note that in both cases the hardware level entry point output in the Fixed Data area, and therefore that hardware level entry point when the program is run, is the entry point of the last set (of units + library) loaded on the hardware level.

As an example, if more than one set of units + library are to be loaded on hardware level 4, the following would be added to the above loading sequence

```
segment tape 1, level 4(2)
        :
        :
segment tape S2, level 4(2)
Library tape 1
        :
Library tape M
      :
      :
segment tape 1, level 4(x)
        :
        :
segment tape Sx, level 4(x)
Library tape 1
        :
Library tape M
```

2.4 (2.1.5.4) LIBRARY PROCEDURE LOADING

As described in 2.3, the Library tape(s) are loaded after all units of compilation for the current level. The Extended Loader performs a scan of each Library tape and loads only those procedures which have been referenced previously, i.e. having the same Library procedure number. Any number of Library tapes may be scanned until all referenced are satisfied. The Loader outputs a description of the Library procedures loaded, see 4.2.

The following points should be noted:

1) The Loader performs no check on duplicate Library numbers and simply loads the first procedure encountered with the required number (i.e. last compiled if both are within the same unit) - all subsequent procedures with the same number being ignored. This therefore allows the user to redefine Library procedures on that level.

2) Since communication with a Library procedure is via the Library procedure number and not the name, reference to different procedure names which have the same number will cause calls to the same procedure at run time.

3) The Library tape supplied with the 920C CORAL Compiling System, CAPQF, contains the Compiler Floating Point Library Procedure which has the Library Procedure Number 1. The user should therefore avoid the use of this number since redefinition of this procedure would no doubt have disasterous consequences.

4) Library procedures which have not previously been references are simply ignored - they are not loaded. The user should therefore ensure that any Library procedure called only by another Library procedures is loaded after (i.e. compiled before) the Library procedure(s) calling it. This will save multiple reading of any one library tape.

2.5(2.4) INTERFACE WITH THE USER

2.5.1(2.4.1) Command Language

Communication between the operator and the Extended Loader for selection of loading options is by means of commands typed in at the teletype in response to an invitation to type from the Loader.

Commands are split into two types of command: Intialisation and Load-time, see 2.2. Once one load-time command has been typed in, no more initialisation commands are accepted.

If an Initialisation command is typed with incorrect parameters it can be reinput with the correct parameters.

2.5.2(2.4.2) Command Format

The invitation to type a command issued by the Loader is an * at the start of a new line.

A command string is of the format

COMMAND cr

or COMMAND = parameterstring cr

where parameterstring is defined as

PARAM1
or PARAM1, PARAM2

Only the first three characters of the COMMAND are used (except for the GO command, where only two are used), the COMMAND being terminated by either = if parameters are to be used, or cr otherwise.

The parameter(s) PARAM1 and PARAM2 are either alphabetic or numeric. In the case of alphabetic parameters the last three (or two in the NO case) characters of each parameter are the ones accepted (or rejected if invalid). In the numeric parameter case there must be between 1 and 6 numeric characters in any valid parameter.

A parameter is terminated by a (,) if there is a second parameter to follow or cr otherwise.

A description of all the commands and their associated parameters is provided in chapter 3.

To remove a character from the input buffer if it has been incorrectly typed, a← may be typed; n← characters will remove n characters, if typed before the cr.

A← may be output by the Loader if the character just received was of invalid parity. The character will be removed from the input buffer.

A linefeed typed before the cr deletes the whole of the command being input, and an invitation to type a fresh command string is given.

All spaces are ignored.

2.6(2.5.1) COMMON CHECKING

The following checks are performed by the Loader on the Common communicator and its associated segment(s) for the units of compilation of a program:

1) The size of the runtime Common area is the same for all units of compilation.

2) A Common label is only declared once.

3) A Common switch is only declared once.

4) A Common procedure is only declared once and all are declared.

There are no other checks performed on Common and it is the responsibility of the user to ensure that the same Common communicator is used with each unit of compilation of a CORAL program and that Common procedures are not shared between interrupt levels. It must be noted that the Loader only loads the first Common area it encounters - all other Commons are simply checked as described above.

3 OPERATING INSTRUCTIONS

3.1(3.1.5.1) DETAILED OPERATING INSTRUCTIONS

1) Load the Extended Loader binary paper tape using the hardware initial instructions.

   If the Extended Loader is to be run in an 8K machine OR the machine is a 903, 920B or 920M, use the tape: "CORAL 8K EXTENDED LOADER, Binary Mode 3".

   If the Extended Loader is to be run in a ≥16K machine AND the machine is a 905 or 920C, use the tape: "CORAL 16K EXTENDED LOADER, Binary Mode 3".

   The only significant difference between these two versions of the Extended Loader is the amount of dictionary space available inside them:-

   3066 words in the 8K version,
   6491 words in the 16K version.

   Apart from this, the choice of version has no direct relationship with the size of the run-time machine: either loader can theoretically handle 128K programs, and programs for any of the above 900-Series machines.

2) Trigger to 8 using the hand-keys.

3) Type the required Initialisation commands on the teletype; see 3.2.1.
An * will be printed on the teletype as an invitation to type each of these.

4) Place a relocatable binary paper tape of the program in the paper tape reader The end of the RLB tape last output by Pass 2 of the compiler should be read first by the Loader.

5) Type any required Load-time options on the teletype, see section 3.2.2.

ENTry — if this is the unit whose first executable instruction is to be the program (last time) or level entry point.

LEVel — before the first unit is loaded on this level. When the second and subsequent LEVel commands are used, a new absolute binary tape is begun. The multi-level and single-level options are mutually exclusive.

ADDress — If the unit to be laoded is to be located by the user; see 5.3.2 for details of the use of this option.

6) Type the GO command, whereupon the RLB tape will be read in and processed. At the same time the absolute binary tape containing the object program will be produced and the core map details printed on the teletype (unless suppressed at step (3) by the MAP option).

7) On completion of the unit's processing the message(s)
UNDECLARED COMMON PROCEDURES
UNDECLARED LIBRARY PROCEDURES
will be output on the teletype where appropriate followed by an invitation to type. The operator may use the UNDeclared option to obtain a list of these undeclared procedures.

8) Repeat steps (4) - (7) for each relocatable binary tape of the program to be loaded. Do not forget to keep to the correct loading order, especially in the case of multi-level programs; see 2.3 and 2.4.

9) When all program tapes have been loaded and all library procedure calls are satisfied on the final level type the END command.

The absolute binary tape(s) for the loadable object program will be completed and the entry point, entry level and code checksum will be output to the teletype.

10) The sequence 2-9 may be repeated for another program without reloading the Extended Loader.

3.2(3.1.5.2) COMMANDS

For command language format see section 2.5.2.

3.2.1 Initialisation Commands

COMMAND	PARAMETERS	DEFAULT	MEANING
AUT	YES NO	NO	Object program tape to be self-triggering.
COR	[m] 1≤m≤16	m=4	No. of core modules in object machine (i.e. maximum module number + 1). This remains unchanged if the parameter is omitted. The new object machine configuration is printed; see 4.2.
MAP	YES NO	YES	Object program core map {is / is not} required
MLB	m,n 0≤m≤max module 0≤n≤8191 n=8192	n=0 except for m=0 where n=8	The object machine module m has an inclusive lower bound of n. Module m has no core available.
MUB	m,n 0≤m≤max module 0≤n≤8191 n=8192	n=8191 except for m=0 where n=8166	The object machine module m has an inclusive upper bound of n. Module m has no core available.
RAD	OCT DEC	DEC	User interface number radix - Octal - Decimal
TAPe	W	$W=16384_{10}$	Maximum number of object code words to be output on any one absolute binary tape.

3.2.2 Load-time commands

COMMAND	PARAMETERS	DEFAULT	MEANING
ADDress	m ,n 0≤m≤max module 0≤n≤8191	Unit of compilation's location decided by Loader (see 5.3.2)	The start of the unit of compilation's object code not including loader generated code, see section 5.3.2. m=module number n=module relative address
END	YES NO	NO	All RLB tapes {have / have not} been loaded
ENTRY	none	Entry point is first executable word in first unit loaded.	Entry point is first executable word in the unit loaded after the next GO command.
GO	none	none	Start processing RLB tape in reader.
LEVel	l l=0 l=1,2,3 or 4	l=0	Level at which following units are to be loaded. For a single-level program the LEVel command may be used a maximum of once (l=0) For a multi-level program a LEVel command with the appropriate parameter must be typed before loading the units for that level.
RLB	3	automatic recognition	RLB format version, use "3" for current issue of Compiler
UND	none	none	Print out a list of undefined procedure names which have been referenced.

3.3(3.1.6) OBJECT PROGRAM

After loading using the 920C Extended Loader, the object program is on the absolute binary tape(s) produced. It is not in core.

To execute the absolute binary object program:

1) Load the first absolute binary tape in the object machines reader, ensuring the legible tape number will not be read, and load using the hardware initial instructions.

2) Repeat step 1 for each absolute binary tape produced, loading in strict numerical order, until all have been loaded.

3) If the AUT=YES command was used, execution of the object program occurs when the last absolute binary tape has been loaded.
Otherwise, trigger to the entry point provided, using the hand-keys.

3.4(3.3.1) PAPER TAPE OUTPUT SEPARATION

If the paper tape punch runs out during the loading process the user has no choice but to reload the paper tape punch, retrigger the Loader, and start loading the program again, from the beginning. He may not runout blanks during the loading sequence. It would be advisable therefore to ensure that there is enough tape in the punch before starting the loading of a program.

In the case of a multi-level program, the Loader starts a new tape for each level loaded. There will be at least 4 tapes, therefore, for a multi-level program.

When a tape has been split by the Loader into more than one absolute binary tape, the user may subsequently physically divide the tapes between the end of the "tail" of one section and before the beginning of the "head" of the next. It is suggested that the best place to split the tape is just before the legible tape number, in order to execute the loading into the object machine in the correct sequence.

It should be emphasised that tapes must be loaded in STRICT NUMERICAL SEQUENCE, since each new tape (excluding the first) checks that the checksum of the previous tape is the same as the checksum stored on the front of the current tape.

4 DIAGNOSTIC OUTPUT

4.1(4.1.5) LOADER DIAGNOSTIC OUTPUT

The format of error messages printed on the teletype by the Loader is:

>> nnn [message]

where nnn is a three digit error number

The error number is followed by a message except in the case of Loader software errors (see 4.1.3).

4.1.1 Fatal error messages

Number	Message	Meaning	Result
100	ACM/EPT OVERFLOW	Too much use is being made of the ADDress option, resulting in the Loader's available Core Map having too many entries due to the fragmentation of core. OR: The number of External Procedures called on this level is too high, see P-SYD-1166 section 5.	Loading Halts
101	INVALID TAPE	Tape read is not a valid RLB tape, or has been misread.	Loading Halts
102	PARITY FAILURE	Character on RLB tape is invalid OR User is attempting to load Version 2/3 RLB when Version 1 type is expected.	Loading Halts
103	CHECKSUM FAILURE	Checksum failure on input of RLB tape	Loading Halts
104	PRL/ICL OVERFLOW	Too many external procedure references on this level; OR excessive use of the ADDress option, therefore not using the Optimum Location of Units of Compilation Algorithm, see 5.3.2, resulting in too many inter-module communication code blocks, see 5.3 and 5.4.2.	Loading Halts
105	CORE FULL-CODE	This is an object program code overflow. An attempt is being made to reserve code for this unit of compilation: no free area large enough can be found A different loading order or different use of the ADDress option may be successful, if use is made of the object code core	Loading Halts

Number	Message	Meaning	Result
106	CORE FULL -DATA	This is an object program data overflow. The compiler-generated data area, see 5.4.1, will not fit into the available space in module zero. As this is the first data area to be allocated the MLB and MUB options for module 0 should be altered to increase the available core.	Loading Halts
107	CORE FULL DATA	This is an object program data overflow. The common area is too large to fit into the available space in module 0. Either the common size must be decreased or the MLB, MUB options altered to increase the module zero available core.	Loading Halts
108	CORE FULL DATA	This is an object program data overflow. The data area, including local data, constants address constants and library parameter blocks will not fit into any of the available free area(s) in module 0. More space in module zero must be made available for data, or the data requirement must be reduced on subsequent loads.	Loading Halts
109	CORE FULL DATA	This is an object program code overflow. No room is left to take the 10 word area for the entry code sequence, see 5.4.4. As this is the last area to be allocated, the area available in module zero should be increased or the data requirements decreased or a unit of compilation placed in another module.	Loading Halts

Number	Message	Meaning	Result
110	COMMON CHECK FAILURE	The size of common in this unit of compilation is not the same as that of previously loaded unit(s) of compilation OR a Common segment unit of compilation has been loaded.	Loading Halts
111	LAS/CLS OVERFLOW	The loader label stack has overflowed. This may be overcome by loading the units of compilation with the largest number of forward references first and those units containing the most common label declarations last. see P-SYD-1166 section 5.	Loading Halts
112-113	INVALID TAPE	Tape read is not a valid RLB tape, or has been misread.	Loading Halts

4.1.2 Non-Fatal Error Messages

Number	Message	Meaning	Result
400	INVALID TAPE BLOCK	Tape being read is not of a valid RLB format. Possibly the tape is being loaded "back to front".	The tape is rejected. An asterisk is output awaiting instructions from the teletype.
401	COMMAND. UNKNOWN	First three characters of the command input do not correspond to any vaid user command.	The command is ignored and an asterisk prompt is output awaiting a valid command.

Number	Message	Meaning	Results
402	COMMAND ILLEGAL	This command cannot be input at this stage in the loading sequence.	The command is ignored and an asterisk prompt is output awaiting a valid load-time command. See 3.2 for details.
403	PARAMETER INVALID	The parameter(s) in the command string are not valid or maybe inconsistent.	The command is ignored and an asterisk prompt is output awaiting a valid command.
405	LAST LEVEL NOT LOADED	No RLB tape has been sucessfully loaded at the level last specified OR in the multi-level case, not all LEVels have been read in.	This command is ignored and an asterisk prompt is output awaiting a valid command.
406	COMMAND ERROR	One of the mandatory parameters is missing.	This command is ignored and an asterisk prompt is output awaiting a valid command.
407	LIBRARY LOADED ON THIS LEVEL	An attempt is being made to load another program tape after a library tape has been loaded at this level OR the LEVel option has not been input to change level after the LIBRARY tape(s) have been input on the previous level.	This tape is rejected and the loader awaits instruction to change level, or continue loading at this level.
408	INVALID TAPE	An attempt is being made to load an invalid RLB tape OR an RLB tape has been incorrectly loaded.	The tape is rejected. An asterisk prompt is output awaiting instructions from the teletype concerning the next tape to be loaded.
409	LIBRARY NOT EXPECTED	A library tape has been loaded at this level before a program tape has been accepted.	The library tape is rejected. An asterisk prompt is output awaiting instructions from the teletype for loading a program tape at this level.

Number	Message	Meaning	Result
413	ADDRESS NOT AVAILABLE	The ADDress option has been used specifying a previously allocated area OR the address specified overlaps module boundaries.	An asterisk prompt is output awaiting further ADDress and GO or just a GO option input.
414	NAME DECLARED TWICE procedure name	The Common procedure designated has been declared in a previous unit of compilation.	All references to the Common procedure apply to the first declaration; the second is ignored, though still loaded. Loading continues.
415 - 420	INVALID TAPE	see 408	see 408
421	COMMON LABEL DECLARED TWICE	A common label declared in the current unit has been declared in a previously loaded unit. This includes the case where this tape has been loaded previously.	This label declared in the first unit is used. Loading continues.
422	INVALID TAPE	see error 408	see error 408

4.1.3 Loader Software Error Messages

In this case only the error number is output. Error numbers are 700-999 and are all fatal errors, causing loading to halt. They arise due to Loader inconsistencies and should be referred to Maintenance.

4.2(4.1.5.2) CORE UTILISATION INFORMATION

Object Program Core Map

The following information is output by the Loader, if the core map is not suppressed by the MAP option, when the CORe command is used. Note that in all cases core area bounds are inclusive

CORE AVAILABLE

MODULE	BOUNDS
0	<current lower/upper bounds>
1	etc.
etc.	

For each unit of compilation all that is relevant of the following information is output.

{ PROGRAM / LIBRARY / COMMON / SEGMENT / LIBPROC } <name of unit/sub-unit of compilation>

PROCEDURE <name> ENTRY <address>
LINK <address>

FIXED CONSTANTS AREA	<lower bounds>	-	<upper bounds>
COMMON AREA	"		"
DATA AREA	"		"
CODE AREA	"		"
SWITCH AREA	"		"

UNIT CODE BOUNDS <lower bounds> - <upper bounds>

The following messages are output if appropriate

UNDECLARED COMMON PROCEDURES
UNDECLARED LIBRARY PROCEDURES

By invoking a suitable option, the user can then obtain a list of the undeclared procedures, thus:

UNDECLARED PROCEDURES

{COMMON / LIBRARY} <procedure name>

At the end of loading a level the following message is printed

LEVEL <level number> ENTRY AT <level entry point>

At the end of loading the following message is printed

PROGRAM ENTRY SEQUENCE <entry point> - <upper bound>
AT LEVEL <level number>
CODE CHECKSUM <checksum of code in units> (OCTAL)

5. LOADER STRATEGY

This chapter explains the algorithms adopted by the Loader to optimise the location of compilation units within the object machine and describes the additional code and data areas generated by the Loader.

Instructions generated by the CORAL compiler are not changed by the Extended Loader save to resolve addresses and make them core relative rather than code relative. For details of object code sequences generated by the CORAL compiler for individual data declarations or source statements see chapter 5 CORAL Compiler Users Manual.

5.1 OBJECT MACHINE STORAGE ALLOCATION

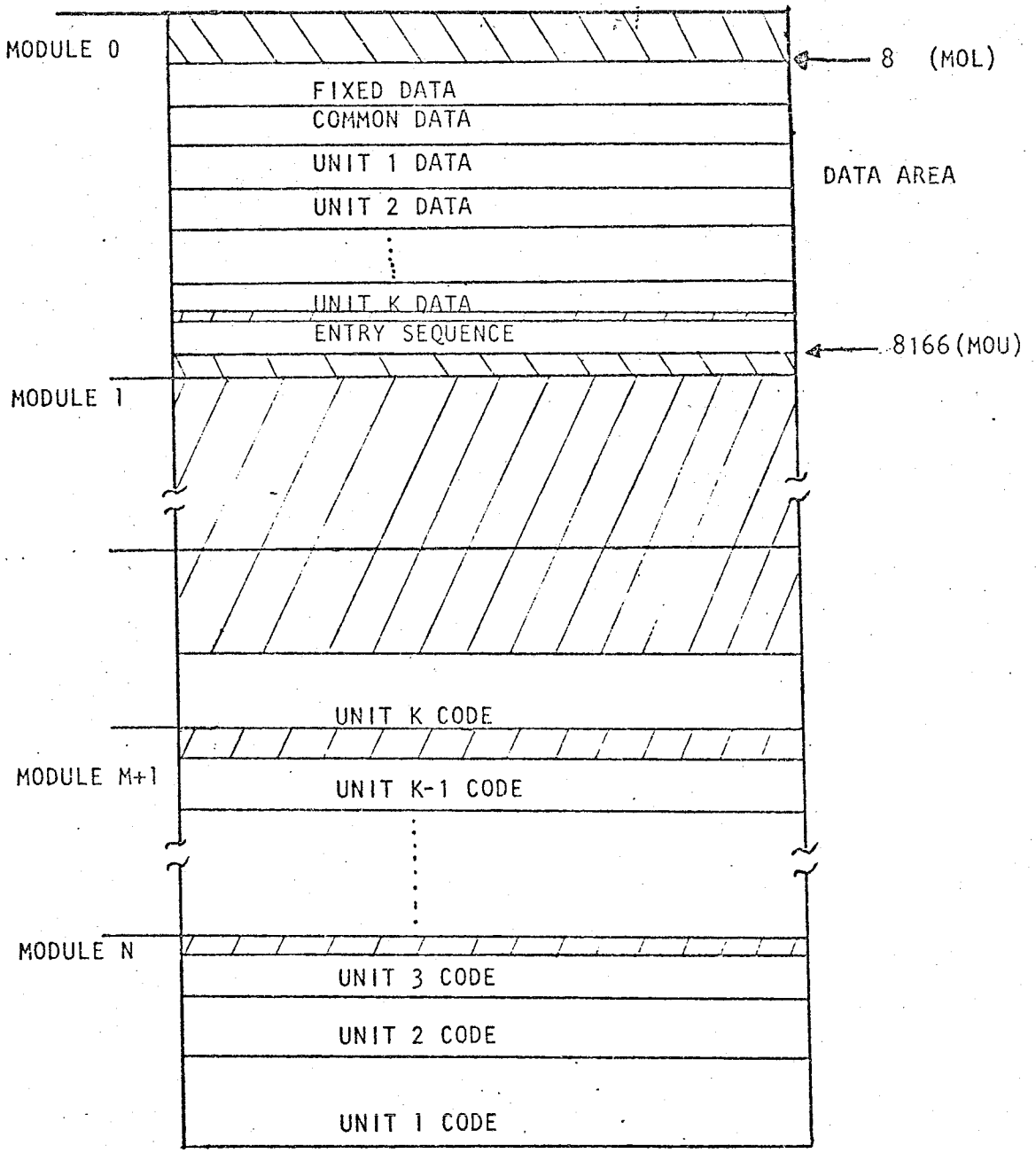
5.1.1 Description

Object code of a CORAL program contains data and executable code held statically at runtime in the object machine core store.

Data resides in store in module 0 and is absolutely addressed. Code may reside in any core store module.

Each unit of compilation comprising a program has a data and/or code area associated with it.

The following diagram describes, in outline only, the runtime storage allocation of a program comprising a number of units of compilation. The diagram assumes that no operator intervention has been taken to adjust the location of the code for a particular unit of compilation and that loader optimisation is inoperative. As will be seen in 5.3 this is a simplified version of the actual loading algorithm.



Simplified Diagram of Object Machine Storage Allocation

Note: address 8 in module 0 equates to the MOL of U.M. 5.1.1. This is a default address and can be varied by user command.

/// unused core areas

prescribed core area

The Loader loads the object program within the module bounds set by the user to describe the intended object machine. Normal default values for these bounds are:

Module 0 lower bound	- 8
Module 0 upper bound	- 8166
Module 1 lower bound	- 8192
Module 1 upper bound	- 16383
⋮	
Module 3 lower bound	- 24576
Module 3 upper bound	- 32767

Commands available for varying the number of object machine modules and the bounds within each module are described in 3.2.1.

In general the data area for each unit of compilation is located at the lowest available address within module 0. A detailed description of the components of the unit data areas is given in 5.2.1. The common data area loaded is that which accompanies the first unit of compilation being loaded and it is assumed that all subsequent Common data areas accompanying the following units of compilation are the same (a limited number of checks are performed by the Loader and are described in 2.6).

The code area for each unit of compilation is loaded from the highest available object machine module until the remaining space within the specified module bounds is too small for the current unit whereupon the next highest module is tried. Remaining space in the highest module is used for a subsequent unit if possible. The Loader maintains an available core map which describes free space in each object machine module and only when the code area of a unit of compilation will not fit in any module does it report that core is full.

5.1.2 Loader Generated Information

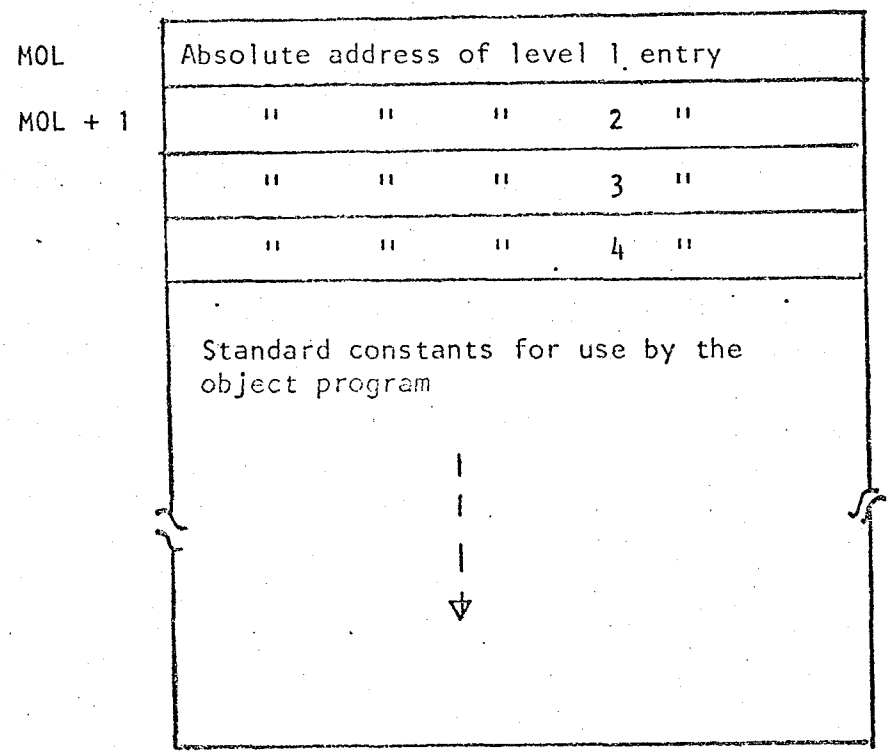
In addition to locating the code and data components of a unit of compilation within the object machine, the loader generates a number of code and data sequences which are necessary ingredients of the loaded program.

Only two of the loader generated components are specifically mentioned in the Object Program Core Map (4.2):-

- Fixed Data Area;
- Program Entry Sequence.

Other code sequences and data areas generated by the loader are included within the bounds of the unit of compilation code and data areas for the purposes of producing the Object Program Core Map.

All the loader generated information is described fully in 5.4, however a diagram of the Fixed Data Area is included here for parity with the Coral Compiler Users Manual.



Loader Generated Fixed Data Area

5.2 DATA SPACE ALLOCATION

The data area of a unit of compilation contains CORAL data overlayed according to block structure and compiler generated data (strings, constants, addresses and workspace). This area may include additional data space generated by the Loader to contain library parameter space, and/or Loader workspace.

Data space allocation for each item of CORAL data is described in U.M. 5.2.

The various sub-divisions of the unit of compilation data area are aggregated by the Loader and are reported on the Object Program Core Map as a single contiguous area.

5.2.1 Data Areas

There are3 classes of data areas allocated by the loader:-

- Fixed Data Area
- Common Data Area
- Unit of Compilation Data Area

The contents of the Fixed Data Area and its use is described in 5.4.1.

Data included within the Common Data Area is wholly defined by the CORAL compiler and described in U.M. 5.2.

Unit of Compilation Data Areas include data as described in U.M. 5.2 and in addition contain space for library procedure linkage and parameters. The allocation of information for communication with external library procedures in this way is wholly under the control of the loader.

5.2.2 Allocation Strategy

As described in 5.1.1, allocation of data areas is confined to module 0 of the object machine and proceeds from the lowest available address to the highest.

When no contiguous free space exists within module 0 of sufficient size to accommodate a data area then the loader reports that the core is full and halts.

Allocation of data areas proceeds in the following order:-

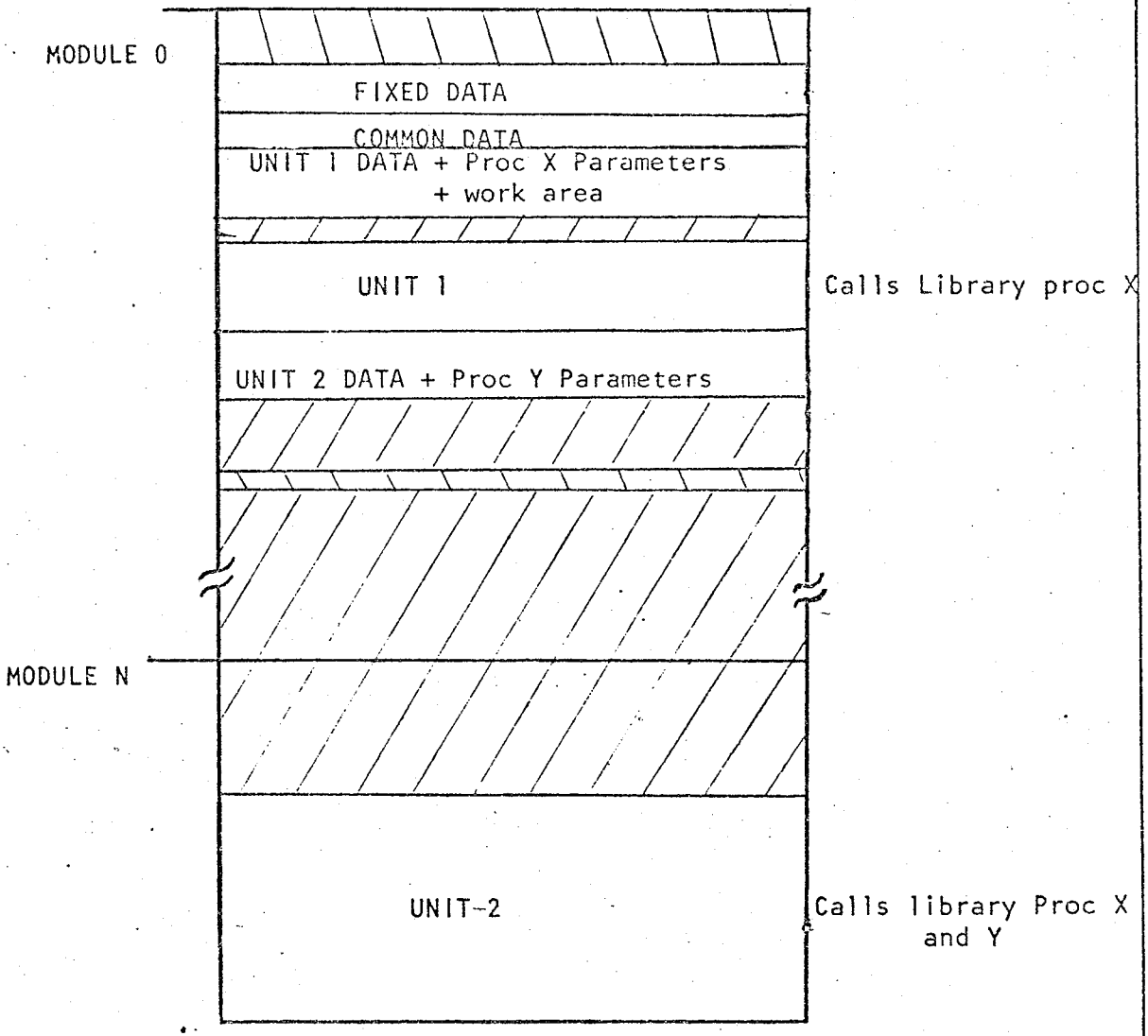
- Fixed Data is allocated when the first unit of compilation is loaded and occupies the lowest available locations in module 0.

- Common Data is allocated from the top of the Fixed Data upwards immediately after allocation of Fixed Data.

- Unit of Compilation data is allocated as a contiguous area in the lowest available area of module 0 at the tiem of allocation. Since code may be allocated within module 0 (5.2.2) unit data may not be contiguous with the last data area allocated.

The size of data area for a unit is a function of the loading order of a program unit. It consists of the data area as generated by the CORAL compiler, plus linkage and parameter space for library procedures referenced by the current unit of compilation which were not referenced by previously loaded units (see U.M. 5.2.3), plus an Inter-module work area if the current unit of compilation is the first on this level.

The following diagram illustrates the strategy described above:



Notes: 1 The unused area between UNIT 1 DATA + Proc. X Parameters and UNIT 1 CODE is not sufficiently large to contain UNIT 2 DATA + Proc Y parameters although it may be large enough for UNIT 2 data on its own.

2 UNIT 2 DATA includes Proc Y Parameters but not Proc X Parameters since since only one allocation is required for library procedure linkage and parameters and this has already been done for procedure X.

5.3 CODE SPACE ALLOCATION

The code areas for a unit of compilation contain executable code for CORAL statements plus switch arrays and jumps for common labels. In addition, the code areas contain Loader generated executable code sequences described in 5.4.

Code generated for each CORAL statement and for switch arrays is described in detail in U.M. 5.3, U.M. 5.4.

5.3.1 Code Areas

Two types of code area exist for each unit of compilation:-

- Code Area
- Switch Area

The bounds of these areas are detailed separately on the Object Program Core Map (4.2) but are considered by the Loader to be contiguous and hence are always allocated as a contiguous area. In all sections of the manual other than 5.3.1.1 and 5.3.1.2, the area described as the Code area or Unit of Compilation code area is taken to mean the combined code and switch areas.

5.3.1.1 Code

The unit of compilation code area contains the executable code generated for the segments of a unit and is loaded such that the first segment in the source listing is loaded at the low end of the code area, and the last segment compiled is loaded at the high end of the area.

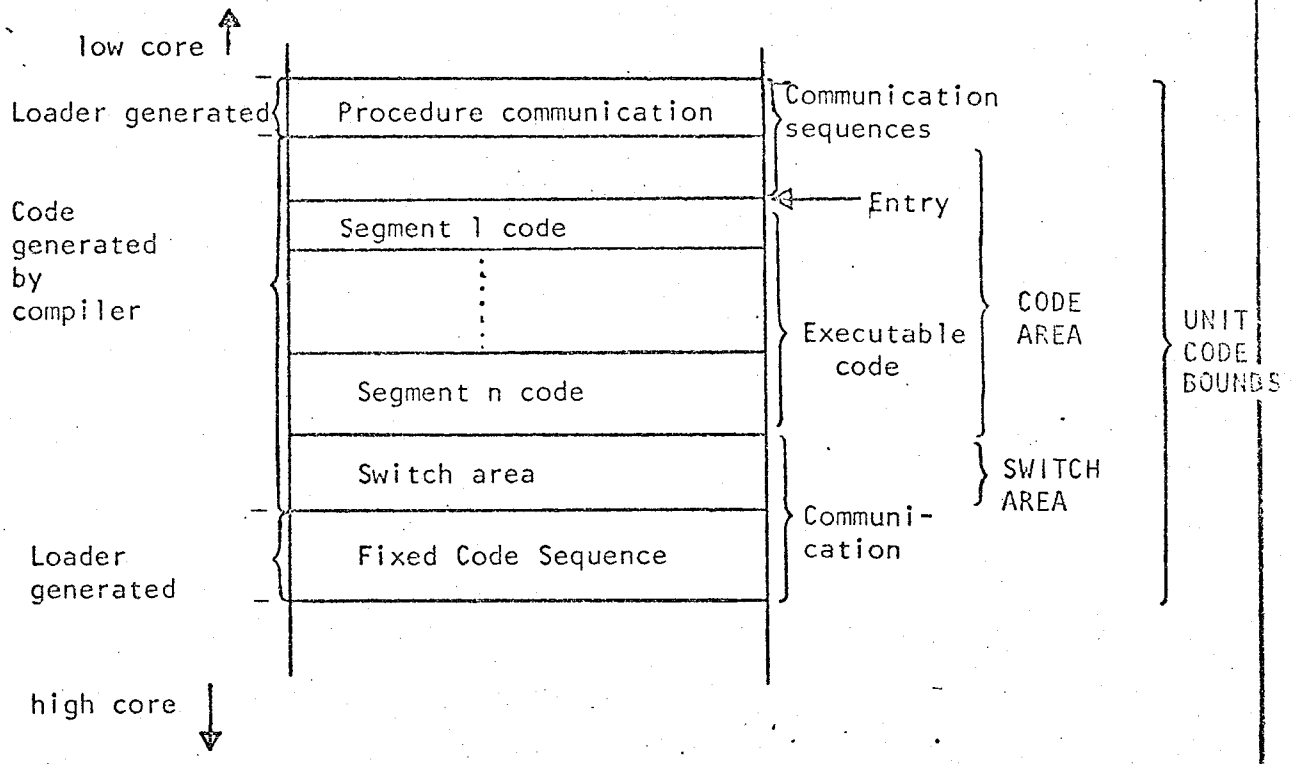
In addition to executable code produced from CORAL statements this area contains:-

-- jumps to Common labels
-- procedure call linkage code.

For each common label declared in the source program, the compiler adds to the low end of the code area a jump sequence consisting of two instructions.

Procedure call linkage code, consisting of two instruction jump sequences, is included at the low end of the code area to facilitate communication with external procedures.

The diagram below demonstrates the structure of the code area:



5.3.1.2 Switches

The unit of compilation switch area reported on the Object Program Core Map consists of the switches generated for the unit by the CORAL compiler (U.M. 5.3) and a Fixed Code sequence generated by the loader.

Fixed Code is not appended to every unit of compilation but is present in every core module of the object machine into which code is loaded. For a multi-level program there is one copy of the Fixed Code for each program level represented in a core module.

Fixed Code is appended to the first unit of compilation (at the current program level) loaded into a core module and is a constant 16 words in length. Section 5.4 describes the code sequences.

The diagram in 5.3.1.1 shows the positioning of the Fixed Code sequence, when present.

5.3.2 Allocation Strategy

As was described in 5.1.1, allocation of code areas proceeds from the highest available object machine address down to the lowest. An estimate of the size of the Unit of Compilation code area (code + switches) is generated by the Loader and a contiguous block of core is allocated at the most suitable location in the object machine.

In the simplest cases, the code area is allocated at the highest available address, but 5.3.2.1 and 5.3.2.2 will show how the Loader and the user may alter this scheme for a more optimum loading pattern.

Irrespective of how the Loader chooses the core module best suited to contain the code area for the current unit, the Loader has to estimate the size of core block needed for the code. 5.3.1.2 described the algorithm for determining whether Fixed Code must be appended to the code area. In addition an estimate is generated for the size of the procedure communication area. This area contains two instructions per procedure referenced by the unit of compilation and not declared within it.

Once an estimate for the code area size has been made and the code area allocated then loading can commence. At this point the area reserved for procedure communication is relinquished since it is not expected that all of the area will be used. In fact only two instructions are required within each core module for

references to procedures external to the module, although estimates are generated on a unit basis.

The diagram below shows how this strategy develops during loading of 3 units into one core module

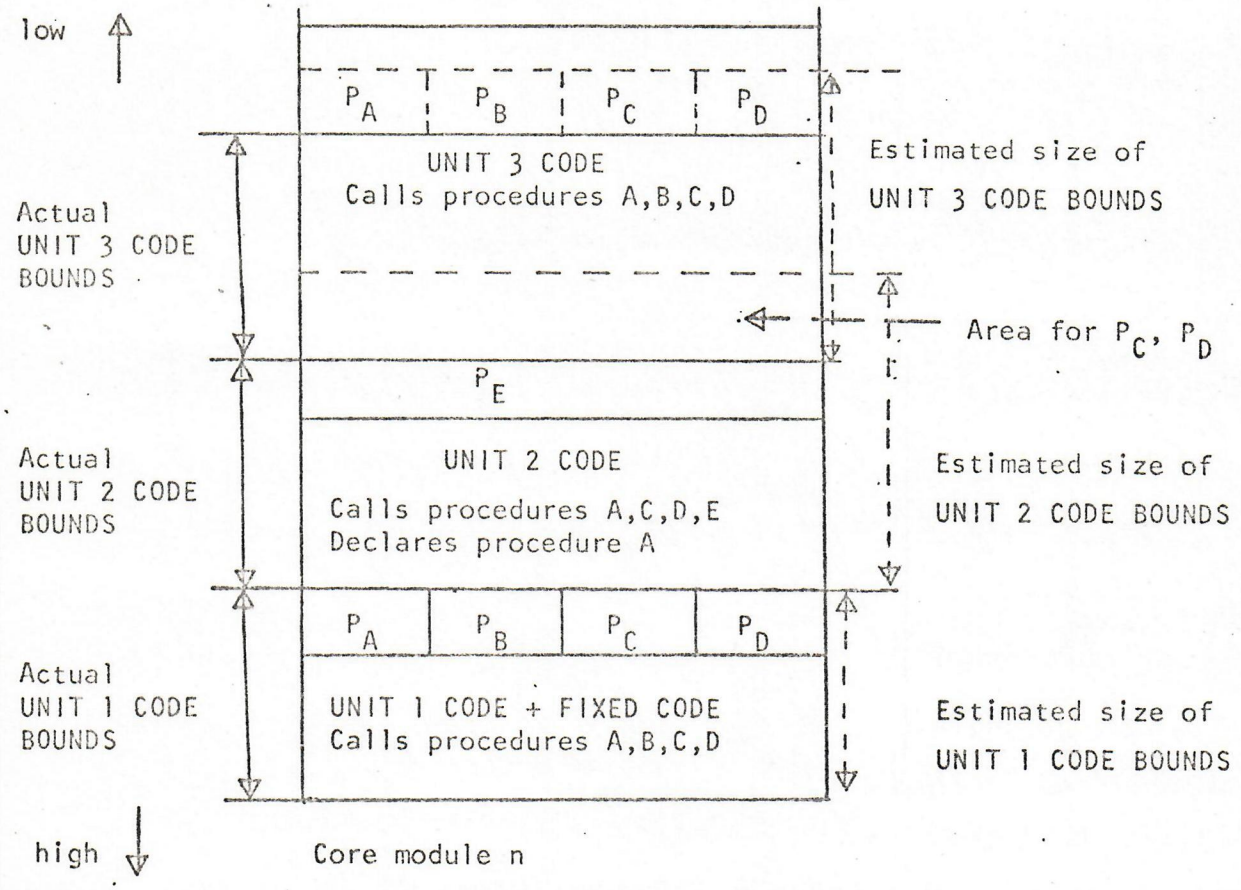


Diagram of allocation of procedure communication areas

Notes 1. Px = 2 instructions used to communicate with procedure x.

2. Fixed code is allocated with UNIT 1 as the first unit in the module.

3. Space for P_A is not included in UNIT 2 estimates as it is declared in UNIT 2. P_A is included in UNIT 3 estimates because the loader does not know where procedure A is declared when estimating.

4. If the available space in module m were not sufficient for UNIT 3 estimated size the unit would be loaded in a different module even if there was space sufficient for the actual UNIT 3 size.

5. The actual redundancy in generated code is only P_A (2 instructions).

The following two sub-sections (5.3.2.1 and 5.3.2.2) describe how the actual core module chosen to contain a unit of compilation differs from the simplified highest to lowest scheme so far described, whether due to user intervention or loader optimisation.

5.3.2.1 Loader Module Allocation Algorithm

The Loader determines the optimum module to contain a unit's code by examining calls to external procedures already loaded in other units of compilation and calls to library procedures declared in the current unit of compilation.

The basic rules to be applied when predicting the location of a unit of compilation are as follows:-

1) The Extended Loader attempts to minimise the number of procedures with inter-module calls, be they Common procedures or Library procedures. Therefore, if the current unit of compilation contains references to a previously declared procedure then the loader gives a one "point" weighting to the core module containing the declaration; this is repeated for each previously declared procedure referenced in this unit. If the unit contains a declaration of a procedure previously referenced the Loader gives a one "point" weighting to each core module containing a reference to the procedure; this is repeated for each procedure declared in this unit and referenced externally. The Loader has thus set up a Module Preference Table for this unit which will decide the location of the unit subject to the further rules:

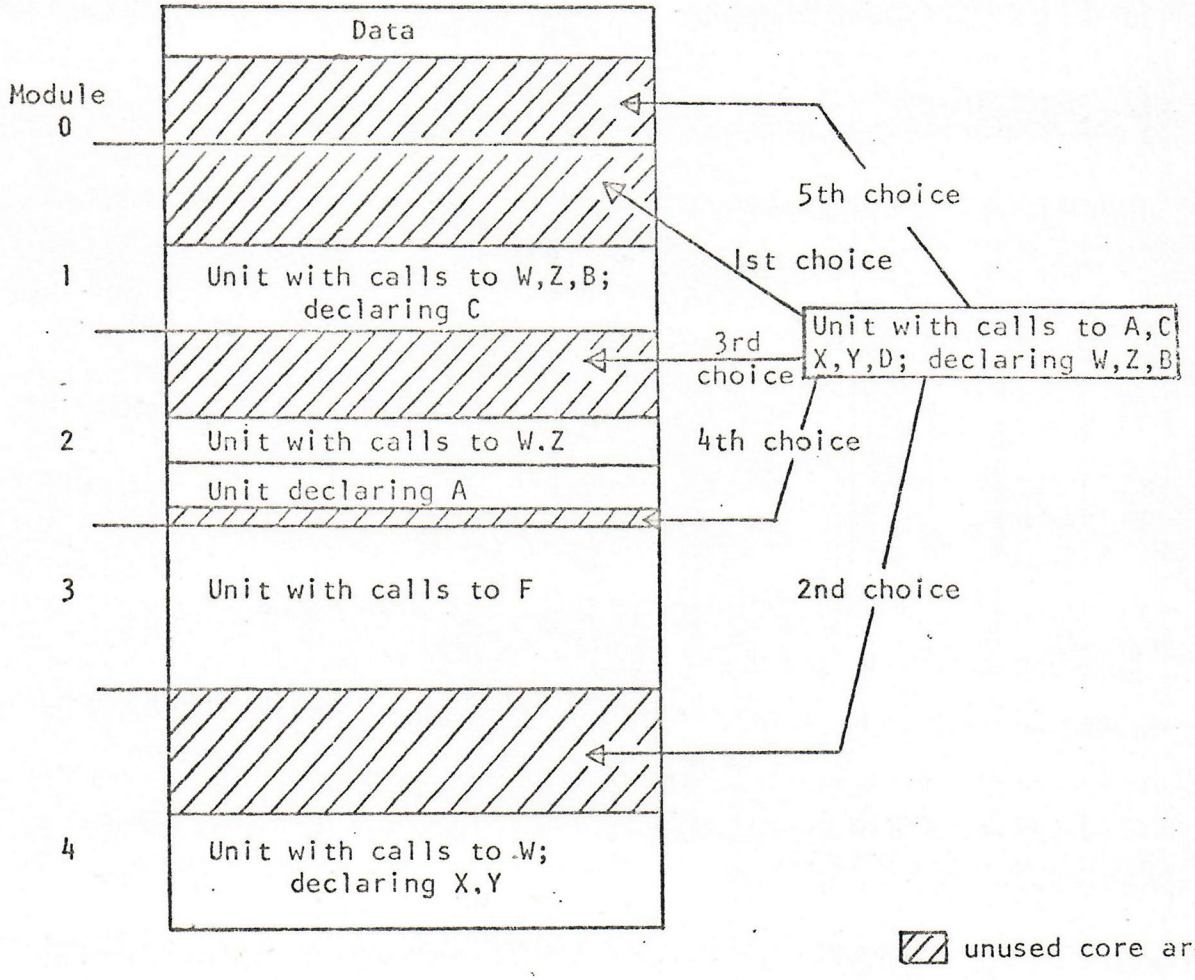
2) Where a unit of compilation has equal weightings of preference for two or more modules the highest module is always preferred.

3) The core module must have a contiguous free area of core large enough to take the whole of the unit's code as defined in 5.3.1, where the procedure call linkage area is the maximum, but not necessarily actual, size of the area.

4) The area chosen, if there is more than one free block in the module, will be in the lowest block of large enough size.

5) The code block will be located at the high core end of the chosen free core area.

The following example demonstrates the use of these rules. With the layout on the left, the Loader selects a module for the unit on the right.



Applying the above rules, it is clear that the modules have a preference weighting of 0:0, 1:4, 2:3, 3:0, 4:3.

Module 1 is thus the optimum module, and the unit will be loaded immediately below the unit already present, if there is enough room.

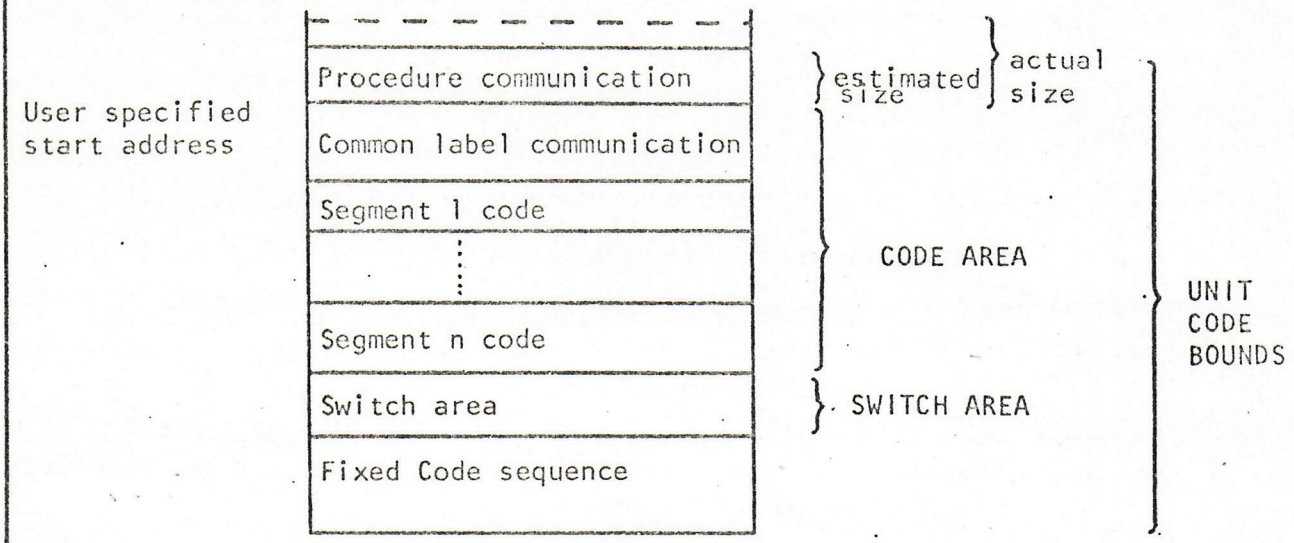
If the block was too big to fit into the module 1 free area, and the area in 4 was large enough the unit would be loaded there, and so on, as shown in the diagram.

5.3.2.2 User Module and Address Allocation

The user can determine the module into which the current unit is to be loaded by the use of the ADDress option, see 3.2.2. The restricting factor is that there must be room in the module for the ESTIMATED SIZE defined in 5.3.1. If there is room the Loader will locate the unit's code at the highest available address in the First Free block available in the module requested.

This option, of defining the module into which a unit of compilation is to be used, may be used to locate units with references to the same procedures in the same module, while still leaving room in other modules for library procedures and/or coordination with the normal Loader optimisation, as described in 5.3.2.1, the optimum core usage can be attained.

The user may determine the exact start address of a unit of compilation's code area, by using the ADDress option (see 3.2.2) and specifying the module, and the offset within the module. The diagram below demonstrates that the address specified is the base of the Code area (as printed in the core map) and NOT the unit code base (lower bound of Unit Code Bounds), since this may be ambiguous.



It should be remembered that the Fixed Code Sequence is only included on the occasion of loading the first unit on a particular level. It is a fixed sixteen words in length.

The CODE and SWITCH areas are of fixed length for each unit of compilation.

The procedure communication area estimated size is two words for each procedure referenced, but not declared, in the unit of compilation. This figure is used when checking that the area specified by the user is large enough. If it is, the procedure communication area is deallocated and two words are allocated as necessary, for each procedure referenced for the first time, but not declared, in the core module selected. The actual size of the procedure communication area is the sum of these two word links, and this area is included in the Unit Code Bounds.

NOTE: Each library procedure is treated as a separate unit. If the user wishes to keep complete control of library allocation, each library procedure should be on a different RLB tape. The ADDress option can only affect the address of the first library procedure on an RLB tape.

5.4 LOADER GENERATED CODE & DATA

In addition to the object code and data produced by the Compiler, the Loader inserts code and data, where necessary, as detailed below.

5.4.1 Fixed Data Area

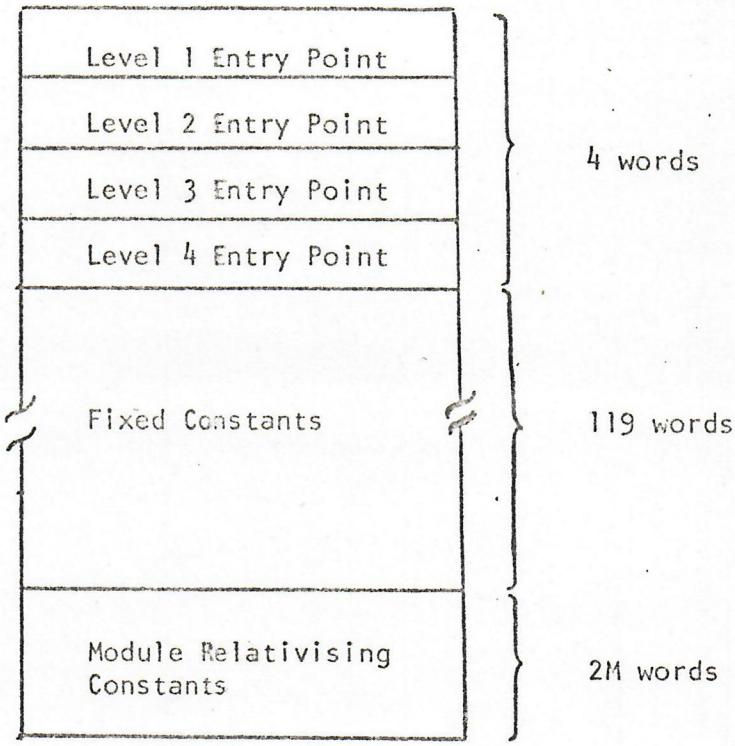
The first data area to be allocated in module zero is a Fixed data area. This is of length 123 words plus 2 words for each object machine core module. This can be split as shown below, into 4 level entry point words, 119 constants and 2 words/ module.

The first four words contain the object program entry points for hardware levels 1 to 4 respectively. The level 2 to 4 entry points are always set to -1 in the case of single-level programs.

The next 119 words are fixed constants used by the Compiler to save some multiple constant generation.

The final section in this area consists of module-relativising constants. Each pair consists of a constant which is used to convert an address from module relative to absolute, and one which can convert absolute to module relative. e.g. for module 2 (instruction format) 2 0 and /14 0.

The following diagram summarises the above:



(m modules in object machine)

5.4.2 Procedure Call Sequences

If a procedure is called within the same module in which it is declared the communication can be made in the following way:

```
STS LINK
J   ENTRY
```

where LINK and ENTRY are those printed in the core map, save that ENTRY is, of course, module relative.

If the procedure declaration is in a different module to the reference, the communication can be made using the sequence:

```
STS  PCLINK
J    PCENT
```

Where PCLINK is the first word of the 3-word area allocated with the first data area for this level (see 5.2.2). PCENT is the entry point of the two-word Inter-module communication sequence (procedure communication sequence of 5.3.2.2) for this procedure in this module, defined as

```
PCENT    LDB    PCO
         J      IMENT
```

where PCO is the first word of the procedure communication block (U.M. 5.2.3.2), and IMENT is the entry point of the inter module communication code in this module for the current level.

The latter is the first 13 words of the 16 word fixed code sequence (see 5.3.2.2) viz:

```
ST      WS2       (save accumulator)
/LD     1         (load entry address)
ST      WS1       (store entry address)
COL     &360000   (highlight module bits)
NEG     PCLINK    (load module-relative link)
ADD     MODBIT    (Add current module bits)
/LDB    0         (Load B register with address of link)
/ST     0         (Store return address in link)
LD      MODBIT    (Load current module bits)
NEG     WS1       (Create module-relative entry address)
ST      BREG      (copy to current level Bregister)
LD      WS2       (Restore accumulator)
/J      0         (Jump to procedure)
```

where PCLINK, WS1, WS2 make up the three word work area (5.2.2)

5.4.3 Label and Switch sequences

The remaining three-words of the 16 word Fixed Code area on each level in each module are:

```
ADD   MRELAD
ST    BREG
/J    0
```

where MRELAD is taken from the second of the modules module relativising pair at the end of the Fixed Data Area, and BREG is the B register location for the current level.

This is used for intermodule jumps to make the address module relative.

5.4.4 Program Entry Sequence

After the END = YES command has been used a program entry sequence is produced and located in the top free area of module zero, so that the program may be triggered from a module zero address. The sequence for a single level program is

```
SAB          (replaced by SH 0 for 8K machines)
LDB  ENTAD
/J   0
```

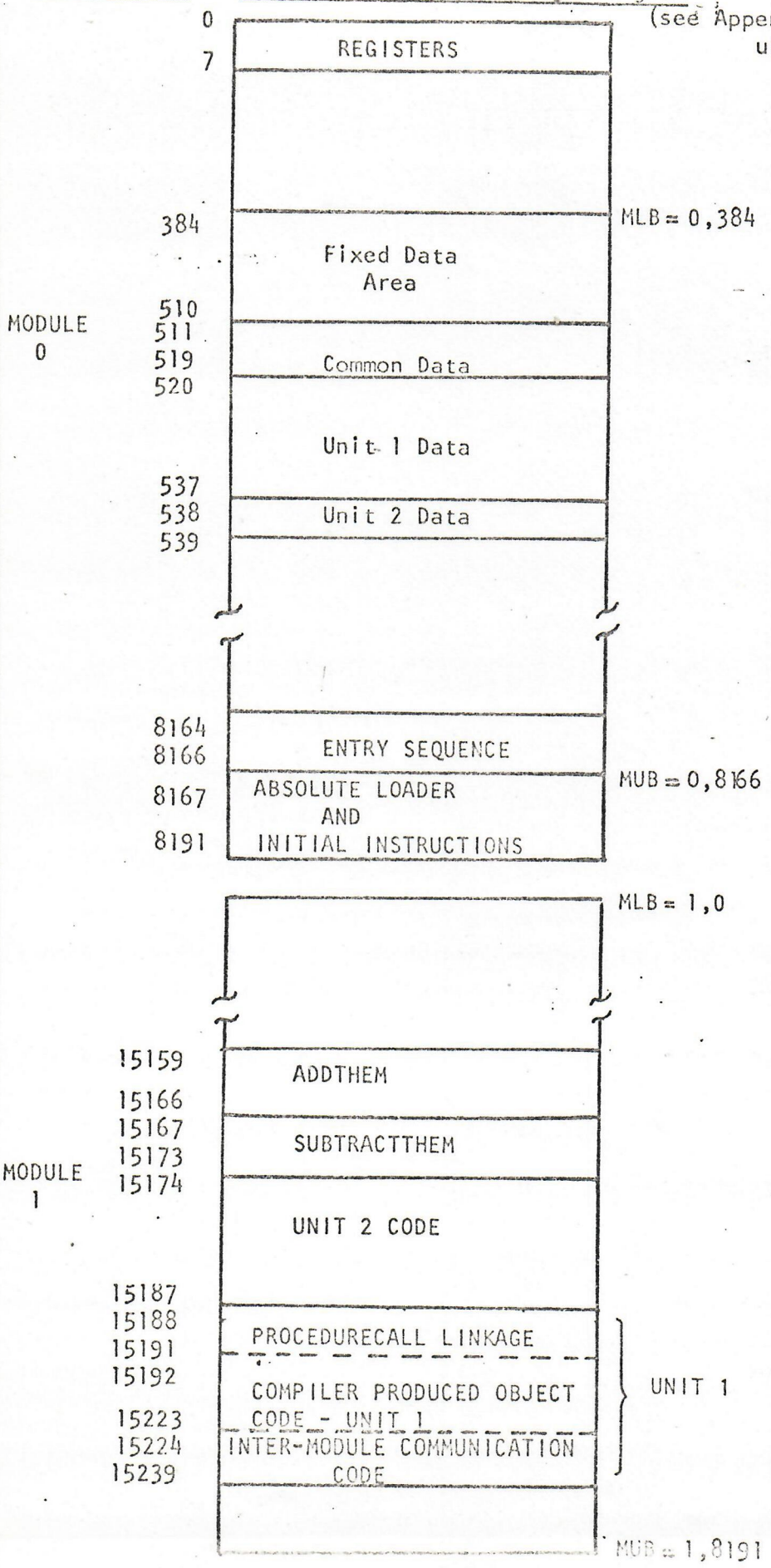
where ENTAD is the level 1 entry point in the Fixed Data area. For multi-level programs, the code is dependent on the program entry level
E.g. for level 4:

```
LD      ENTAD
ST      0
LD      ENTAD + 1
ST      2
LD      ENTAD + 2
ST      4
SAB             (replaced by SH 0 for 8K machines)
LDB     ENTAD + 3
/J      0
```

Appendix A Table of User Manual Sections amended or amplified by this Document

User Manual Reference	Operational Documentation Reference
1.1.1	1.1
1.1.1.4	1.2
1.2.3.1	1.3.1
1.2.3.2	1.3.2
1.2.3.3	1.3.3
2.1.5	2
2.1.5.1	2.1
2.1.5.2	2.2
2.1.5.3.1	2.3.1
2.1.5.3.2	2.3.2
2.1.5.4	2.4
2.4.1	2.5.1
2.4.2	2.5.2
2.5.1	2.6
3.1.5.1	3.1
3.1.5.2	3.2
3.1.6	3.3
3.3.1	3.4
4.1.5.1	4.1
4.1.5.2	4.2
4.4.1	4.1.2
5.1	5.1
5.2	5.2
5.3	5.3
5.4	5.4
Appendix D	3.1/3.3
Appendix E	Appendix C
Appendix G	Appendix D

Appendix B Object Machine Core Usage Diagram



Appendix C (E) Example of Loading Information

```
*AUT = YES
*COR = 2

   CORE AVAILABLE

MODULE  BOUNDS
000000  000008 - 008167
000001  008192 - 016383
*MLB = 0,384
*OG
>>401 COMMAND UNKNOWN
*ADD = 1,7000
*GO
PROGRAM EXAMPLE CORAL
  FIXED DATA AREA          000884 - 000510
  COMMON AREA              000511 - 000519
  DATA AREA                000520 - 000537
  CODE AREA                015192 - 015223

SEGMENT DEMON2
SEGMENT DEMON1
COMMON  PLACE

UNIT CODE BOUNDS           015188 - 015239

UNDECLARED LIBRARY PROCEDURES

*UND
  UNDECLARED PROCEDURES
  LIBRARY SUBTRACTTHEM
  LIBRARY ADDTHEM
 *GO
PROGRAM EXAMPLECODE
    DATA AREA              000538 - 000539
    CODE AREA              015176 - 015187

    SEGMENT DEMON3
    COMMON  PLACE

    UNIT CODE BOUNDS       015174 - 015187

UNDECLARED LIBRARY PROCEDURES

*O<-GO
LIBRARY EXAMPLECORAL

LIBPROC SUBTRACTTHEM
  CODE AREA                015167 - 015173

 PROCEDURE SUBTRACTTHEM    ENTRY  015167
                           LINK   000528

  UNIT CODE BOUNDS         015167 - 015173
```

```
LIBPROC ADDTHEM
     CODE AREA                    015159 - 015166

PROCEDURE ADDTHEM                 ENTRY    015159
                                  LINK     000534

UNIT CODE BOUNDS                  015159 - 015166

*END=YES

PROGRAM ENTRY SEQUENCE            008164 - 008166
          AT LEVEL                000001
          CODE CHECKSUM           247476 (OCTAL)
```

Appendix D - Minimum Operating Instructions

The following instructions assume that the user is using all the default options.

Single level

1) Load Extended Loader under hardware initial instructions
2) Trigger to 8.
3) Load 1st program tape in reader
4) Type GO cr
5) Repeat 3 and 4 for each program tape
6) Repeat 3 and 4 for each library tape
7) Type END = YES cr
8) Tear off tape
9) Load program binary tape in object machine under Initial Instructions.
10) Trigger at entry point specified by core map.

Multi-level

1) Load Extended Loader under Initial Instructions
2) Trigger to 8.
3) Type LEV = n cr for level n programs (n = 1,2,3,4)
4) Load 1st program tape on this level in reader
5) Type GO cr
6) Repeat 4 and 5 for each program tape on this level
7) Repeat 4 and 5 for each library tape on this level
8) Repeat 3-7 for each level
9) Type END = YES cr
10) Tear off tape
11) Load each program binary tape in object machine under Initial Instructions (not reading the legible tape number)
12) Trigger at entry point specified by core map.