Oxford University Computing Laboratory

Computer Manuals

4112

SCIENTIFIC LANGUAGES

ALGOL- Paper Tape Compilers

1900

ICL

International Computers Limited
©

User notice

MANUAL (NOTICE NO.) 23/1/69

4112 ALGOL: PAPER TAPE COMPILERS (1)

File one copy of this notice with each of the manuals indicated.

THE 'SPACE' STATEMENT

To the description of the 'SPACE' program description line on page 27 add:

s must be greater than 10.

ERROR NUMBERS

There is an additional compiler error number, which may be added to the list on page 53.

101 'LABEL' has been used as a declarator.

Execution error number 52 on page 55 should be deleted. This error is covered by the HALTED:-RL message - see page 48.

LISTING LINE LENGTH

The 'NOLIST' statement described on page 13 may also take the form 'NOLIST' (XX/n) to describe line length. For this statement and for 'LIST' (page 11) and 'SHORTLIST' (page 12), n must be at least 96 if input is from cards.

TYPOGRAPHICAL ERRORS

On page iii, line -3, the last word should be "manual".
On page 17, line -6, the reference should be to page 29.
On page 33, in the <u>comment</u> line of example 2, insert the symbol ↑ between (-1) and i.

OXFORD UNI.E . I Y COMPUTING LABORATORY

COMPUTING SERVICE

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1DX

FORM 1/230/45(10.68)

ICL International Computers Limited ©

User notice

MANUAL (NOTICE NO.)

19/2/69

4112

ALGOL: PAPER TAPE COMPILER (2)

File one copy of this notice with each of the manuals indicated.

MIXED LANGUAGE PROGRAMMING

PLAN Procedure Segments

Note that at the time of the OBEY to obtain a parameter address in a PLAN procedure segment, the accumulator X1 must still contain the link. Thus, when a parameter is being picked up, the PLAN sequence

```
LDX  2   1
OBEY    (2)
```

is equivalent to

```
OBEY    (1)
```

but the sequence

```
LDX  2   1
LDX  1   A
OBEY    (2)
```

is not, and may produce erroneous results.

OXFORD UNIVERSITY COMPUTING LABORATORY
COMPUTING SERVICE

FORM 1/230/45(10.68)

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1DX

ICL International Computers Limited ©

User notice

MANUAL (NOTICE NO.)

14/5/69

3340 ALGOL (20)

4112 ALGOL: PAPER TAPE COMPILERS (3)

File one copy of this notice with each of the manuals indicated.

SRA1/2 SUBROUTINE GROUP

The SRA1 subroutine groups (now only used by the Algol paper tape compilers #XALP and #XASP) is reissued. As stated in the User Notice Algol (10) all magnetic tape and disc procedures have been withdrawn from the group.

© International Computers Limited, Reading, 1969.

FORM 1/230/45(3.69)

OXFORD UNIVERSITY COMPUTING LABORATORY
COMPUTING SERVICE

Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

ICL International Computers Limited

User notice

MANUAL (NOTICE NO.) 9/7/69

4129	1900 ALGOL: 16K DISC COMPILER (7)
4122 X	1900 ALGOL: MAGNETIC TAPE COMPILER (5)
4112	1900 ALGOL: PAPER TAPE COMPILERS (4)
3340	1900 ALGOL MANUAL (24)

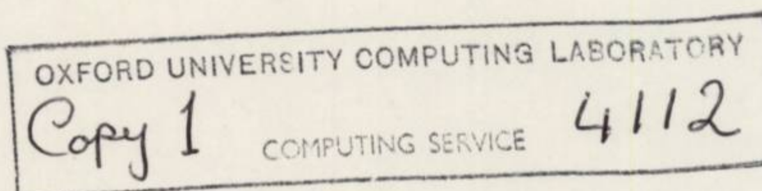
File one copy of this notice with each of the manuals indicated.

SEMICOMPILED INPUT TO 1900 ALGOL COMPILERS

Since the Program Description of an Algol program consists of instructions to the Compiler, semicompiled program has no place in this segment. The end of the P.D. is signalled either by the first word of source program ('BEGIN', 'PROCEDURE' etc.) or by 'CONTINUE'. Thus if only semicompiled program is to be input the corresponding 'READFROM' statement(s) should be preceded by 'CONTINUE' and followed by 'FINISH'.

© International Computers Limited, Reading, 1969.

FORM 1/230/45(3.69)



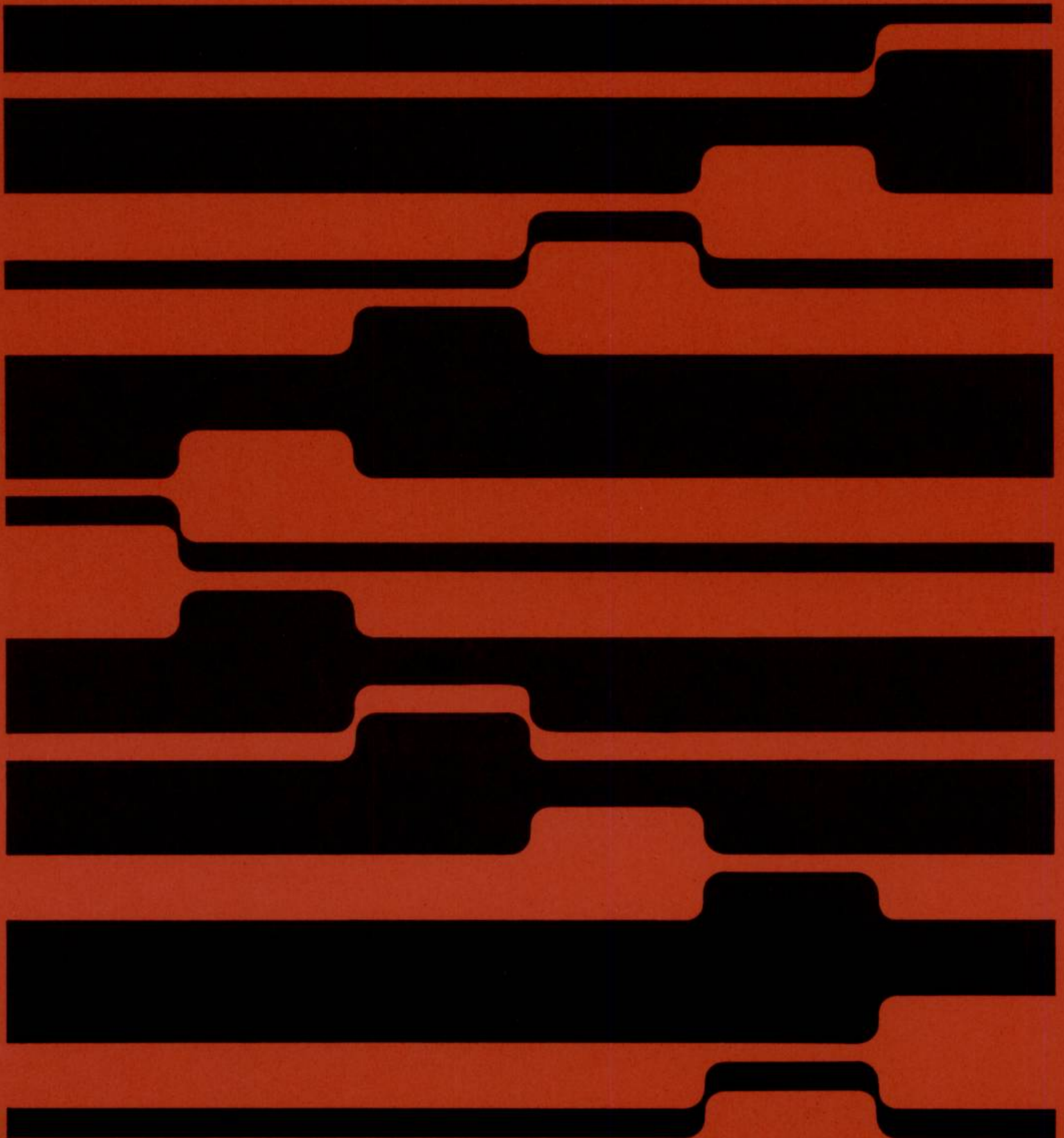
Issued by Technical Publications Service, 30 - 31 Friar Street, Reading, Berkshire. RG1 1JP

ICL

Algol: paper tape compilers

1900 Series

OXFORD UNIVERSITY COMPUTING LABORATORY
Copy 1
COMPUTING SERVICE 4112.



ICL

Algol: paper tape compilers

1900 Series

OXFORD UNIVERSITY COMPUTING LABORATORY
Copy 1 COMPUTING SERVICE 4112

With effect from 9th July 1968 the name of
International Computers and Tabulators Limited
has been changed to
International Computers Limited.

Technical Publication 4112

© International Computers Limited 1968

First Edition October 1968

Issued by Technical Publications Service
International Computers Limited
Head Office: ICL House, Putney, London SW15
and printed in Great Britain by
ICL Printing Services, Letchworth, Hertfordshire

Preface

The Algol paper tape compilers, XALP and XASP, are paper tape versions of the 1900 Algol compiler. They compile Algol programs held on cards or paper tape and produce output on paper tape and a listing on a line printer or a paper tape punch. No magnetic medium is used by the compilers. This manual describes the system of compilation used by the compilers and the facilities offered by them.

The minimum configuration required by XALP and XASP is as follows:

a central processor with floating point facilities (extracode or hardware)

26, 624 words of core store for XALP
or
13, 560 words of core store for XASP
one paper tape reader or card reader
one paper tape punch
one line printer (as an optional listing device)

The manual is intended for use by programmers who are fully conversant with 1900 Algol. Part 2 is meant to be used by operators involved in the compiling and running of Algol programs.

Part 1 describes the considerations to be taken into account when writing an Algol program for compilation by XALP or XASP. Both compilers can be used with a program description which enhances their scope and efficiency. Programmers who do not wish to use program descriptions need read only Chapter 1. Chapters 2 and 8 describe the statements that can be included in the program descriptions (Chapter 7 gives details of program testing facilities that are available). Chapter 9 explains how PLAN and FORTRAN segments can be incorporated into Algol programs.

Part 2 describes the operating system used by the compiler and gives the operating instructions for compiling and running programs using XALP and XASP.

Two other I. C. L. manuals should be used in conjunction with this manual. The manual *Algol: Language* contains the Algol Report and describes 1900 Algol. The manual *Algol: Compiler Library Procedures* describes the Algol subroutine library that is used by the paper tape compilers.

Contents

	Page
Introduction	1
ALGOL PAPER TAPE COMPILERS	1
PART 1: PROGRAMMING CONSIDERATIONS	
Chapter 1: Simple usage	3
Chapter 2: Program description	5
ORDER OF STATEMENTS	5
Preceding the program	5
Between segments	5
BRIEF DESCRIPTION	5
Chapter 3: Object program input/output	7
PERIPHERAL DESCRIPTION STATEMENTS	7
'INPUT' statement	7
'OUTPUT' statement	7
'OMIT IO' statement	8
Chapter 4: Listing	9
LISTING	9
Full listing	9
Intermediate listing	11
Basic listing	11
LISTING STATEMENTS	11
'LIST' statement	11
'SHORT LIST' statement	12
'NO LIST' statement	13
Intersegment listing statements	13
'TAB' statement	13
COMPILATION ERROR MESSAGES	13
Chapter 5: Compiler output	15
'SEND TO' STATEMENT	15
'PROGRAM' AND 'SEGMENTS' STATEMENTS	15
'PROGRAM' statement	15
'SEGMENTS' statement	16
'COPY' STATEMENT	16

Chapter 6 : Compiler input 17
PROCEDURE SEGMENTS 17
MULTI-REEL PROGRAMS 18
'READ FROM' STATEMENT 18
'CONTINUE' AND 'FINISH' STATEMENTS 19
'CONTINUE' statement 19
'FINISH' statement 19
'LIBRARY' STATEMENT 19

Chapter 7 : Program testing 21
'TRACE' STATEMENT 21
TRACE LEVEL 0 22
TRACE LEVEL 1 22
Error numbers 60 and 61 22
Error number 62 22
TRACE LEVEL 2 22
PROGRAM TESTING PROCEDURES 23
The mplist procedure 24
The mpname procedure 24
The pause procedure 24
The overflow procedure 24
SYSTEM SWITCHES 25
The on procedure 25
The off procedure 25
The test procedure 25
The monoutput procedure 25

Chapter 8 : Miscellaneous statement 27
'PRIORITY' STATEMENT 27
'SPACE' STATEMENT 27

Chapter 9 : Mixed language programming 29
PROCEDURE SEGMENTS 29
Declaration 29
Communication of data 29
Procedure libraries 29
ALGOL PROCEDURE SEGMENTS 30
Declaration 30
Procedure 30
FORTRAN PROCEDURE SEGMENTS 30
Declaration 31
Calling by name 31
Procedure 31
PLAN PROCEDURE SEGMENTS 32
Declaration 32
Procedure 32

SEGMENT NAME 32
PICKING UP PARAMETERS 32
Integer 32
Real 33
Boolean 34
Array 34
String 37
Label 37
Switch 38
Procedure 39
Example 40
PROCEDURES USING UPPER COMMON VARIABLES 41
CONSOLIDATION 42

PART 2 : OPERATING CONSIDERATIONS

Chapter 10 : System description 43
LISTING 43
ERROR MESSAGES 44
ENTRY POINTS 44

Chapter 11 : Operating instructions 45
ERROR HALTS 46
Compilation 46
Execution 47
OTHER CONSOLE MESSAGES 49

APPENDICES

Appendix 1 : Compilation error numbers 51
SYNTACTIC ERRORS 51
SEMANTIC ERRORS 52
OTHER MESSAGES 54

Appendix 2 : Execution error numbers 55

Appendix 3 : 1900 Series paper tape and card codes 57

Appendix 4 : Algol hardware representation 59

Index 61

Introduction

Although a computer can only obey instructions written in its own numerical machine code, programs are usually written in a convenient programming language, such as Algol. A program written in one of these languages is called a *source* program and it must be translated into an equivalent machine code program, known as an *object* program, before it can be run on a computer. The process of translating a source program into an object program is known as *compilation* and is performed by a *compiler*, a program supplied by the manufacturer. After compilation, the object program is loaded into store and its instructions are obeyed.

In common with all other 1900 Series compilers, the Algol compilers do not translate Algol source programs directly into an object program but first translate the source program into *semi-compiled* form, an intermediate state between source language and machine code.

Algol compilers normally translate the whole of an Algol program between the first **begin** and its associated **end** into one semi-compiled segment, known as the *master segment* . It is possible, however, to split up an Algol program so that more than one semi-compiled segment is produced from the source program. This is known as *segmentation* .

The advantages gained from the segmentation of Algol programs are as follows.

1 An Algol procedure that is required in more than one program can be translated into semi-compiled form once and incorporated into each program that requires it. Semi-compiled segments require less machine time to be input than the corresponding source program.

2 During the development of a multi-segment program, any segments that are fully tested can be held in semi-compiled form to avoid recompilation when changes are made to other segments of the program.

3 During compilation, the compiler sets up lists for certain categories of information. For complex programs, these lists may become too large for the available store. If the program is written in segments, this problem is overcome because each segment sets up its own lists and does not require the lists for previous segments.

4 Segments can be written in another language and incorporated into an Algol program in semi-compiled form. Similarly, Algol segments can be incorporated into a program written in another language.

There can be only one master segment, other segments being in the form of procedures. Full details of procedure segments are given in Chapter 9, page 29.

The semi-compiled segments, together with any subroutines requested from the Algol library (a group of standard subroutines used frequently in Algol programs and held in semi-compiled form - see Chapter 5, page 15) are then combined to form *consolidated* semi-compiled data, consisting of:

the semi-compiled program,
a request slip (containing information such as the program name, priority number and store space used),
the General Purpose Loader (a program in binary object form used to load the program into store),
consolidated leader information (used by the General Purpose Loader in loading the program).

The final conversion of the program into a binary object program is effected by the General Purpose Loader.

ALGOL PAPER TAPE COMPILERS

The Algol compilers XALP and XASP are held on paper tape and will compile programs read from paper tape or cards. They write semi-compiled output (optionally consolidated into a loadable program) to paper tape. The Algol library used in conjunction with these compilers is held on paper tape.

Algol programs held on cards can only use the 1900 Series 64-character card code and a hardware representation has been chosen for the Algol basic symbols which uses these characters. Programs on paper tape can use the same representation (*normal mode*) or an alternative compiling mode which allows the use of the lower case letters in the paper tape code (*full mode*).

A table of 1900 Algol hardware representations is given in Appendix 4, page 59 .

Compilation is initiated by the operator message GO # XALP (or XASP) NN where NN is an *entry point*, the word at which the compiler is entered.

The course of compilation is directed by program description statements which appear before the program or between segments.

Besides occupying different amounts of core store, XALP and XASP differ only in the number of program description facilities available - there are fewer facilities available with XASP.

PART 1 PROGRAMMING CONSIDERATIONS

Chapter 1 Simple usage

After XALP and XASP have been entered at one of the appropriate entry points, the Algol source program is read and the semi-compiled program is output to paper tape. A *listing* is produced on the line printer. This listing consists of a copy of the source program, information about errors detected during the compilation and other details, such as the time (if available), date, compiler used and core store requirement of the object program (if appropriate).

If no errors are found during compilation, the consolidation phase is entered automatically and the request slip, General Purpose Loader (G. P. L.) and consolidated leader information are output on paper tape. If there are compilation errors, consolidation does not take place unless it is decided to *force* consolidation by entering the compiler at a special entry point. Consolidation is usually forced if it is known that there are compilation errors which will not affect the purpose of the run.

When an error is detected, the compiler will usually ignore the statement in which the error has occurred. Some errors may cause the compiler to misinterpret subsequent program statements and, therefore, a genuine error message may be followed by spurious messages. Thus, if the cause of some reported errors can be discovered while other messages seem meaningless, as many errors as possible should be corrected and the program recompiled. Compilation will be abandoned if 40 errors have been detected because the program would be so much in error that no useful information would be obtained by continuing the compilation.

Although the compilers perform a standard sequence of operations, it may often be convenient to inhibit, redirect or alter some of these operations. In order to allow the programmer some control over the compilation process to suit particular requirements, the compiler can be steered by *program description statements* which appear in a program description before the Algol program or between segments of a multisegment program.

Each line of a program description gives information about a particular function of the compiler. If a program description statement line is omitted, the standard procedures will be adopted. An error in a program description statement will usually mean that the compilation cannot be continued.

For example, the 1900 Algol input/output procedures require Algol programs to specify *channel numbers* for input and output peripherals used by the program. If the relevant peripheral description statement line is omitted, the following channel numbers are assigned:

input channel	*peripheral*	*unit*
0 and 1	paper tape reader	0
2	paper tape reader	1
3	card reader	0

output channel	*peripheral*	*unit*
0 and 2	line printer	0
1	paper tape reader	0
3	card punch	0
4	paper tape punch	1
5	line printer	1

If other channel numbers are selected, the appropriate program description statement must appear, informing the compiler of the desired channel numbers (see Chapter 3, page 7).

It is important to note that multi-segment programs will only be compiled if the relevant program description statements appear (see *'CONTINUE' AND 'FINISH' STATEMENTS*, page 19). If these statements do not appear, the program will be regarded as consisting of a single source segment.

Program description statements can also be used to produce an abbreviated listing, on paper tape if necessary; to allow segments in the same program to be read from cards or paper tape, in source or semi-compiled form; to expand the error detection facilities and to provide the other facilities described in the remaining chapters of Part 1.

If no program description appears, the program will be assigned the name AXXX and will be given a priority of 50.

Chapter 2 Program description

The compilation process can be directed by a *program description* which consists of steering information read by the compiler before the Algol program or between segments of a program. Program description statements are punched one per line (not separated by semi-colons) and each statement line begins with the statement enclosed in apostrophes. The program description statements are punched in accordance with the 1900 hardware representation of Algol basic symbols described in Appendix 4, page 59, but peripheral identifiers used in statement lines (e.g. TR - for paper tape reader; NONE - output to be suppressed) must be punched in upper case letters.

ORDER OF STATEMENTS

Although statements can be omitted from the program description, those statements that are included must appear in the following order (* indicates that the facility is not available with XASP).

Preceding the program

listing statement line ('LIST', 'SHORT LIST' or 'NO LIST')

*'TAB' *statement line*

'SEND TO' *statement line*

'PROGRAM' or 'SEGMENTS' *statement line*

'COPY' *statement line*

* *Peripheral description statement lines* ('INPUT', 'OUTPUT', 'OMIT IO')

*'PRIORITY' *statement line*

*'SPACE' *statement line*

'CONTINUE' *statement line*

Between segments

*'READ FROM' *statement line* (can also appear at the beginning or end of the program description preceding the program)

* *listing statement line*

*'LIBRARY' *statement line*

'TRACE' *statement line*

If 'CONTINUE' is included in the program description, the last segment of the Algol program must be followed by the

'FINISH' *statement line*

BRIEF DESCRIPTION

THE LISTING STATEMENT determines the nature of the listing - see Chapter 4.

THE 'TAB' STATEMENT determines the layout of the source program in the listing - see Chapter 4.

THE 'SEND TO' STATEMENT determines whether or not there will be any compiler output produced on the paper tape punch - see Chapter 5.

THE 'PROGRAM' AND 'SEGMENTS' STATEMENTS determine whether consolidation will take place - see Chapter 5.

THE 'COPY' STATEMENT determines whether the Algol library segments called by the program will be copied to the compiler output - see Chapter 5.

THE PERIPHERAL DESCRIPTION STATEMENTS select the channel numbers for peripheral devices and determine whether the standard peripheral facilities are required - see Chapter 3.

THE 'PRIORITY' STATEMENT selects the priority number of the program - see Chapter 8.

THE 'SPACE' STATEMENT indicates the amount of storage space required by the program - see Chapter 8.

THE 'CONTINUE' STATEMENT informs the compiler that the program consists of more than one segment - see Chapter 6 ('FINISH' must always be associated with the 'CONTINUE' statement).

THE 'READ FROM' STATEMENT specifies the input peripheral and type of input and allows segments to be input from different devices and in varying modes - see Chapter 6.

THE 'LIBRARY' STATEMENT allows the user to build up a library of subroutines other than those provided in the Algol library - see Chapter 6.

THE 'TRACE' STATEMENT specifies the extent of the error detection facilities required - see Chapter 7.

Chapter 3 Object program input/output

Although the Algol Report does not provide any facilities for the input and output of data and results for an Algol program, 1900 Algol provides a full input/output system. This system consists of a group of procedures defined in strict accordance with the rules of Algol and is described in the manual *Algol: Language*. Each peripheral used in the system is assigned a channel number in the program description by means of the 'INPUT' and 'OUTPUT' statement.

PERIPHERAL DESCRIPTION STATEMENTS

'INPUT' statement

FORMAT

'INPUT' *k* = *xxy*

where *k* is a list of one or more arbitrarily chosen integers, separated by commas, representing channel numbers.

xx is a two-letter code (TR for a paper tape reader; CR for a card reader) specifying the peripheral to be associated with *k*.

y is the peripheral unit number of *xx* in the range 0 to 15.

DESCIPTION

The input peripheral defined by *xxy* is assigned the channel number (or numbers) specified in the list *k*.

Example

'INPUT' 2, 4, 6 = TR2

Channel numbers 2, 4 and 6 will be associated with a paper tape reader which will be alloted as unit 2.

'OUTPUT' statement

FORMAT

'OUTPUT' *k* = *xxy*

where *k* is a list of one or more arbitrarily chosen integers, separated by commas, representing channel numbers.

xx is a two-letter code (LP for a line printer; TP for a paper tape punch; CP for a card punch) specifying the peripheral to be associated with *k*.

y is the peripheral unit number of *xx*, in the range 0 to 15.

DESCRIPTION

The output unit defined by *xxy* is assigned the channel number (or numbers) specified in the list *k*.

Example

'OUTPUT' 14 = LP0

Channel number 14 will be associated with a line printer which will be allotted as unit 0.

'OMIT IO' statement

FORMAT

'OMIT IO'

DESCRIPTION

If none of the standard peripheral devices (card or paper tape readers and punches or line printers) are used, this statement will prevent the compiler from supplying the normal peripheral facilities, thus saving core space. The statement can be used, for example, with a program which only uses a graph plotter, in which case the programmer will use the Graph Plotter Library routines.

Notes:

1 If the 'INPUT' and 'OUTPUT' statements are omitted, the compiler assumes the following:

input channel	*peripheral*	*unit*
0 and 1	paper tape reader	0
2	paper tape reader	1
3	card reader	0

output channel	*peripheral*	*unit*
0 and 2	line printer	0
1	paper tape punch	0
3	card punch	0
4	paper tape punch	1
5	line printer	1

2 The 'INPUT' and 'OUTPUT' statements can appear in any order but each device should be mentioned only once.

For example,

'OUTPUT' 0, 2 = LP0

should be punched and not

'OUTPUT' 0 = LP0

'OUTPUT' 2 = LP0

3 The peripheral description statements are not available with XASP and only the standard channel numbers given above may be used.

Chapter 4 Listing

During compilation, the statements of the source program are numbered by the compiler. If the compiler finds an error in a statement, it outputs an error message, which indicates the type of error and the statement concerned. The statement is identified in the error message by its statement number. The number associated with each line of the program can be obtained by certain program description statements and is called a *listing* ; the program description statements are called *listing statements* .

The compiler detects two kinds of errors - *syntactic* and *semantic* errors. Syntactic errors are found during the first stage of compilation when the syntax of each statement is checked. If the rules of the Algol Report have been broken, a syntactic error is indicated. If no syntactic errors are found, the second stage of compilation, in which semantic errors are detected, is entered automatically. A semantic error is indicated when a statement is logically incorrect, although it is correct. For example, a statement punched as:

'GTOO' *Lab3* ;

would produce a syntactic error because 'CTOO' is an illegal Algol symbol, whereas the statement:

'GOTO' *Lab3*;

where *Lab3* is undefined, would produce a semantic error.

LISTING

Listings can be obtained in three degrees of detail. The most detailed form is the *full listing*', which is available on a line printer only; an intermediate listing is available on either a line printer or a paper tape punch; the least detailed form, the *basic listing* , is available only on a paper tape punch.

Full Listing

The first line of a full listing contains the name of the compiler, the date of compilation and the time (if the processor has an internal clock) at which compilation commenced. The source program is then printed with the layout with which it was input (the 'TAB' statement can be used if the conventional PLAN tab settings are not required - see *'TAB' statement*, page 13). Lower case letters in the source program appear in the listing as capitals, overprinted with apostrophes. Figure 1, page 10, shows the first part of a full listing in which only upper case letters have been used in the source program and Figure 3, page 10, illustrates the use of lower case letters.

The statement number that is current at the beginning of each line is printed to the left of the line. The three exclamation marks are to separate the statement number from the line of program. No indication is given of how these numbers have been assigned within lines and so, at best, errors can be associated with one line of the source program. If the end of a line coincides with the end of a statement, the statement number is incremented before the next line is output. Therefore, if the next line is blank or starts with comment, it will be given the same number as the next statement. (Note: The program description is considered to be one statement).

If the program has been consolidated, the penultimate line of the listing gives the amount of core store that is required by the object program. The final line gives the name of the program and a two-digit character code indicating whether the compilation has been error free - EC signifies that no errors have been found and ZZ indicates that errors have been detected. The format of the last two lines of a full listing is the same as for an intermediate listing, as illustrated in Figure 2.

```
00000000          12/10/68          COMPILED BY XALP MK.  10
ST=     0 !!!                 'PROGRAM'(MAMMOTH76104)
ST=     0 !!!                 'OUTPUT'0=LPO
ST=     0 !!!                 'SPACE'300
ST=     0 !!!                 'PRIORITY'60
ST=     0 !!!
ST=     0 !!!     'BEGIN'
ST=     1 !!!     'COMMENT'  INTEGER TEST;
ST=     1 !!!     'PROCEDURE'WRC;
ST=     1 !!!       WRITETEXT('('''('C')'CALCULATED%RESULT=')');
ST=     1 !!!     'PROCEDURE'WRE(S);'STRING'S;
ST=     4 !!!     'BEGIN'  WRITETEXT('(''('C')'EXPECTED%%%RESULT= ')');
ST=     6 !!!         WRITETEXT(S);
ST=     7 !!!     'END';
ST=     7 !!!     'INTEGER' P,Q,R,S,T,U,V,W,X,Y,I;
ST=     8 !!!          P:= -5; Q:= 3; R:= 6; S:= 110;
ST=    13 !!!          T:= P ↑ Q * R '/' S + Q - R;
ST=    14 !!!          U := P -Q + S '/' R * Q ↑ R;
ST=    15 !!!          V:= (P * R - S '/'(-11)*Q) ↑ R;
ST=    16 !!!          W:= S ↑ (P ↑ ((Q - R) '/' P));
ST=    17 !!!          X:= S ↑ (S + P * Q * R + P * Q * 4 '/' 3);
ST=    18 !!!          Y:= -R ↑ (P * Q * R * S '/' P '/' Q '/' R '/' S) ↑ Q;
ST=    19 !!!       WRC;
ST=    20 !!!           PRINT (T, 5,0);
ST=    21 !!!           PRINT (U,5,0);
ST=    22 !!!           PRINT (V,5,0);
ST=    23 !!!           PRINT (W,5,0);
ST=    24 !!!           PRINT (X,5,0);
```

Figure 1 Full listing, no errors

```
11/25/01          19/05/68          COMPILED BY XALP MK.  10
     0,      0,      0,      0,      0,      1,      2,      4,      4,      6,
     7,      9,      9,     10,     10,     12,     14,     14,     15,     16,
    16,     19,     20,     20,     24,     28,     29,     30,     31,     33,
    34,     35,     36,     38,     43,     45,     50,     59,     63,     67,
    70,     74,     78,     82,     85,     89,     93,     97,    100,    104,
   108,    111,    114,    116,    118,    119,    122,    123,    127,    131,
   135,    139,    143,    147,    151,    155,    158,    160,    161,    162,
   163,    166,    167,    167,    170,    170,    171,    172,    175,    176,
   177,    178,    179,    180,    181,    182,    182,    183,    185,    186,
   191,    192,    192,    195,    196,    200,    201,    202,    202,    207,
   209,    210,    211,    212,    213,    217,    220,    223,    226,    229,
   230,    231,    234,    237,    240,    243,    246,    249,    249,    249,
CORE  8704
COMPILED    PETE          EC
```

Figure 2 Intermediate listing

```
10/57/34          27/06/68          COMPILED BY XALP MK. 10
ST=     0 !!!     'PROGRAM'(TEST201)
ST=     0 !!!     'BEGIN''PROCEDURE'P;P;
ST=     2 !!!           'INTEGER'I;'ARRAY'B[1:4];
ST=     4 !!!     I:=P;
ST=     6 !!!     'COMMENT'THAT SHOULD GIVE 201;
ST=     6 !!!     I:=4;
ST=     7 !!!     I:=I+B;
ST=     8 !!!     'END'THAT SHOULD GIVE 92;
STATEMENT      5 ERROR NUMBER    201
STATEMENT      7 ERROR NUMBER     92 CONCERNING    ]B
COMPILED   #TEST201        ZZ
```

Figure 3 Full listing, with semantic errors

4112(8.68)

With XALP, when a semi-compiled segment is read, a line of the form:

SEMI COMPILED: *name*

is printed, where *name* is the four-character name of the segment.

Paper throw will occur before each new source segment.

Intermediate listing

Figure 2 shows the complete intermediate listing of a program. The first line is the same as for a full listing; note that the time of compilation is given. The statement numbers that would be printed at the left hand side of a full listing are then printed in rows; the numbers are separated by commas. Each number refers to one line of a source program, blank lines and comments included. The last two lines are the same as for a full listing. In this case, 8,704 words of core will be used, the name of the program is PETE and no errors have been detected in the compilation.

The same conventions for semi-compiled segments (with XALP only) and paper throw described for a full listing are applied to an intermediate listing.

Basic listing

A basic listing consists of the first line and last two lines of the other forms of listing, i.e. the time and date of compilation, the name of the compiler, the amount of core store required, the name of the program or segment and EC or ZZ. No information is given about statement numbers and so a basic listing should be used only when a program has previously been compiled without errors.

LISTING STATEMENTS

Program description listing statements determine the type of peripheral that is to be used for listing, i.e. the *listing device*, and the type of listing that is to be obtained initially. The type of listing may be changed by intersegment statements but the peripheral chosen in the program description preceding the program must be used for the whole compilation.

'LIST' statement

FORMAT

'LIST' (*xx*)

or

'LIST' (NONE)

where *xx* is a two-character peripheral identifier (LP for a line printer; TP for a paper tape punch).

Line printer

If *xx* = LP, a full listing is obtained on the line printer. With XASP, a 96 print position line printer is always assumed. With XALP, a 120 print position line printer is assumed but this can be altered by changing the format of the 'LIST' statement to:

'LIST' (LP/*n*)

where *n* is the number of print positions required; for example, 'LIST' (LP/96) for a 96 print position line printer.

Paper tape punch

If *xx* = TP, an intermediate listing is obtained on paper tape. The listing will have a line length of 60 characters. It is possible, with XALP only, to alter the line length by changing the format of the 'LIST' statement to:

'LIST' (TP/*n*)

where *n* is the line length, in characters, that is required (maximum 120 characters). Figure 3, page 10, shows the printout of a listing produced by 'LIST' (TP); a 60 character line length gives ten statement numbers (equivalent to ten lines of source program) per line of listing.

'LIST' (NONE) statement

If no listing at all is required, the 'LIST' (NONE) statement should appear in the program description; NONE has the status of a peripheral and so a listing cannot be obtained by a following, intersegment statement.

If no listing is required initially but is required later in the program, the 'NO LIST' statement should be used (see *'NO LIST' statement*, page 13).

Note:

1 If no listing statement appears in the program description, 'LIST' (LP) is assumed by the compiler.

'SHORT LIST' statement

FORMAT

'SHORT LIST' (*xx*)

where *xx* is a two-character peripheral identifier (LP for a line printer; TP for a paper tape punch).

DESCRIPTION

Line printer

If *xx* = LP, an intermediate listing is obtained on the line printer. A 96 print position line printer is always assumed for XASP and a 120 print position line printer for XALP. If a different number of print position is required (with XALP only) the format of the 'SHORT LIST' statement can be altered to:

'SHORT LIST' (LP/*n*)

where *n* is the number of print positions required.

Paper tape punch

If *xx* = TP, a basic listing is obtained on paper tape. The listing will have a line length of 60 characters but this can be altered (with XALP only) by changing the format of the 'SHORT LIST' statement to:

'SHORT LIST' (TP/*n*)

where *n* is the line length, in characters, that is required.

'NO LIST' statement

FORMAT

'NO LIST' (*xx*)

where *xx* is a two-character peripheral identifier (LP for a line printer, TP for a paper tape punch).

DESCRIPTION

This statement is used when no listing is required initially but will be required later in the program. Intersegment listing statements (see below) can be used to obtain listings for subsequent segments. With XALP, the same conventions for specifying print positions and line lengths apply to the 'NO LIST' (*xx*) statement as are described for the 'LIST' and 'SHORT LIST' statements.

If no listing at all is required, the 'LIST' (NONE) statement should be used.

Intersegment listing statement

Intersegment listing statements may be used to alter the form of listing. The listing device may not be changed and so the peripheral identifier and (with XALP only) print position or line length specification should be omitted from intersegment statements. If a segment is not preceded by a listing statement, the type of listing used for the previous segment will be obtained.

'TAB' STATEMENT

FORMAT

'TAB' *n*

where *n* is an integer in the range 1 to 63.

DESCRIPTION

The statement causes the compiler to assume equally spaced tab settings, *n* characters apart, when printing the source program in a full listing.

Notes:

1 If this statement is omitted, the PLAN tab settings are assumed.

2 This statement is not available with XASP.

COMPILATION ERROR MESSAGES

The first stage of compilation is the reading of the source program and the numbering of the statements. At this stage the compiler checks the syntax of each statement. If a syntactic error is found, an error message is output to the listing device and is printed, on a new line, immediately after the listing of the statement concerned. An error message that indicates a syntactic error always takes the following form:

STATEMENT *n* SYNTAX ERROR *m*

where *n* is the number of the statement in which an error has been found and *m* is the *error number* associated with the syntactic error. The error numbers that may be encountered during compilation and the error conditions associated with the number, are given in Appendix 1. Syntactic

error numbers start at 300.

If a syntactic error is found, the rest of the program will be read and any further errors are detected but compilation does not procede further. ZZ will appear in the final line of the listing and the message HALTED:-ZZ will appear on the console typewriter.

If no syntactic errors are found, the second stage of compilation is entered automatically; the semi-compiled form of the program is produced and semantic errors are detected. At this stage, all but the last two lines of the listing will have been produced and so any error messages indicating semantic errors will be printed together at the end of the listing. Multi-segment programs are semi-compiled segment by segment and so semantic error messages will follow the listing of the segment to which they apply. Error messages that indicate semantic errors take one of the following forms.

STATEMENT *n* ERROR NUMBER *m*

STATEMENT *n* ERROR NUMBER *m* CONCERNING *identifier*

where *n* and *m* have meanings as before; the value of *m* will be in the range 1 to 299. With certain error conditions it is possible to associate the error with an identifier within a statement and the second type of message above is used in such cases; *identifier* is the particular identifier concerned.

Figure 3, page 10, shows a full listing with one of each type of semantic error message. The identifier concerned in error 92 is B; the right hand bracket indicates that B was upper case in a mixed case program. No indication of lower case is given or of upper case in a single case program.

As soon as the first semantic error is found, semi-compiled output ceases but compilation continues in order that any further semantic errors can be found. If there are no errors, the program will be consolidated unless the 'SEGMENTS' statement appears in the program description. Compilation will be abandoned when 40 semantic errors have been found. Under certain conditions, one error may cause the compiler to misinterpret several of the following statements. Therefore, if some error messages seem meaningless, the program should be recompiled with as many errors as possible corrected.

Note:

A common cause of error is the omission of the semicolon after 'END'. Because of the syntax of the Algol language this condition cannot be detected by the compiler and so programs should be checked both before and after punching to see that all 'END' symbols are followed by semicolons. If this error is present in a source program, the compiler will treat the part of the program between 'END' and the next semicolon as comment and the statement number will not be incremented until the semicolon has been passed.

Chapter 5 Compiler output

After reading the Algol source program, XALP and XASP produce one or more segments of semi-compiled programs. If the compilation is error free and consolidation is requested, or it if is required to force consolidation, the Algol semi-compiled library is read from paper tape and the request slip, General Purpose Loader and consolidated leader information are output.

The semi-compiled program, request slip, General Purpose Loader and consolidated leader information can be output on paper tape only (the relevant Algol library segments will only be copied to the output if 'COPY' has been specified in the program description - see *'COPY' STATEMENT*, page 16).

'SEND TO' STATEMENT

FORMAT

'SEND TO' (TP)

or

'SEND TO' (NONE)

DESCRIPTION

'SEND TO' (TP)	Compiler output is to be directed to the paper tape punch.
'SEND TO' (NONE)	All compiler output is to be suppressed.

Notes:

1 The 'SEND TO' (NONE) statement is usually included if, at an early stage of program testing, the compilation is being used only to find compilation errors. If this statement is included, the 'SEGMENTS' statement will be assumed irrespective of whether or not it is specified later in the program description.

2 If the 'SEND TO' statement line is omitted, 'SEND TO' (TP) will be assumed.

'PROGRAM' AND 'SEGMENTS' STATEMENTS

'PROGRAM' statement

FORMAT

'PROGRAM' (*progname*)

where *progname* is the program name, consisting of from four to twelve alphabetic (upper case) and numeric characters, with the first character alphabetic. The first four characters of *progname* are used to identify the object program and any further characters are available for use as an accounting code. This code is included in the final console and listing messages.

DESCRIPTION

The program is compiled and if the compilation is error free, consolidated.

Note: The character combination SIGN is not permitted as the object program name.

'SEGMENTS' statement

FORMAT

SEGMENTS' (*segname*)

where *segname* is the segment name, consisting of from four to twelve alphabetic (upper case) and numeric characters, with the first character alphabetic. The first four characters of *segname* serve to distinguish the compilation from others in the series but do not identify the object program.

DESCRIPTION

This statement is an alternative to 'PROGRAM' and if it appears in the program description, the following segment (or segments) is compiled but not consolidated. Note: The formats of procedure segments are described in Chapter 9, page 29.

Note:

If no 'PROGRAM' or 'SEGMENTS' statement appears in the program description, 'PROGRAM' (AXXX) will be assumed.

'COPY' STATEMENT

FORMAT

'COPY'

DESCRIPTION

This statement, which is only valid after 'PROGRAM' (as opposed to 'SEGMENTS'), causes the semi-compiled library segments called by the Algol program to be output on paper tape after the semi-compiled program segments.

The library segments are copied without their associated segment leaders (information output at the end of a semi-compiled segment) since these leaders are only required for the consolidation phase and not for loading the program into store.

Note:

If this statement is omitted, the relevant library segments are not copied and the library must be read again when the program is being loaded. The library can be split into two parts with the semi-compiled segments in one part and their associated leaders in another. Then, if 'COPY' is not used, only the leaders need to be read at compilation time and the semi-compiled segments are read when the program is being loaded. The program (XMUL) for splitting the library is described in the *Algol: Compiler Library Procedures* manual.

Chapter 6 Compiler input

An Algol program, or segments of a program, can be input to XALP and XASP from paper tape or cards, in source or semi-compiled form. Source programs must be punched in accordance with the 1900 hardware representation of the Algol basic symbols. Programs held on paper tape can be punched in either normal mode which utilizes the 1900 Series 64-character card code, or in full mode, which allows the use of lower case letters. A table describing the 1900 Algol hardware representation is given in Appendix 4.

Compilation is initiated by entering the compilers at one of three entry points corresponding to the following initial conditions.

1 Compilation is commenced by reading source program from cards.

2 Compilation is commenced by reading source program from paper tape, punched in normal mode.

3 Compilation is commenced by reading source program from paper tape, punched in full mode.

In each of these three cases, the program can be optionally preceded by a program description.

Segmented programs compiled by XALP can have segments input from varying devices and in varying forms. The 'READ FROM' statement is used to switch input devices and types of input between segments and, when incorporated in the program description preceding the program, it can be used to alter the initial conditions corresponding to the three compiler entry points.

With XASP, however, the 'READ FROM' facility is not available and therefore no switching is allowed between input media and modes of input. Semi-compiled segments can be read only from paper tape when the Algol library is being read, after the source segments have been compiled.

PROCEDURE SEGMENTS

An Algol program written in segmented form consists of one *master* segment and one or more *procedure* segments, written and declared in the same way as any other Algol procedure. Procedure segments can be called in the master segment or in another segment and can be written in Algol, FORTRAN or PLAN. FORTRAN and PLAN segments are always held in semi-compiled form for incorporation in Algol programs. It is also advisable to keep fully-tested Algol procedure segments in semi-compiled form because semi-compiled program requires less machine time to be input than source program. The method of declaring and writing procedure segments is described in Chapter 9.

It is useful to employ segmentation when writing a large program because fully tested procedures can be held as semi-compiled procedure segments thus avoiding recompiling the procedure every time a change is made to another part of the program.

A procedure that is required in more than one program can be translated into a semi-compiled segment once and then incorporated into each program that requires it. Such procedure segments can be grouped to form a *library* of subroutines. An Algol library is issued with all 1900 Algol compilers consisting of standard mathematical functions (*sin*, *exp*, etc), input/output procedures (*read*,*print* , etc) and other commonly used routines. Some of the procedures in the Algol library are known to the compiler and need not be declared, while others have to be explicitly declared as **external** or **Algol** procedures (see *PROCEDURE SEGMENTS*, page 59). The Algol library issued with XALP and XASP is supplied on paper tape and contains the SRA1 group of subroutines as described in the manual *Algol: Compiler Library Procedures.*

The user may insert additional procedures in the Algol library or, with XALP, set up a private library by means of the 'LIBRARY' statement (with XASP, private libraries must be read in with the Algol library). If a user's private procedure inserted into the Algol library contains a call to

any other procedure already in the library, the new procedure must be inserted before any procedure it calls. The safest place to insert such a procedure is at the front of the Algol library.

MULTI-REEL PROGRAMS

If a program is held on several reels of paper tape, each reel must start with at least one foot of blank tape and all but the last tape must end with:

newline

TC4 (see Appendix 3)

newline

followed by at least a foot of blank tape. The last (or only) tape must end with:

newline

****

newline

TC4

newline

followed by at least a foot of blank tape.

'READ FROM' STATEMENT

FORMAT

'READ FROM' (*xx*/*yy*)

where *xx* is a peripheral device (TR for paper tape reader, CR for a card reader).

yy describes the type of input.

DESCRIPTION

Statement	*Instruction to compiler*
'READ FROM' (CR)	Next, read source program from cards
'READ FROM' (CR/SC)	Next, read semi-compiled program from cards.
'READ FROM' (TR/NM)	Next, read source program from paper tape, punched in normal mode hardware representation.
'READ FROM' (TR/FM)	Next, read source program from paper tape, punched in full mode hardware representation.
'READ FROM' (TR/SC)	Next, read semi-compiled program from paper tape.
'READ FROM' (TR)	Next, read program, of the same type as the previous segment, from paper tape. Card source has the implied type NM.

Notes:

1. When switching between peripherals and type of input, the input device and type of input remain set until another 'READ FROM' statement is encountered.
2. A change in input device causes the previously used peripheral to be released by the compiler; a change in type of input causes the peripheral to be disengaged.

3 The 'READ FROM' statement may appear between segments or at the beginning or end of the program description preceding the program.

4 This statement is not available with XASP.

5 A 'READ FROM' statement is required when switching from semi-compiled to source program (or vice versa) even when the input medium is not being changed.

'CONTINUE' AND 'FINISH' STATEMENTS

'Continue' Statement

FORMAT

'CONTINUE'

DESCRIPTION

This statement must appear in the program description when the program consists of more than one segment (excluding segments called from the Algol library). The compiler will compile all segments until it encounters the 'FINISH' statement.

Note: This statement must appear if the compiler is required to read past the end of the source program, for example to read a 'READ FROM' statement directing it to a special set of library routines. If the 'CONTINUE' statement is omitted, the compiler will not read past the first source segment it encounters.

'Finish' Statement

FORMAT

'FINISH'

This statement, which is used only in association with the 'CONTINUE' statement, indicates that:

1 if the 'PROGRAM' statement has been included (or implied) in the program description, the Algol library tape must be read and the program consolidated;

2 if the 'SEGMENTS' statement has been encountered, compilation is to stop.

Note: The 'CONTINUE' and 'FINISH' statements are not available with XASP.

'LIBRARY' STATEMENT

FORMAT

'LIBRARY'

Description

If the 'PROGRAM' statement has been included (or implied) in the program description, the segments input after the 'LIBRARY' statement will be consolidated into the program only if they have

been called for. In this way, the user can set up private libraries of semi-compiled segments.

Notes:

1 Private libraries made up of source segments are not recommended since each segment will be compiled during every compilation, although only those requested will be consolidated into the program.

2 If 'SEGMENTS' has appeared in the program description, the 'LIBRARY' statement has no effect.

3 This statement is not available with XASP and any private semi-compiled library routines must be read in with the Algol library.

Chapter 7 Program testing

Syntactic and semantic errors in Algol programs are detected by the compiler and appropriate error messages are output on the listing device during compilation (see Chapter 4). Error detection facilities are also available during the execution of an object program and if an error is detected at this stage, the run is abandoned after all output buffers have been emptied and an error message has been output on the currently selected output peripheral. If no output device has been selected or if the 'OMIT IO' statement appears in the program description, the error message is output on the console typewriter.

Execution error messages take the form:

ERROR TYPE *n*

where *n* is the error number that corresponds to the error condition that has occurred. The execution error numbers and their explanations are described in Appendix 2.

There are three levels of error detection, called *trace levels*, available with XALP and XASP. At the lowest level, level 0, only the basic error checks applied to all Algol programs are carried out. Level 1 includes three additional checks. Level 2 extends one of the checks made at level 1 and includes facilities for following the course of the part of the program that led up to an error or of any part of the program chosen by the programmer. Each level contains all the facilities of any lower level, as well as any additional facilities.

The trace level to be applied to each segment of an object program is determined by the 'TRACE' statement.

'TRACE' STATEMENT

Format

'TRACE' *n*

where = 0, 1 or 2.

Description

n is taken as the trace level to be applied to the following segment. If a segment is not preceded by a 'TRACE' statement, the trace level of the preceding segment continues to apply; level 0 is assumed until the first 'TRACE' statement is encountered.

Notes:

1 The statements 'TRACE' 1 and 'TRACE' 2 cause the compiler to insert extra code into an object program, in order to provide the facilities that are not available at level 0. Although these facilities may be useful for test runs of a program, the extra code slows down execution even if there are no errors in the program. Fully tested and corrected programs should therefore be recompiled with all 'TRACE' statements removed before production runs are made. This practice is particularly advisable when a program is to be used several times.

2 'TRACE' is an intersegment statement and should be the last statement (excepting an optional 'READ FROM' statement) that is read before the start of the Algol segment. If the statement is to appear before the first segment of the program, the program description is considered as a separate segment and 'TRACE' will then be an intersegment statement between the program description and the first segment of the program.

TRACE LEVEL 0

No extra code is incorporated at level 0 and only basic error checks are carried out. Level 0 is assumed if no 'TRACE' statements appear in the program description and so the statement 'TRACE' 0 is used only to revert to level 0 when previous segments have been compiled at level 1 or 2.

The following errors are detected at level 0.

Error number	*Reason*
10 to 29	Arithmetical errors.
30 to 39	Input/output system errors.
40 to 49	Magnetic tape backing store package errors.
50 to 59	Errors peculiar to Algol.
70 to 79	Additional input/output system errors.

Full details of the error conditions that correspond to these error numbers are given in Appendix 2.

TRACE LEVEL 1

In addition to the error checks performed at level 0, three more tests are carried out at level 1.

The three additional error checks have the following error numbers.

Error number	*Reason*
60	An array element subscript is too small.
61	An array element subscript is too large.
62	Overflow has occurred.

Error Numbers 60 and 61

The address of each array element is checked to see whether or not the element is within the area of core store allowed for the array. The bounds of each dimension are not checked individually.

Error Number 62

The state of the overflow register is examined on entry to all standard functions. If overflow has occurred, i.e. the overflow register is set, the error message indicating error number 62 is output. This error check can be inhibited by the setting of system switch 9 to on, either by an operator message or by use of the external procedure *on* (see page 25). Overflow can then be detected, if necessary, by means of the external procedure *overflow* (see page 25). When switch 9 is on, the the overflow indicator is left set and the program does not halt. However, if some other error is detected or if the run is successfully completed, the overflow register is checked automatically and error 62 is indicated if the register is set, even if system switch 9 is on.

TRACE LEVEL 2

In addition to the error checks performed at levels 0 and 1, the check for overflow (error number 62) is carried out on entry to functions and procedures written by the user, as well as for standard functions.

At level 2 there is a facility that enables parts of the execution of the program to be followed. If the program halts because of an error, a *trace list* is output on the currently selected output peripheral. A trace list is a list of the last 60 *monitor points* passed in the execution of the program. A monitor point is:

1 a label,

2 the **begin** of a block,

3 the **end** of a block,

4 a library procedure or a procedure written by the user.

The first line of the trace list is the heading TRACE OF # *name*, where *name* is the program name or, if only a procedure segment has been compiled at level 2, the first four characters of the procedure identifier. Each of the next 60 lines represents a monitor point in one of the following six ways.

1 *label* AT ST. *n*
where *label* is the first five characters of a label and *n* will be the statement number associated with the label.

2 **BEG AT ST. *n*
This line indicates the **begin** of a block, where *n* is the statement number of the **begin**.

3 **END AT ST. *n*
where *n* is the statement number of the **end**.

4 * *identifier*
where *identifier* is the identifier of an implicitly declared procedure.

5 *xxxxx* AT ST. *n*
where *xxxxx* is the first five characters of the identifier of a procedure declared by the programmer and *n* is the statement number of the call of the procedure.

6 *string*
where *string* is the five characters of the string presented to the procedure *mpname* (see page 24).

No indication is given in trace lists of whether upper or lower case characters were used for labels, strings and identifiers in the source program.

If the same monitor point is repeated or if two or more identical monitor points are adjacent, this is indicated by only two lines of the list. The first line is the normal representation of the monitor point and the second line is

EXECUTED *n* TIMES

where *n* is the number of consecutive occurrences of the monitor point.

Example

The following statement contains three monitor points, each of which is passed 30 times.

```
for I = 1 step 1 until 30 do
A := sqrt (B) + sqrt (C) + sqrt (D);
```

The two lines of the trace list would be:

```
*SQRT
        EXECUTED 90 TIMES
```

After the heading and subsequent 60 lines, the list is terminated by the line END OF TRACE.

At level 2, a trace list is output automatically when the program is abandoned by the detection of an error. A trace list is also output when the program is halted by the procedure *pause* (see page 24) and when the program is successfully completed. *mplist* may also be used to produce a trace list without halting the execution of the program. If a program, compiled at level 2, loops, the operator message GO #*name* 28, where *name* is the name of the program, should be used to obtain a trace list.

Note: If no output peripheral is available, the trace list is not output on the console typewriter. Trace level 2 should therefore not be used for a program in which the program description statement 'OMIT IO' appears or in which no output peripheral is available at some point.

PROGRAM TESTING PROCEDURES

There are two procedures intended for use only at trace level 2. The procedure *mplist*, when called, causes a trace list to be output and the procedure *mpname* may be used to insert a character string into a trace list to identify the section of the program. At level 2, the procedure

also causes a trace list to be output.

The procedure *overflow* may be used, at any trace level, to detect overflow either in conjunction with, or in preference to, error check 62, which can be suppressed.

The procedure *pause* is implicitly declared; the other procedures must be declared as **external**.

The mplist procedure

procedure *mplist* (*n,m*); **value** *n,m*, ; **integer** *n,m*; **external**;

A call to this procedure causes a trace list to be output; the execution of the program is not halted. The program should have been compiled at trace level 2 and system switch *n*+ 10 should be on when this procedure is called or the procedure will have no effect; *n* may be 1, 2 or 3. The value of *m* should be in the range 0 to 99; when the list is printed it will be preceded by the line

AT MONITOR POINT *m*

The mpname procedure

procedure *mpname* (*string*); **string** *string* ; **external**;

A call to this procedure is a monitor print. If a call is within a section of the program covered by a trace list, the first five characters of the actual string supplied in the call will appear in the list and if the string is unique it will identify the section of the program. If compilation was at level 0 or 1 this procedure has not effect.

The pause procedure

procedure *pause* (*n*); **value** *n*; **integer** *n*;

At all trace levels a call to this procedure causes the execution of the program to halt and the console message # *name* HALTED:- *n* to be output where *name* is the name of the program. The program can be restarted by an operator GO message but if the run is to be abandonded at this point, the call of *pause* should be preceded by a call of the *runout* (see the manual *Algol: Compiler Library Procedures*) procedure for each output channel, so that all output buffers are emptied; alternatively, the same effect may be obtained by the operator message GO # *name* 28, where *name* is the name of the program.

If compilation was at level 2, a trace list will be output if the program is restarted by the operator, provided that system switch 10 is on. The list is preceded by the line

AT MONITOR POINT *n*

where *n* is in the range 0 to 99.

The overflow procedure

Boolean procedure *overflow* ; **external**;

A call to this procedure causes the overflow register to be checked. If the register is set, the procedure takes the value **true**; if not, the procedure takes the value **false**. The overflow register is left clear after the call.

The detection of overflow in this way does not halt the execution of the program and can be used in preference to error check 62 when the execution of the program is to be completed for testing purpose. (Note: Error check 62 is available at trace levels 1 and 2 but may be suppressed by setting system switch 9 to on.)

SYSTEM SWITCHES

The system switches are bits of a reserved word of core store (word 30 of each program). A switch is *on* when the bit is set to 1 and *off* when the bit is set to 0. At the start of a run all switches will be off but can be set on by the operator message

ON # *name n*

where *name* is the name of the program and *n* is the number of the switch. Switches can be set on and off during the execution of a program by calls to the procedures *on* and *off*. The procedure *test* indicates whether a particular switch is on or off and the procedure *monoutput* outputs certain information if a specified switch is on. These four procedures can operate on switches 0 to 23 and their specifications are given later in this section.

Switches 1 to 8 are available for the user's own purposes. The other switches are restricted for use by standard library routines and should not be employed by the user as they may be altered by a library routine being used by the program.

Only switches 9 to 13 are relevant to the facilities described in this chapter.

Switch 9	Error check 62, which detects overflow, is suppressed when this switch is on.
Switch 10	If this switch is on, a trace list will be output when a program compiled at level 2 is restarted by an operator GO message after execution has been halted by a call to the *pause* procedure.
Switches 11, 12 and 13	One of these switches will be associated with each call to the procedure *mplist* and that switch should be on when the call occurs if a trace list is to be produced. Compilation must have been at level 2.

Note: If switch 23 is set, paging will be inhibited, i.e. the line printer will not throw the paper to a head of form when the end of a page of printout is reached and printing will continue over the printout perforations.

The on procedure

procedure *on* (*n*); **value** *n*; **integer** *n*; **external;**

A call to this procedure sets system switch *n* to on. The value of *n* can be in the range 0 to 23.

The off procedure

procedure *off* (*n*); **value** *n*; **integer** *n*; **external;**

A call to this procedure switches off system switch *n*. The value of *n* can be in the range 0 to 23.

The test procedure

Boolean procedure *test* (*n*); **value** *n*; **integer** *n*; **external;**

This procedure takes the value **true** if system switch *n* is on and **false** if it is off. The value of *n* can be in the range 0 to 23.

The monoutput procedure

procedure *monoutput* (*n,t,x*); **value** *n,x*; **integer** *n*; **string** *t*; **real** *x*; **external;**

If system switch *n* is on, the string of characters *t* and the number *x* will be output on the currently selected output device. *x* is output in a standard floating point format and the layout editing characters that can be used in *t* are the same as those described for the *write text* procedure in the manual *Algol: Compiler Library Procedures*

Chapter 8 Miscellaneous statements

'PRIORITY' STATEMENT

Format

'PRIORITY' *p*

where *p* is an integer in the range 1 to 99.

Description

The number *p* is assigned to the object program as the priority number. This number is used by Executive when multiprogramming, with precedence in the order of execution of a group of programs being given to the program with the highest priority number.

Notes:

1 This statement can only appear if a 'PROGRAM' statement has been included (or implied) in the program description.

2 If the statement is omitted, 'PRIORITY' 50 will be assumed.

3 This statement is not available with XASP.

'SPACE' STATEMENT

Format

'SPACE' *s*

where *s* is an integer.

Description

The number *s* is taken as the number of words of variable (or stack) storage required for the compiled program or segments.

Notes:

1 The following guide indicates approximately the amount of work space required by the object program.

(a) Each **real** variable occupies two words.

(b) Each **integer** variable occupies one word.

(c) Each **Boolean** variable occupies one word.

(d) Each *n* dimensional **array** with a total number of elements *m* occupies

$(n+2) + 2m$ words for **real** arrays

$(n+2) + m$ words for integer and **Boolean** arrays

(e) Each **begin** and each entry to a procedure requires eight words.

(f) Storage space is only used for blocks which have been entered (i.e. if an exit has been made from block *A* before block *B* has been entered, then *A* and *B* can use the same storage area).

(g) **own** variables follow the same rules as above except that once a block containing an **own** declaration has been entered, the storage space used for the **own** variables will be permanent, even when exit is made from the block.

(h) It is a reasonable safeguard to allow one word per statement for extra storage used in the evaluation of expressions.

2 If the statement is omitted, 'SPACE' 3000 will be assumed. This will be sufficient for most Algol programs. If more space than was allowed for at compilation time is required during the run, the object program will attempt to obtain more store from Executive and, if successful, will then continue the run. It is therefore not vital that the space required should be specified but if an accurate specification is given, Executive can use the information to ease scheduling problems when it is multiprogramming.

3 This statement is not available with XASP.

Chapter 9 Mixed language programming

XALP and XASP can compile an Algol program that has been divided into segments. The segments can be written in Algol, FORTRAN or PLAN. Algol segments can be input to the compiler in source or semi-compiled form but FORTRAN and PLAN segments can be input only in semi-compiled form. In this way, the programmer can write a segment in the most convenient language available.

This chapter describes the format of Algol segments and contains information on how FORTRAN and PLAN segments can be incorporated into Algol programs. A semi-compiled segment produced by an Algol compiler can be inserted into a program written in another language, provided that the program into which it is consolidated is not overlaid, does not automatically produce a binary dump (e.g. by a DUMP ON statement in FORTRAN) and is written in compact mode. For full details of how Algol segments can be incorporated into such programs, the appropriate compiler manuals should be consulted.

A segmented Algol program consists of one *master segment* (the program between the first begin and its associated end) and one or more *procedure segments*, written in the form of procedures.

PROCEDURE SEGMENTS

Declaration

A procedure segment is declared at the head of a block in which it is called. For procedures written in Algol, the procedure body part of the declaration must consist of only the basic symbol **algol** and for procedures written in FORTRAN or PLAN the procedure body consists of only the basic symbol **external**.

Communication of data

Communication of data between segments written in different languages is achieved by transferring parameters from one segment to another since Algol does not use named common areas. A standard linkage is used for transmitting the parameters and this is fully described in the section 'PLAN procedure segments'. All parameter types can be used with PLAN procedure segments but there are restrictions on the types that can be used with FORTRAN segments. These restrictions are described in the section *FORTRAN PROCEDURE SEGMENTS*, page 30.

Notes:

1 The section *PROCEDURE SEGMENTS USING UPPER COMMON VARIABLES*, page 41 describes precautions that must be taken when using upper common variables in external procedures.

2 When an expression is presented as a parameter to an **external** procedure, it must always be called by value.

Procedure libraries

Segments that are required in more than one program can be grouped to form a *library* of semi-compiled procedure segments. A standard Algol library is issued, on paper tape, with XALP and XASP. This library is the SRA1 subroutine block which is described in the manual *Algol: Compiler Library Procedures*. Some of the procedures, such as the standard functions and input/

output procedures, are known to the compiler and need not be declared but the other procedures must be explicitly declared as **external** or **algol** procedures. The user can set up a private library of subroutines or add extra semi-compiled segments to the standard library. Users' procedures must always be explicitly declared.

ALGOL PROCEDURE SEGMENTS

Procedure segments written in Algol can be input to XALP and XASP in source or semi-compiled form. A semi-compiled segment requires less machine time to be input to a compiler than a source segment and it is therefore more efficient to hold a fully tested segment in semi-compiled form. If a semi-compiled segment is to be compiled with other source segments, it will not have to be recompiled each time amendments are made to other parts of the program. A segment required in more than one program can be translated once into semi-compiled form and incorporated into any program that requires it.

Declaration

A procedure segment can be called from the master segment or from another procedure segment and must be declared at the head of a block in which it is called, in the same way as any other Algol procedure, except that the procedure body part of the declaration must consist of only the basic symbol algol. The hardware representation of **Algol** is the same as for any other Algol basic symbol.

The procedure identifier can have a maximum length of 31 alphanumeric characters, with the first character alphabetic, and it must be written in the appropriate 1900 Algol hardware representation of lower case letters (see Appendix 4, page 59).

Example

real procedure *example1* (*a*, *b*); value *a*; integer ; label *b*; algol;
comment the procedure *example1* is to be used in this block and is
supplied as a separate segment;

Procedure

Algol procedure segments must begin with the basic symbol procedure (optionally preceded by **real**, **integer** or **Boolean**) and must end with a semi-colon.

When the procedure segment is input to an Algol paper tape compiler, a separate semi-compiled segment will be output during compilation.

Note: The procedure segment must be written outside the master segment block and therefore global variables cannot be used in the procedure.

Example

The procedure declared in the previous example could be written in the following form.

real procedure *example1* (*a*;*b*); **value** *a*; **integer** *a*; **label** *b*;
. . . .
. . . . . . procedure body . . .
. . . .;

FORTRAN PROCEDURE SEGMENTS

Procedure segments can be written in FORTRAN and input to an Algol compiler as a semi-compiled segment. Intersegment communication is parametric since Algol does not use named common areas. A standard linkage is used to pass parameters between segments (see *PLAN PROCEDURE SEGMENTS*, page 31) but only the following types can be used since some Algol parameter types

have no corresponding type in FORTRAN.

Algol	FORTRAN
real	REAL
integer	INTEGER
Boolean	LOGICAL
array	REAL array
integer array	INTEGER array
Boolean array	LOGICAL array
procedure (external type)	EXTERNAL procedure

Algol integer variables and array elements are stored in one word. FORTRAN segments should therefore be compiled in COMPRESS INTEGER AND LOGICAL mode so that INTEGER storage will correspond.

Declaration

The procedure body part of the declaration consists of only the basic symbol **external**, which has the same hardware representation as any other Algol basic symbol. The procedure segment name can consist of up to 31 alphanumeric characters with the first character alphabetic, as allowed for in 1900 FORTRAN . The procedure name in the declaration must be written in the appropriate 1900 Algol hardware representation for lower case letters (see Appendix 4, page 59) although upper case letters only will be used in the FORTRAN segment.

Example

procedure *exfort* (*X*,*Y* ,*I*); **value** *X* ; **real** *X* ,*Y* ; **integer** *I* ; **external;**

Note: Procedures of type Algol cannot be used as parameters.

Calling by name

Parameters must be called by name if they are used on the left-hand side of an assignment statement in the FORTRAN procedure or if they are of type **procedure**. Otherwise, they must be called by value. (Note: When an expression is presented as a parameter to an **external** procedure, it must always be called by value.)

Procedure

A FORTRAN procedure segment is written as a FORTRAN FUNCTION or SUBROUTINE segment.

Example

```
SUBROUTINE EXFORT (X, Y, I)
REAL X, Y
.
.
.
END
```

Note: FORTRAN arrays always start with the subscript 1 (for example, A [1] or B[1, 1, 1]) whereas Algol arrays can start from any subscript (for example, *A* [-5:5] has *A* [-5] as the first element). An Algol array which is to be passed as a parameter to a FORTRAN program must, therefore, always start with the first element having a unit subscript (i.e. the array must be declared as, for example, ***array*** *Y* [1:*n*, 1:*m*] where *n* and *m* can be any positive numbers).

PLAN PROCEDURE SEGMENTS

Procedure segments can be written in PLAN and input to an Algol compiler in semi-compiled form. The only restriction on parameter types are that procedures of type **Algol**, cannot be used as an actual parameter. Otherwise, all parameter types can be used and can be called either by value or by name except if the parameter is an expression, in which case it must be called by value (i.e. not by a variable name).

A procedure call in the Algol source segment will cause the compiler to generate a set of object program instructions known as the *standard calling sequence*. This sequence will include a CALL (070) instruction to the procedure which will place the link address in X1 after the CALL instruction has been obeyed. The compiler generates one instruction per parameter in the order in which the parameters appear in the declaration. Each of these instructions, when obeyed from the body of the procedure, loads the address of the parameter into an accumulator. The instruction may directly load the address into an accumulator or it may be a branch instruction but the net effect of obeying this branch instruction will still be to load the address of the parameter into an accumulator. (Note: The address of a label cannot be obtained by this method.)

If a procedure is a function, the results are placed in a standard location. If the result is of type real, it is placed in the floating point accumulator A and if the result is of type integer or Boolean it is placed in X6.

Declaration

The procedure body part of the declaration consists of only the basic symbol **external**, which has the same hardware representation as for any other Algol basic symbol. The procedure name can contain up to eleven alphanumeric characters with the first character alphabetic. The procedure name in the declaration must be written in the appropriate 1900 Algol hardware representation for lower case letters (see Appendix 4, page 59) although only upper case letters will be used in the PLAN procedure.

Example

procedure *planproc* (*a,b*); **real** *a* ; **integer** *b* ; **external**;

Procedure

SEGMENT NAME

The PLAN procedure segment begins with a directive of the form

#PROGRAM / *segname*

where *segname* is the name of the procedure segment.

PICKING UP PARAMETERS

The address of a parameter is loaded into an accumulator when the relevant instruction in the standard calling sequence is obeyed. If the parameter is called by value, the instruction is obeyed once only and the value of the parameter is stored by the PLAN segment. If the parameter is called by name, however, the instruction must be obeyed each time a new value is required and the programmer must preserve the contents of the accumulators because the Algol program that provides a new value of the parameter may use all the accumulators (Note: X1 will be preserved). Details of picking up each type of parameter are given in the following sections (Note: When an expression is presented as a parameter to an **external** procedure, it must always be called by value.)

Integer

When the instruction in the standard calling sequence has been obeyed, X3 will contain the address of the 24-bit word containing the integer.

Example 1

The sixth parameter of type integer of a procedure called *pauline3* is to be picked up. The relevant part of the PLAN segment will be written as follows.

LABEL 1–6	OPERATION 7–12	ACC. 13–15	OPERAND 16–72	PROG. IDENT. 73–75
#PROGR	AM		/PAULINE3	
#LOWER			LINK, INT6	
#PROGR	AM			
	STO	1	LINK [STORE LINK ADDRESS	
	OBEY		5(1) [SET X3=ADDRESS OF 6TH PARAMETER	
	LDX	4	(3) [PICK UP VALUES OF PARAMETER	
	STO	4	INT6 [STORE VALUE IN INT6	

If *pauline 3* has seven parameters and the return to the Algol program is not made via a switch or a label, the return will be made as follows

LABEL	OPERATION	ACC.	OPERAND	PROG. IDENT.
	LDX	1	LINK [RESTORE LINK ADDRESS	
	EXIT	1	7 [RETURN TO ALGOL PROGRAM	

Example 2

integer procedure *even* (*i*); **value** *i* ; **integer** *i* ; **external;**

comment this procedure produces the result : *even* := (-1) *i* ;

This procedure, if written in PLAN, could take the following form.

LABEL 1–6	OPERATION 7–12	ACC. 13–15	OPERAND 16–72	PROG. IDENT. 73–75
#PROGR	AM		/EVEN	
#PROGR	AM			
	OBEY		(1) [SET X3=ADDRESS OF I	
	LDX	6	(3) [SET X6=VALUE OF I	
	ANDN	6	1 [X6=0 IF I EVEN, 1 IF ODD	
	ERN	6	1 [X6=1 IF I EVEN, 0 IF ODD	
	BNZ	6	*+2 [RESULT CORRECT IF EVEN	
	NGN	6	1 [RESULT FOR ODD	
	EXIT	1	1 [RETURN TO ALGOL PROGRAM	
#END				

Real

When the relevant instruction in the standard calling sequence has been obeyed, X3 will contain the address of the first of two adjacent 24-bit words containing the real parameter as a floating point number.

Example

The third parameter of type real of a procedure called *claudine* is to be picked up. The relevant part of the PLAN segment will be written as follows.

LABEL (1–6)	OPERATION (7–12)	ACC. (13–15)	OPERAND (16–72)	PROG. IDENT. (73–75)
#PROGRAM			/CLAUDINE	
#LOWER			LINK	
#PROGRAM				
	STO	1	LINK [STORE LINK ADDRESS	
	OBEY		2(1) [SET X3=ADDRESS OF 3RD PARAMETER	
	LFP		(3) [SET FLOATING POINT ACCUMULATOR =	
			[VALUE OF 3RD PARAMETER	

Boolean

When the relevant instruction in the standard calling sequence has been obeyed, X3 will contain the address of the 24-bit word containing the Boolean quantity. Bit 23 = 0 represents false and bit 23 = 1 represents true.

Example

procedure *eve*(*i*, *even*); **value** *i* ; **integer** *i* ; **Boolean** *even* ; **external**;

comment this procedure produces the result *even* : = *true* if *i* is even.

This procedure, if written in PLAN, could take the following form.

LABEL (1–6)	OPERATION (7–12)	ACC. (13–15)	OPERAND (16–72)	PROG. IDENT. (73–75)
#PROGRAM			/EVE	
#LOWER			LINK,SAVE	
#PROGRAM				
	OBEY		(1) [SET X3=ADDRESS OF I	
	LDX	6	(3)	
	STO	6	SAVE	
	OBEY		1(1)	
	LDX	6	SAVE	
	ANDN	6	1	
	ERN	6	1	
	STO	6	(3)	
	EXIT	1	2	
#END				

Array

Array elements are stored by the Algol compilers in consecutive storage units (1 or 2 words of store) so that the first subscript varies most rapidly and successive subscripts vary less rapidly. For example, an array declared as array [1:2, 1:2, 4:5]; would be stored in the following order.

a[1, 1, 4], *a*[2, 1, 4], *a*[1, 2, 4] *a*[2, 2, 4], *a*[1, 1, 5], *a*[2, 1, 5], *a*[1, 2, 5], *a*[2, 2, 5]

In order to ease the manipulation of complete arrays, information about the array is stored separately in an *array header*, which contains information such as the type of the array and the number of dimensions.

When the instruction in the standard calling sequence has been obeyed, X3 will contain the first word of the array header.

Two PLAN subroutines (GETAH and GENAH) are available for dealing with array headers and should be used whenever it is required to deal with arrays in a PLAN segment of an Algol program. GETAH will generate, in a specified area, an information block giving details of the array and GENAH generates an array header given the address of the information block. The information block used in GETAH and GENAH take the following form.

Word 0 Number of dimensions (*n*)

Word 1 Number of words per element.

Word 2 Base address, i.e. the address of the zero element.

Word 3	Address of the first word of the array.
Word 4	Address of the first word after the end of the array.
Word 5	First partial product.
Word 6	Second partial product.

.

.

.

Word $(n + 3)$	n th partial product.

Notes:

1 The zero element is the element with subscript [0, 0 ... 0]. If the array does not have a zero element, the zero element is the address of the word that would contain the element if it were present.

2 The k th partial product $p[k]$ of an n dimensional array with dimensions sizes $d[1]$, $d[2]$, $d[3] \ldots d[n]$ is defined as

$$P[k] = d[1] \times d[2] \times d[3] \ldots d[k]$$

3 FORTRAN arrays and array headers have the same format as described above.

GETAH

GETAH unpacks an Algol array header into an information block.

The calling sequence for GETAH is:

	CALL	1	GETAH
	LDX	3	A
	LDX	3	B

where A is a word containing the address of the array header,

B is a word containing the address of the first word of the area that is to contain the information block. For an n dimensional array, the length of the information block will be $(n + 4)$ words.

All accumulators are used by the routine but the overflow register is not used and will remain unchanged. No subroutines are called and the routine will return control to word CALL + 3

1 lower variable location and 35 program locations are used by GETAH.

GENAH

GENAH packs an information block into an Algol array header.

The calling sequence for GENAH is:

	CALL	1	GENAH
	LDX	3	B
	LDX	3	A

where B is a word containing the address of the first word of the information block.

A is a word containing the address of the first word of the area where GENAH is to generate the array header.

If A and B specify the same address, the effect will be to condense the information block into a standard array header.

It is recommended that the number of words set aside for the array header should be the same as the information block (although all the words may not be used).

All accumulators are used by the routine but the overflow register is not used and will remain unchanged. No subroutines are called and the routine will return control to word CALL + 3.

1 lower variable location and 26 program locations are used by the routine.

Example

procedure *copyarray* (a,b); **array** a,b ; **external;**

comment this procedure replaces an array a by another array b. It is assumed that the arrays have a similar structure and do not have more than three dimensions. The procedure is to be written in PLAN, using the subroutine GETAH.

LABEL 1–6	OPERATION 7–12	ACC. 13–15	OPERAND 16–72	PROG. IDENT. 73–75
#PROGRAM			/COPYARRAY	
#LOWER			LINK, IBA(7), IBB(7), ARRA, ARRB	
#PROGRAM				
	STO	1	LINK [STORE LINK ADDRESS	
	OBEY		(1) [SET X3=ADDRESS OF ARRAY A	
	STO	3	ARRA	
	OBEY		1(1) [SET X3=ADDRESS OF ARRAY B	
	STO	3	ARRB	
	CALL	1	GETAH	
	LDX	3	ARRA [ARRA=ADDRESS OF A	
	LDN	3	IBA [IBA=INFORMATION BLOCK FOR A	
	CALL	1	GETAH	
	LDX	3	ARRB [ARRB=ADDRESS OF B	
	LDN	3	IBB [IBB=INFORMATION BLOCK FOR B	
	LDX	3	IBA+4 [X3=ADDRESS OF END OF A	
	SBX	3	IBA+3 [X3=NUMBER OF WORDS IN A	
	LDX	7	IBB+4 [X7=ADDRESS OF END OF B	
	SBX	7	IBB+3 [X7=NUMBER OF WORDS IN B	

SHEET OF

LABEL 1–6	OPERATION 7–12	ACC. 13–15	OPERAND 16–72	PROG. IDENT. 73–75
	BXL	7	3,ERR [ERROR IF B TOO SMALL	
	LDX	1	IBA+3	
	LDX	2	IBB+3	
LAB1	BXGE	3	'513',LAB2 [TEST IF MORE THAN 512 WORDS	
	MOVE	1	0(3)	
	LDX	1	LINK [RESTORE LINK ADDRESS	
	EXIT	1	2 [EXIT TO ALGOL PROGRAM	
LAB2	MOVE	1	0 [MOVE FIRST 512 WORDS	
	SBN	3	512	
	ADN	1	512	
	ADN	2	512	
	BRN		LAB1	
ERR	SUSWT		2HE1 [ERROR HALT	
#END				

A call to the subroutine COPYARRAY would take the form

array *arra, arrb* [1:10, 1:12]

.

.

.

copyarray (, *arrb*);

This call would be equivalent to, but faster than, the Algol statement

for *i* : = 1 **step** 1 **until** 10 **do**

for *j* : = 1 **step** 1 **until** 12 **do**

arra [*i,j*] : = *arrb* [*i,j*];

String

When the instruction in the standard calling sequence has been obeyed, X3 will specify the address of the word containing the number of characters in the string. The word after the one specified in X3 points to the address of the word containing the first character of the string. (A string always starts in character 0 of a word in upper preset.)

Example

The first parameter of a procedure is a string of less than 512 characters. The procedure requires a pointer to the characters of the string. The PLAN procedure could begin as follows:

LABEL 1–6	OPERATION 7–12	ACC. 13–15	OPERAND 16–72	PROG. IDENT. 73–75
#PROGRAM			/ANNA	
#LOWER			LINK, POINT	
#PROGRAM				
	STO	1	LINK [STORE LINK ADDRESS	
	OBEY		(1) [X3=POINTER TO NO. OF CHARS	
	LDX	4	(3) [X4=NO OF CHARS	
	SRC	4	9 [FORM COUNT	
	ADX	4	1(3) [ADD ADDRESS	
	STO	4	POINT [STORE POINTER	

A character counter/modifier has been set up in POINT pointing to the first character of the string parameter.

Label

The coding required to branch to a label parameter is more complex than for the parameter types already considered. Therefore a subroutine, GOTOLAB, is used when it is required to branch to a label. The effect of GOTOLAB is the same as for the Algol statement goto.

The calling sequence for GOTOLAB is as follows:

LABEL	OPERATION	ACC.	OPERAND	PROG. IDENT.
	CALL	1	GOTOLAB	
	LDN	3	A	
	LDN	3	B	

or, if the parameters are in upper data,

LABEL	OPERATION	ACC.	OPERAND	PROG. IDENT.
	CALL	1	GOTOLAB	
	LDX	3	'A'	
	LDX	3	'B'	

where A is the address of the word containing the link address to the procedure,

B is the address of the word containing the position of the label parameter in the parameter list. (The first parameter in the list has position number 1, the second parameter has position number 2, etc.)

GOTOLAB uses accumulators X1, X2 and X3.

When the exit from a procedure is an unconditional branch to a label, an EXIT instruction is not needed.

Example

A procedure, declared as

procedure *hig* (*a, b, fail*); **integer** *a,b* ; label *fail* ; **external**;

is written in PLAN. The section of the segment referring to the label parameter will take the following form

LABEL (1–6)	OPERATION (7–12)	ACC. (13–15)	OPERAND (16–72)	PROG. IDENT. (73–75)
#PROGRAM			/HIG	
#LOWER			LINK	
#LOWER				
POSIT			3 [FAIL IS 3RD PARAMETER	
#PROGRAM				
	STO	1	LINK [STORE LINK ADDRESS	
	CALL	1	GOTOLAB [BRANCH TO FAIL	
	LDN	3	LINK	
	LDN	3	POSIT	
	LDX	1	LINK [RESTORE LINK ADDRESS	
	EXIT	1	3 [NORMAL EXIT TO ALGOL PROGRAM	
#END				

Switch

The subroutine GOTOSW is used when it is required to branch to a switch parameter and is similar in operation to GOTOLAB

The calling sequence for GOTOSW is as follows.

LABEL	OPERATION	ACC.	OPERAND	PROG. IDENT.
	CALL	1	GOTOSW	
	LDN	3	A	
	LDN	3	B	
	LDN	3	C	

or, if the parameters are in upper data,

LABEL	OPERATION	ACC.	OPERAND	PROG. IDENT.
	CALL	1	GOTOSW	
	LDX	3	'A'	
	LDX	3	'B'	
	LDX	3	'C'	

where A is the address of the word containing the link address to the procedure,

B is the address of a word containing the position number of the switch parameter in the parameter list (see GOTOLAB above),

C is the address of a word containing the value of the switch subscript.

The subroutine uses accumulators X1, X2, X3 and X6

Example

A procedure declared as

procedure *workout* (*s, tensor*); **array** *tensor* ; **switch** *s* ; **external;**

is written in PLAN. The sections of the segment relevant to the switch parameter will take the following form.

```
#PROGRAM        /WORKOUT
#LOWER          LINK,SBSRIPT
#LOWER
POSIT           1                   [S 1S 1ST PARAMETER
#PROGRAM
      STO    1  LINK                [STORE LINK ADDRESS
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      STO    7  SBSCRIPT            [X7=VALUE OF SWITCH
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      CALL   1  GOTOSW              [BRANCH TO SWITCH
      LDN    3  LINK
      LDN    3  POSIT
      LDN    3  SBSCRIPT
#END
```

Procedure

A procedure of type external (but not of type Algol) can itself be a parameter of a procedure of type external. A parameter procedure can be called by using the subroutine PROC, which has the following calling sequence

```
      CALL   1  PROC
      LDN    3  A
      LDN    3  B
      LDN    3  ARG1
      LDN    3  ARG2
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
      LDN    3  ARGN
```

where A is the address of the word containing the link address to the procedure,

B is the address of a word containing the position number of the procedure parameter in the parameter list (see GOTOLAB above),

ARG1, ARG2 ...ARGN are the addresses of arguments (i.e. parameters) required by the procedure parameter.

If these addresses refer to upper data areas, the LDX instruction should be used with a literal address.

The subroutine uses accumulators X1, X2, X3 and X6.

Example

A PLAN procedure, *xproc* , is written using a procedure parameter, which has two arguments, as the fifth parameter. The section of the coding involving the procedure parameter could take the following form.

```
#PROGRAM        /XPROC
#LOWER          LINK,PARA(2)
#LOWER
POSIT           5
#PROGRAM
      STO    1  LINK                [STORE LINK ADDRESS
```

LABEL	OPERATION	ACC.	OPERAND
	CALL	1	PROC
	LDN	3	LINK
	LDN	3	POSIT
	LDN	3	PARA
	LDN	3	PARA+1
#END			

Example

This example illustrates some of the methods of picking up parameters described in this section. Consider the Algol segment below. It is assumed that overflow cannot be set for any actual x and y

```
procedure all (x, y, s, l, p); value x,y; integer x,y;
                    switch s; label l;
                    integer procedure p;
      begin integer a;
            integer procedure even (x); value x; integer x; external;
            a := p (x↑2 + y↑2) - even (x);
            if a ≥ 10₁₀4 then goto l;
            if a < 0 then goto s[2] else if a = 0 then
            goto s[3];
            goto s [1]
      end;
```

If this procedure were written in PLAN, the following declaration should appear in the body of the Algol program.

```
procedure all ( x,y,s,l,p); value x, y; integer x,y ;
                  switch s; label l;
                  integer procedure p;
                  external;
```

The PLAN segment will take the following form.

LABEL (1–6)	OPERATION (7–12)	ACC. (13–15)	OPERAND (16–72)	PROG. IDENT. (73–75)
#PROGRAM			/ALL	
#LOWER				
TEN4			10000	
THREE			3,4,5	
#LOWER			XTEMP,XSQ,ARG,PX,LINK	
#PROGRAM				
	STO	1	LINK	
	OBEY		(1) [OBTAIN X AND EVALUATE X↑2	
	LDX	3	(3)	
	STO	3	XTEMP	
	MPY	3	XTEMP	
	STO	4	XSQ	
	OBEY		1(1) [OBTAIN Y AND EVALUATE Y↑2	
	LDX	3	(3)	
	MPY	3	3	
	ADX	4	XSQ [OBTAIN X↑2+Y↑2	
	STO	4	ARG	
	CALL	1	PROC [OBTAIN PARAMETRIC	

SHEET OF

LABEL (1–6)	OPERATION (7–12)	ACC. (13–15)	OPERAND (16–72)	PROG. IDENT. (73–75)
	LDN	3	LINK [PROCEDURE WHOSE	
	LDN	3	THREE+2 [PARAMETER	
	LDN	3	ARG [IS IN ARG	
	STO	6	PX [STORE P(X↑2+Y↑2)	
	CALL	1	EVEN [EVALUATE EVEN(X)	
	LDN	3	XTEMP	
	SBS	6	PX	
	LDX	6	PX [FORM P(X↑2+Y↑2)-EVEN(X)=A	
	SBX	6	TEN4	
	BNG	6	ON	
	CALL	1	GOTOLAB [BRANCH TO LABEL IF A≥10↑4	
	LDN	3	LINK	
	LDN	3	THREE+1	
ON	LDX	6	PX [SET X6=1, 2, 3	
	BPZ	6	TESTO [DEPENDING ON A +VE OR -VE OR ZERO	
	LDN	6	2	
	BRN		CON	
TESTO	BZE	6	*+3	

SHEET OF

LABEL (1–6)	OPERATION (7–12)	ACC. (13–15)	OPERAND (16–72)	PROG. IDENT. (73–75)
	LDN	6	1	
	BRN		CON	
	LDN	6	3	
CON	STO	6	SUB	
	CALL	1	GOTOSW [GOTO S(1), S(2) OR S(3)	
	LDN	3	LINK [DEPENDING	
	LDN	3	THREE [ON C(SUB)=1, 2 OR 3	
	LDN	3	SUB	
#END				

PROCEDURES USING UPPER COMMON VARIABLES

Care must be taken when **external** procedures using upper common variable store are incorporated into an Algol program.

The Algol program's stack (a work area used during the running of the object program) is in upper common variable and, as the stack may have to be extended during the program, it must be the last area of store allocated to the program. Therefore, no other common variables must occur in higher addresses than the end of the stack. If this does occur, the stack and the upper common variables in higher address will be corrupted.

There are two methods that can be used to overcome this problem.

1 If no 'SPACE' statement appears in the program description, a stack of 3, 000 words will be allocated at the top of the program's store thus permitting any extension of the store that is necessary.

2 The 'SPACE' directive should specify sufficient store to avoid the need for extending the stack area.

Method 1 is, in most cases, more reliable.

Note: Some of the graph plotter routines and any FORTRAN segment containing a COMMON area that is not preset by means of the DATA statement contain upper common variables.

CONSOLIDATION

The subroutines GOTOLAB, GOTOSW and PROC are included in the SRA1 subroutine block of the Algol library. Therefore, when a program is consolidated by an Algol paper tape compiler, these subroutines will be incorporated in the object program if necessary. If, however, PLAN segments require subroutines from a PLAN library group, the programmer must ensure that the appropriate subroutines are presented in semi-compiled form to the Algol compiler.

PART 2 OPERATING CONSIDERATIONS

Chapter 10 System Description

XALP and XASP are paper tape versions of the 1900 Algol compiler. XALP is intended for use with core stores of at least 32K words and XASP for stores of 16K words. Although XALP provides wider programming facilities than XASP, the two compilers have almost identical operating characteristics.

Algol programs can be input to the compilers from cards or paper tape. Programs input on paper tape can be punched in *normal* or *full* mode, where normal mode consists of the 1900 Series 64-character card code and full mode also allows the use of lower case letters.

The source program is translated into an equivalent semi-compiled program, which is output to paper tape. If consolidation is required, the Algol library (issued with the compiler, on paper tape) or a private user's library must be read. Loading messages, a request slip, the General Purpose Loader (G. P. L.) and consolidated leader information are then output, also on paper tape. If consolidation has not been requested, the compiler halts without reading the library or producing the G. P. L.(Note: The programmer can inhibit the production of semi-compiled output.)

The semi-compiled program is separated from the information output during consolidation (G. P. L., request slip, etc.) by a length of blank tape in the middle of which is one delete character. When the program is to be run, the two parts are separated at the delete character. The G. P. L. and associated information are then input before the semi-compiled program is loaded.

It may be necessary to read in the Algol library (or private user's library) again at this stage because the programmer can choose whether or not the relevant library segments are copied to the semi-compiled output during compilation. If the library segments are not copied the library must be re-read. (Note: There is a standard library program XMUL which can split the Algol library segments in the other. If this has been done, only the segment leaders need be read during compilation and the semi-compiled segments are read when the program is being loaded. The specification of XMUL is given in the manual *Algol: Compiler Library Procedures*).

Algol programs can be written in more than one segment and XALP allows different segments to be input from different devices and in differing modes (normal or full) and types (source or semi-compiled). When there is a change of device, the previously used peripheral is released by XALP and the requested one is allotted; a change in mode or type will cause the peripheral to be disengaged only. With XASP, however, no switching is allowed between segments and therefore semi-compiled segments can only be input when the Algol library is being read.

LISTING

The programmer can obtain a listing on the line printer or paper tape punch. In both cases a *full* or a *short* listing can be chosen. On the line printer, a full listing contains a copy of the source program, with a statement number assigned to each line, error messages (for use by the programmer) and details about the name of the compiler, data and time (if available) of compilation, amount of core store used, name of the program and whether or not the compilation was error free. The short listing does not contain a copy of the source program but merely prints a list of the statement numbers, error messages and information about the nature of the compilation as for a full listing. Characters which appear in the source program as lower case letters are overprinted with an apostrophe in the full listing.

Note: The programmer can choose to have no listing produced.

ERROR MESSAGES

Error messages output with the listing are of interest to the programmer only. Other messages can be produced at run time on any device being used by the Algol program (or on the console typewriter, if no device is being used) and these are also of use only to the programmer.

Messages of interest to the operator are always output on the console typewriter and these are listed in Chapter 11, page 45.

ENTRY POINTS

There are five entries to XALP and XASP (XA*x*P represents either XALP or XASP).

Console Message typed by operator	*Instruction to compiler*
GO #XA*x*P 20	Begin compilation by reading source program from cards.
GO #XA*x*P 21	Begin compilation by reading source program from paper tape, in normal mode.
GO #XA*x*P 22	Begin compilation by reading source program from paper tape, in full mode.
GO #XA*x*P 28	Abandon compilation (the buffers are emptied and all peripherals are released).
GO #XA*x*P 29	Force consolidation.

Chapter 11 Operating instructions

In this chapter, XA*x*P represents either XALP or XASP and *name* represents the four-character name of the Algol program being compiled.

The compiler as supplied has a priority of 50.

Narrative	*Console Message*
COMPILATION	
1 Load the compiler.	LO #XA*x*P
2 Load the Algol source program in the appropriate reader.	
3 (a) If the source program is on cards, activate the compiler by typing:	GO #XA*x*P 20
(b) If the source program is on paper tape, activate the compiler by typing either	
(i) if the program is punched in normal mode:	GO #XA*x*P 21
or	
(ii) if the program is punched in full mode:	GO #XA*x*P 22
4 The Algol program is read and semi-compiled segments are optionally output on paper tape. If the program is on more than one reel of paper tape, the message:	UNIT *y* FIX
is output after each tape has been read, where *y* is the unit number of the paper tape reader from which the program is being input.	
5 When all the Algol program has been read, the message:	UNIT *y* FIX
is output. If consolidation is requested, the Algol library (or library leaders) is input from the paper tape reader with unit number *y*. Loading messages, a request slip, the G. P. L. and consolidated leader information is output on the same tape as the semi-compiled program.	
When the compilation is successfully completed, the compiler halts with the message:	0#XA*x*P; HALTED:-EC

LOAD AND RUN

6 The compiler output is in two parts:

(a) the semi-compiled version of the source program,

(b) loading message, request slip, G. P. L. and consolidated leader information.

The two parts are separated by a length of blank tape in the middles of which is one delete character (eight holes).

7 To load the program, tear the tape at the delete character and place the second part (b) in a paper tape reader. Press the appropriate 'F' button or 'Control + A' to read the Executive LO message at the front of the tape. The message: — LO # *name* 0

is output on the console typewriter and part (b) is read.

8 After part (b) has been read, the message: — UNIT *y* FIX

is output. Part (a) is now read and either

(a) the paper tape reader is released and the program halts with the message: — 0# *name* ; HALTED:- LO

signifying that the object program has been loaded,

or

(b) the message: — UNIT *y* FIX

is output after tape (a) has been read signifying that the Algol library (or semi-compiled library segments) must be input on the tape reader with unit number *y*. After reading the library, the program will halt with the message: — 0# *name* ; HALTED:- LO

signifying that the object program has been loaded.

9 To activate the program, type: — GO # *name* 20

10 The program halts with the end of run message: — 0# *name* ; HALTED:-AH

ERROR HALTS

Compilation

Message	*Reason*	*Action*
0#XA*x*P; HALTED:- CE	Checksum error when reading semi-compiled program.	Either (a) if the error is on paper tape, backspace one block and continue by typing: GO #XA*x*P or (b) abandon the run by typing: GO #XA*x*P 28
0#XA*x*P; HALTED:- CP	No card punch is available.	Make a card punch available and type: GO #XA*x*P
0#XA*x*P; HALTED:- CR	No card reader is available.	Make a card reader available and type: GO #XA*x*P

Message	*Reason*	*Action*
0#XA*x*P; HALTED:- LF	The program is too large for the compiler to cope with.	Abandon the run by typing: GO #XA*x*P 28
0#XA*x*P; HALTED:- LP	No line printer is available.	Make a line printer available and type: GO #XA*x*P
0#XA*x*P; HALTED:- NEEDS *procnames*	Unsatisfied procedure calls, where *procnames* is a list of the missing procedure segments up to a total of 40 characters. (This message will only occur when the 'LIBRARY' statement is used or with a single segment program.)	Either (a) abandon the run by typing: GO #XA*x*P 28 or (b) if available, provide the semi-compiled segments and type: GO #XA*x*P or (c) force consolidation by typing: GO #XA*x*P 29
0#XA*x*P; HALTED:- PF	Peripherals free. This message occurs after GO #XA*x*P 28	The compilation has been abandoned A new compilation may now be started.
0#XA*x*P; HALTED:- TP	No paper tape punch is available.	Make a paper tape punch available and type: GO #XA*x*P
0#XA*x*P; HALTED:- TR	No paper tape reader is available.	Make a paper tape reader available and type: GO #XA*x*P
0#XA*x*P; HALTED:- ZZ	Errors have been detected in the compilation. No loadable program has been produced.	Destroy any semi-compiled output that may have been produced and either (a) if there are more programs to be compiled, return to operating instruction 3 or (b) abandon the run.

Execution

Message	*Reason*	*Action*
0# *name*; HALTED:- CP	No card punch is available.	Make a card punch available and type: GO # *name*
0# *name*; HALTED:- CR	No card reader is available.	Make a card reader available and type: GO # *name*
0# *name*; HALTED:- EE	Execution error detected. All output is completed.	Either (a) If another set of data is to be read, return to operating instruction 9 or (b) abandon the run.

Message	*Reason*	*Action*
0# *name* ; HALTED:- LP	No line printer is available.	Make a line printer available and type: GO # *name*
0# *name* ; HALTED:- RL	A program with own arrays has no more stack space available	Type: GO # *name* 28 which will empty the buffers and release all peripherals. If more core store is available, reload the program by direct console operation and specify a larger store allocation in the LO message. The tape (b) of operating instruction 7 must be read without its LO message.
0# *name* ; HALTED:- ST	Program stack requires more space.	Either (a) make more core store available, if possible, and type: GO # *name* or (b) if no extra store is available, abandon the run by typing: GO # *name* 28 which will empty the buffers and release all peripherals.
0# *name* ; HALTED:- TP	No paper tape punch is available.	Make a paper tape punch available and type: GO # *name*
0# *name* ; HALTED:- TR	No paper tape reader is available.	Make a paper tape reader available and type: GO # *name*
0# *name* ; HALTED:- X*c*	An impermissible character, *c* , has been detected in a numeric input field using the 1900 Algol input/output system. If *c* is ? (question mark), the impermissible character is one of ? $] ↑ ← or a delta shift character other than FE2 or FE5.	Either (a) to abandon the run, type: GO # *name* 28 or (b) if it is required to continue the run, type: GO # *name* (The impermissible character will be replaced by zero.)

OTHER CONSOLE MESSAGES

Message	*Reason*
0#XA*x*P; DISPLAY:- COMPILED *fullprogname*	This message appears immediately before the compiler finally suspends itself (i.e. for halts EC or ZZ) where *fullprogname* is the full name of the program, consisting of the four character name identifying the program plus an optional accounting code.
0# *name* ; DISPLAY:- ED	The program has attempted to read more data after the end-of-data marker (****) has been read.
0# *name* ; DISPLAY:- IC	No input or output channel has been selected.

Note: Other information of interest only to the programmer may be displayed on the console typewriter.

APPENDICES

Appendix 1 Compilation Error Numbers

Errors in the syntax of a program are detected when the source program is being read. If no syntactic errors are discovered, any semantic errors in the program are found.

SYNTACTIC ERRORS

These errors cause an error message to be output immediately after the line containing the error. The message takes the form:

STATEMENT *x* SYNTAX ERROR *n*

where statement *x* has error type *n*. The error numbers marked with an asterisk (*) will have an identifier associated with the error message.

Error Number	*Reason*
301	A minus sign is preceded by an operator that should not appear before a simple arithmetic expression.
302	A statement starts with an improper operator.
303	An arithmetic expression starts with an improper operator.
305	A statement is wrongly formed, probably containing an extra left parenthesis or left bracket.
306	'SWITCH' is not followed by an identifier.
307	A declarator is followed by a basic syntactic element which is neither an identifier nor a declarator.
308	A primary is followed by a primary.
309	A left delimeter is not matched by a proper right delimeter. For example, $(x+y$ or $x+y$ 'THEN' $x+y)$.
310	'THEN' is followed by an expression that is not concluded by 'ELSE'.
311	A comma has been misplaced.
312	A for list contains a comma or an assignment operator followed by an expression that is concluded by an improper right delimeter. The only proper right delimeters in this case are: 'DO' 'STEP' 'WHILE' , (comma)
313	A for list contains 'WHILE' or 'STEP' or 'UNTIL' followed by an expression that is concluded by an improper right delimeter. The only proper right delimeters in this case are: 'UNTIL' 'DO'
314	A full stop is not followed by a digit.
315	A procedure declaration has an incorrect format.
316	Either (a) the first 'BEGIN' of a program is preceded by 'END' or (b) 'INTEGER', 'BOOLEAN' or 'REAL' is not followed by 'PROCEDURE' at the beginning of a segment.

Error Number	*Reason*
319	'ELSE' is not matched by 'THEN'.
320	An assignment operator (:= or ←) has been misplaced.
321	An improper parameter delimeter has been detected.
322	An assignment operator is missing in a switch declaration.
332	A left parenthesis is missing in a program description statement.
333	A left parenthesis in a program description statement is not followed by an identifier.
334	A right parenthesis is missing in a program description line.
336	A non-numeric channel number has been specified in a program description statement (XALP only).
337	An error has been detected in a program description statement referring to a magnetic tape device.
338	The number of line printer print positions specified in a program description statement is non-numeric (XALP only).
341	A parenthesis appears in a 'SPACE' , 'TRACE' or 'PRIORITY' statement.
342	A non-numeric symbol appears after 'SPACE', 'TRACE' or 'PRIORITY'.
350	There is an error in an input/output program description statement. The device specified is not an acceptable type for the compiler or is the wrong type for the statement (XALP only).
351	A 'PRIORITY' statement has been specified for an unconsolidated program (XALP only).
352	There is an error in the format of a number.
*400	An illegal character has been detected (XASP only).
*401	An illegal symbol has been detected. A frequent cause of this error is that one of the apostrophes surrounding a correct Algol symbol has been omitted. The compilation continues assuming 'INTEGER' has been read (XASP only).
402	The compiler is in an indeterminate state. This is usually caused by a previous error.

There are two non-numbered errors (with XALP only):

*ILLEGAL CHARACTER	An illegal character has been detected.
*ILLEGAL SYMBOL	A non-Algol symbol has been written between apostrophes (e.g. 'BENGI' instead of 'BEGIN'). A frequent cause of this error is that one of the apostrophes surrounding a correct Algol symbol has been omitted.

SEMANTIC ERRORS

These errors cause an error message to be output to the listing device at the end of the segment. The message takes the form:

STATEMENT *x* ERROR *y*

where statement *x* has error type *y*. The error numbers marked with an asterisk (*) will have an identifier associated with the error message.

Error Number	*Reason*
1	The program is too large for the compiler to cope with. This corresponds to the console typewriter message, HALTED:- LF

Error Number	*Reason*
2	The compiler has no more symbol space available. The program cannot be compiled in its existing form.
3	A request for core store in the compiled program exceeds 32, 767 words. The program is too large to be run in compact mode on a 1900 Series computer.
70	An impossible type matching has been attempted in an arithmetic expression (this error is detected only for the operators + - * /).
77	A procedure that is not a function designator appears when an expression was specified.
*78	Too few parameters have been supplied to a procedure.
*79	A parameter presented to a procedure has not been specified or has been incompatibly specified (e.g. 'STRING' has been presented where 'REAL' has been specified).
80	An expression is too deeply bracketted for the present compiler to cope with.
*81	A formal parameter of a procedure has been specified to be called by value where the parameter has a kind (for example, switch identifier) that makes a call by value meaningless.
*82	A duplicate label definition has been detected.
*83	An identifier that appears in a switch list has been declared something other than a label.
*84	An identifier in an arithmetic or Boolean expression has not been previously declared.
*85	This parameter has not been specified.
*86	A switch identifier is not followed by a left bracket ([) or a comma (,).
*87	This identifier is an undefined label.
*88	A label has been used outside a designational expression.
*89	A simple variable is followed by a left bracket.
*90	A simple variable is followed by a left parenthesis.
*91	A procedure identifier is not followed by a proper operator.
*92	An array identifier is not followed by a proper operator.
*98	A duplicate declaration of an identifier has been detected.
99	A 'VALUE' part contains information other than identifiers and commas.
102	An attempt has been made to use 'OWN' as a specifier.
103	'OWN' is not followed by a declarator.
104	An attempt has been made to use 'VALUE' as a declarator.
105	'PROCEDURE' is not followed by an identifier.
106	The formal parameter part of a declaration is not followed by a semi-colon.
107	'OWN' is followed by a declarator or specifier that is illegal after 'OWN'.
108	Information other than an identifier is contained in a type declaration.
109	The declarator which follows a type declarator is improper (for example, 'REAL' 'OWN').
110	An attempt has been made to use 'STRING' as a declarator.
111	An identifier in an array has been followed by an operator which is neither a comma nor a left bracket.
114	An operator other than a colon or semi-colon has been used in a specification.
116	'ARRAY' is not followed by an identifier.

Error Number	*Reason*
*151	An improper identifier has been included in a specification, i.e. the identifier did not appear in the formal parameter list.
163	In an array declaration, either an upper or a lower bound has been omitted.
*191	An illegal procedure identifier has been used, i.e. the procedure is not type 'PROCEDURE'.
200	An incomplete statement of the form: A < *operator* > B; has been detected, where A is an expression, B is an identifier or a constant and *operator* is an arithmetical, Boolean or relational operator (for example, X = 7; will produce this error).
201	An impossible type matching situation has occurred in an arithmetical or assignment statement.
202	Previous errors have caused the compiler to flag an error at a right bracket.
203	A comma has been used in a switch designator.
204	An operand of an integer division or of a Boolean operator was not 'INTEGER' or 'BOOLEAN' in type.
205	An impossible type matching situation has occurred in a conditional expression.
206	A non-identifier has appeared as a label.

There is one non-numbered error:

IMPOS. HAPPENED	A previous error has confused the compiler.

OTHER MESSAGES

The following messages can be output on the listing device, usually between segments.

CHECKSUM ERROR	A checksum error has occurred when reading the semi-compiled program. The run should be repeated and the semi-compiled program regenerated, if necessary. If it is suspected that there has been a reader error, the semi-compiled tape or card can be repositioned to the start of the current record and the operator can restart the program by typing GO #XA*x*P
PROGRAM INCOMPLETE	The end of program marker (****) has occurred before the end of the program.
NEEDS	When a pause block is encountered after reading a semi-compiled segment and the compiler is reading in selective mode (after the 'LIBRARY' or 'FINISH' statement) the names of the missing segments are listed after the NEEDS message. The operator can force consolidation, abandon the compilation or continue reading more segments so that consolidation can be eventually completed.

Appendix 2 Execution Error Numbers

Information about errors detected during the execution of the program is output to the currently selected output device. If no output device has been selected or if 'OMIT IO' appears in the program description, the error message is displayed on the console typewriter immediately before the HALTED:- EE message, which is always output when an execution error is detected.

TRACE LEVEL 0 provides detection facilities for error numbers 10 to 59 and error numbers 70 to 79.

TRACE LEVEL 1 provides detection facilities for error numbers 60, 61 and 62 as well as for those provided at trace level 0.

TRACE LEVEL 2 provides extended detection facilities for error number 62, as well as for those provided at trace levels 0 and 1. (Note: The 'OMIT IO' statement cannot be incorporated in programs using trace level 2.)

The trace facilities available with the paper tape compilers are fully described in Chapter 7, page 21.

The error message takes the form

ERROR TYPE *n*

where *n* is the error number.

Error Number	*Reason*
10	The evaluation of zero raised to a negative power has been attempted.
11	*sqrt* has been called with a negative parameter.
12	*ln* has been called with a negative or zero parameter.
13	Overflow of an *exp* function has occurred.
14	The result of $a \uparrow i$ has overflowed (where *a* is real or integer and *i* is integer).
31	An attempt has been made to read past the end of data marker (****).
33	An attempt has been made to input a number, real or integer, the value of which lies outside the permitted range.
34	No input or output channel has been selected.
40 to 49	These errors are associated with the magnetic tape backing store package (see the manual *Algol: Magnetic Tape Compiler*).
50	The lower bound is larger than the upper bound in an array declaration.
51	A mixed language program has run out of storage space. The program should be recompiled with a larger core store allocation in the 'SPACE' statement. (Note: This error will occur only with Algol segments that are included in programs written in another language.)
52	A program containing own arrays has run out of storage space. The program should be recompiled with a larger core store allocation in the 'SPACE' statement.
60	An array element has been found to be outside the declared bounds of its array - an array element subscript is too small.
61	An array element has been found to be outside the declared bounds of its array - an array element subscript is too large.
62	Overflow has occurred.
70	The dimensions of an array presented to a *read array* or a *write array* procedure do not correspond in number or size to the dimensions of the formal parameter.

Error Number	*Reason*
71	A column or row checksum is not at the end of a column or row of an array being input.
72	The grand checksum of an array being input is not followed by the character £.
73	One of the checksums of an array being input is incorrect.
74	S, G or Z is not followed by a semi-colon.
75	An improper symbol has been read by the procedure *inbasic* .
76	An improper symbol has been found by the procedure *outbasic* .

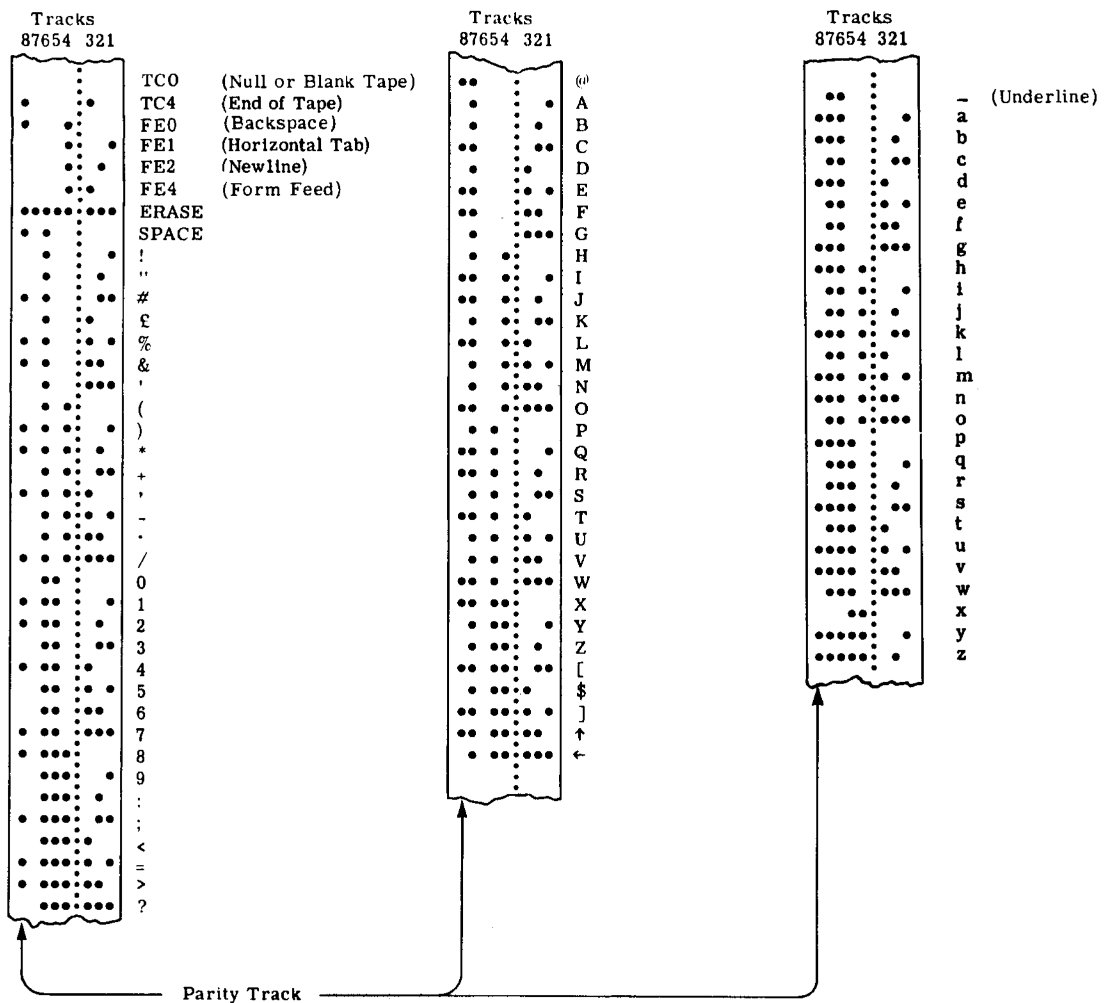
Note: Error numbers 70 to 79 are associated with the KDF9 input/output system.

Appendix 3 1900 Series Card And Paper Tape Codes

CARD CODE

Symbol	Card Punching	Symbol	Card Punching	Symbol	Card Punching
0	0	F	10/6	- (minus/hyphen)	11
1	1	G	10/7	" (quotes)	11/0
2	2	H	10/8	/ (solidus)	0/1
3	3	I	10/9	+ (plus)	10/2/8
4	4	J	11/1	. (stop)	10/3/8
5	5	K	11/2	; (semi-colon)	10/4/8
6	6	L	11/3	: (colon)	10/5/8
7	7	M	11/4	' (apostrophe)	10/6/8
8	8	N	11/5	! (exclamation)	10/7/8
9	9	O	11/6	[(left bracket)	11/2/8
space	NONE	P	11/7	$ (dollar)	11/3/8
& (ampersand)	10 or 10/0	Q	11/8	* (asterisk)	11/4/8
# (number)	3/8	R	11/9	> (greater than)	11/5/8
@	4/8	S	0/2	< (less than)	11/6/8
((left parenthesis)	5/8	T	0/3	↑	11/7/8
) (right parenthesis)	6/8	U	0/4	£ (pound)	0/2/8
] (right bracket)	7/8	V	0/5	, (comma)	0/3/8
A	10/1	W	0/6	% (percentage)	0/4/8
B	10/2	X	0/7	? (question)	0/5/8
C	10/3	Y	0/8	= (equals)	0/6/8
D	10/4	Z	0/9	←	0/7/8
E	10/5				

1900 8- TRACK PAPER TAPE CODE



4112(8.68)

Appendix 4 1900 Algol Hardware Representation

Algol Basic Symbols	1900 Representation: Cards and Paper Tape Normal Mode	1900 Representation: Paper Tape Full Mode
a	A	a
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
z	Z	z
A	no representation	A
.		.
.		.
.		.
.		.
.		.
Z		Z
0	0	0
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
9	9	9
true	'TRUE'	'true'
false	'FALSE'	'false'
array	'ARRAY	'array'
Boolean	'BOOLEAN'	'Boolean'
.	.	.
.	.	.
.	.	.
.	.	.
.	.	.
value	'VALUE'	'value'

Algol Basic Symbols	1900 Representation: Cards and Paper Tape Normal Mode	1900 Representation: Paper Tape Full Mode
while	'WHILE'	'while'
+	+	+
—	—	—
×	*	*
/	/	/
÷	'/'	'/'
↑	↑ or '**'	↑ or '**'
<	< or 'LT'	< or 'lt'
≤	'LE'	'le'
=	= or 'EQ'	= or 'eq'
≥	'GE'	'ge'
>	> or 'GT'	> or 'gt'
≠	# or 'NE'	# or 'ne'
≡	'EQUIV'	'equiv'
⊃	'IMPL'	'impl'
∨	'OR'	'or'
∧	'AND'	'and'
¬	'NOT'	'not'
,	,	,
.	.	.
$_{10}$	& or '10'	& or '10'
:	:	:
;	;	;
:=	:= or ←	:= or ←
⊔	% or '-'	% or '-'
(	(	(
)	)	)
[	[or '<'	[or '<'
]	] or '>'	] or '>'
⌈	'('	'('
⌉	')'	')'

Letters between apostrophes in basic symbols in the right-hand columns may alternatively be upper case with no change of meaning.

Index

	Page
1900 8-track paper tape code	58
1900 64 -character card code	2, 17, 57
1900 Algol hardware representation	2, 17, 59
algol	30
Algol:	
library	1, 16, 17, 29, 43
paper tape compilers (XALP and XASP)	1, 2
procedure segments:	30
declaration	30
procedure	30
Array parameters in mixed language programs	34 to 37
Boolean parameters in mixed language programs	34
Card code, 1900 64-character	2, 17, 57
Channel numbers	3, 7
Compilation:	1
error messages	13, 14, 44, 46, 47, 49, 51 to 54
error numbers	13, 51 to 54
Compiler:	1
input	17 to 20
output	15, 16
Consolidation:	1, 3
forcing of	3, 44
mixed language programs, of	42
'CONTINUE' statement	5, 19
'COPY' statement	5, 16
Entry points	3, 44
Error halts:	46 to 49
compilation	46, 47
execution	47 to 49
Error messages:	44
compilation	46, 47, 49, 51 to 54
execution	47 to 49, 55, 56
Error numbers:	51 to 56
compilation	13, 51 to 54
execution	55, 56
Execution:	
error messages	47 to 49, 55, 56
error numbers	13, 55, 56
external	30, 31, 32
'FINISH' statement	5, 16
FORTRAN procedure segments:	30, 31
calling by name	31
declaration	31
procedure	31
Full listing	9, 10
Full mode	2, 17, 59
GENAH	34, 35
General purpose loader (G. P. L.)	1
GETAH	34 to 37
GOTOLAB	37, 38
GOTOSW	38, 39

Hardware representation, 1900 Algol 2, 17, 59

Input/output procedures 3, 7
'INPUT' statement 5, 7
Integer parameters in mixed language programs 33

Label parameters in mixed language programs 37, 38
Library:
    Algol, standard 16, 17, 29
    user's 17, 19, 20
'LIBRARY' statement 5, 19, 20
Listing: 9 to 14
    basic 11
    examples of 10
    full 11
    intermediate 11
    statements 11 to 13

Master segment 17, 29, 30
Mixed language programs: 29 to 42
    Algol 30
    Algol procedure segments 30
    communication of data 29
    consolidation of segments 42
    external 30 to 32
    FORTRAN procedure segments 30, 31
    PLAN procedure segments 32 to 40
    procedure segments 29
    upper commonvariables, use of 41
Modes, paper tape: 2, 17, 59
    full 59
    normal 59
Monitor points 22
monoutput procedure 25
mplist procedure 24
mpname procedure 24
Multi-reel programs 18

Normal mode 2, 17, 59

Object program: 1, 7, 8
    input/output 7, 8
off procedure 25
'OMIT IO/statement 6, 8
on procedure 25
Operating instructions 45 to 49
'OUTPUT' statement 5, 7
overflow procedure 24

Paper tape code, 1900 8-track 58
pause procedure 24
Peripheral description statements: 7, 8
    'INPUT' statement 7
    'OMIT IO/ statement 8
    'OUTPUT' statement 7
PLAN procedure segments: 32 to 40
    declaration 32
    picking up parameters in 32
'PRIORITY' statement 5, 27
PROC 39
Procedures:
    library of 17, 29
    parameters in mixed language programs 39
    segments 17
    using upper common variables 41

Program description:	3
statements	5, 6
'PROGRAM' statement	5, 15
Program testing:	21 to 25
procedures	23 to 25
system switches	25
'READ FROM' statement	5, 18, 19
Real parameters in mixed language programs	33
Segmentation	17, 29
'SEGMENTS' statement	5, 16
Semantic errors	9, 52 to 54
Semi-compiled:	
form	1
segments	19, 29
'SEND TO' statement	5, 15
Source program	1
'SPACE' statement	5, 27, 28
SRA1 subroutine block	17, 29
Standard calling sequence	32
String parameters in mixed language programs	37
Switch parameters in mixed language programs	38, 39
Syntactic errors	9, 51, 52
System switches	25
test procedure	25
Trace:	
level 0	22
level 1	22
level 2	22, 23
levels	22
list	22, 23
'TRACE' statement	5, 21
XALP	1, 2
XASP	1, 2
XMUL	16, 43

AMENDMENT RECORD

Should any additions or amendments to this Manual be necessary, they will be issued in the form of Amendment Lists.
When such changes have been incorporated, an appropriate entry should be made in the table on the left and the pink instruction sheet(s) should be filed behind this page.

List Number	*Item Numbers*
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to
	to