ATLAS
COMPUTER
LABORATORY

PLASYD
MANUAL

SCIENCE
RESEARCH
COUNCIL

P L A S Y D

M A N U A L

BY

J D THEWLIS,

D C TOLL

AND

F R A HOPGOOD

Atlas Computer Laboratory
Chilton
Didcot
Berkshire
OX11 0QY

November 1973

PREFACE

PLASYD is an ICL internal language which has been used in the production of several of the company's major software products. The most significant example is the optimising FORTRAN compiler, XFEW, and its derivatives. The Atlas Computer Laboratory has permission to use PLASYD on the understanding that the product is unsupported and not necessarily bug-free. This manual is mainly a rewrite of ICL internal publications with modifications indicating how PLASYD will be available on the Laboratory's 1906A. We would like to thank ICL for their co-operation in making the necessary internal publications available to us and permission to incorporate parts of them in this manual. In particular, we would like to dedicate this manual to J K Buckle, of ICL, the designer of the language.

CONTENTS

1.

INTRODUCTION

1.1 Programming Language for System Development

PLASYD, standing for Programming Language for System Development, was designed in mid-1968 as a basic tool for the production of new optimising and conversational FORTRAN compilers for the ICL 1900 series of computers. Although previous scientific compilers for the 1900 series had used special purpose languages for syntax analysis and other specialised tasks, a large amount of the coding was done in PLAN [1]. It was felt that for the new projects a more systematic, easily intelligble and maintainable system would be preferable and an investigation of software production tools was undertaken. The main criteria for the language were that it should itself provide a high standard of documentation and the code generated should be efficient (the compilers to be produced had strict limits both on total size and on compilation time).

PLASYD needed to use all the machine facilities efficiently and also have an algorithmic structure to aid readability. These conditions pointed to an implementation language similar to PL360 [2] but adapted for the 1900 series order code. PL360's block structure and all operations concerned with flows of control were adapted almost intact, being virtually machine independent. However storage types and methods of manipulating them had to be modified to fit in with the 1900 series architecture. In particular, a method of exploiting the directly and indirectly addressable areas of data storage on the 1900 was devised.

1.2 General Form of the PLASYD Language

PLASYD is, superficially, Algol-like, having a block structure, type and procedure declarations, the same form of identifiers and similar assignment and control statements. However, recursion is not provided automatically and consequently all storage assignments may be made at compile time. The compiler generates semi-compiled output similar to other 1900 series compilers. Segmentation and common storage are provided in the language so that it is possible to compile PLASYD routines which can communicate with both FORTRAN and ALGOL programs. Consequently, it is quite feasible to recode the most frequently used routines of a program in PLASYD for greater efficiency.

The most fundamental difference between PLASYD and high level programming languages is the splitting of variables into two classes, accumulators and cells. Accumulators correspond directly to the 1900 integer and floating point accumulators whereas cells occupy core storage and correspond closely to variables in high level languages. The ability to specify the accumulators to be used gives the programmer some control over the form of instructions to be compiled. Unlike high level languages, assignment statements do not allow parentheses and all operators, including the assignment itself, are

obeyed strictly from left to right. For example, the statement

```
X1 : = A - B AND 3 SRL 2 ;
```

produces code which :

(1) Loads integer accumulator X1 with A

(2) Subtracts B from X1

(3) Collates X1 with 3

(4) Shifts the contents of X1 right logically two binary places

In general, each operator/operand pair in an assignment statement produces a single machine code instruction. However in some cases more than one instruction may be generated.

1.3 Layout

PLASYD is, basically, a free format language except that reserved words, such as BEGIN, must be delimited by spaces or newline. PLASYD programs are written using the 1900 64-character set of symbols and can have a maximum of 72 characters in any line. If the program is to be punched on paper tape TAB characters may be used. TABs are considered to be at six character intervals by the compiler and are equivalent to spaces. A SPACE, TAB, NEWLINE or 'END OF CARD' will serve to terminate an identifier or basic symbol. These layout characters can, in general, be used to improve the format of the program. Spaces, however, are significant in strings and character sequences. Newline is also significant within some assignment statements and the INCLUDE statement. The symbol ':=' may be replaced by '←'. This is not to be advised when inputting PLASYD from a MOP terminal.

1.4 Syntax Definitions

The intention is to provide a fairly rigorous definition of the syntax of the PLASYD language at the heads of the various sections. The notation is similar to BNF in that:

(1) ::= is used to mean 'is defined as'

(2) | is used to mean 'or'

Terminal symbols in PLASYD are represented by upper case letters and other symbols in the 1900 character set. Syntactic classes are represented by lower case names without spaces. For example:

```
address ::= wordaddress | CHAR integer | CHAR integer OF wordaddress
| @ identifier
```

indicates that the syntactic class 'address' is defined as one of the

following four possibilities:

(1) The syntactic class 'wordaddress'

(2) The characters CHAR followed by the syntactic class 'integer'

(3) As (2) but followed by the characters OF and the syntactic class 'wordaddress'

(4) The character @ followed by the syntactic class 'identifier'

In some places, several syntactic statements are similar except that one class is replaced by another similar one consistently throughout the statement. In this case they may be written as a single statement using Greek letters to indicate one of several possibilities. For example:

```
αcell ::= simpleαcell | (simplesmo)  {α = x | r }
```

is equivalent to the two statements:

```
xcell ::= simplexcell | (simplesmo)
rcell ::= simplercell | (simplesmo)
```

Throughout the syntax definitions the following associations apply:

```
x     INTEGER
r     REAL
xx    LONG INTEGER
rr    LONG REAL
```

The symbol ? is used to indicate that the preceding syntax class is optional. For example:

```
a ::= b c? d
```

is equivalent to

```
a ::= bd | bcd
```

BASIC SYMBOLS AND COMMENTS

2.1 Syntax

```
letter        ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

digit         ::= 0|1|2|3|4|5|6|7|8|9

char          ::= letter|digit|+|-|*|/|<|=|>|#|,|.|;|:|@|&|%|
                  (|)|[|]|£|$|?|!

reservedword  ::= ACC|AND|A1|A12|
                  BASE|BEGIN|
                  CARRY|CASE|CASEND|CH|CHAR|CNT|
                  DATA|DEFINE|DO|
                  ELSE|END|ENTRY|ER|EX|EXTERNAL|
                  FIX|FLOAT|FOR|FOVERFLOW|FROM|FUNCTION|
                  GLABEL|GLOBAL|GLOBEND|GO|GOTO|
                  IF|INCLUDE|INTEGER|
                  LA|LOGICAL|LONG|LOWEND|LOWER|
                  MINUS|
                  NEG|NONPLASYD|NOT=|NULL|
                  OBEY|OF|OR|OVERFLOW|
                  PROCEDURE|PURE|PUREND|
                  REAL|RETURN|
                  SA|SLA|SLC|SLL|SRA|SRAV|SRC|SRL|STEP|SYN|
                  THEN|TO|TOPGLOBAL|
                  UNDER|UNTIL|
                  WHILE|X0|X1|X2|X3|X4|X5|X6|X7|
                  X01|X12|X23|X34|X45|X56|X67|X70
                  :=|++|--|''|""|←
                  <CH|>CH|<=CH|>=CH|<=|>=

basic symbol  ::= char|reservedword|'|"|
```

2.2 Reserved Words and Comments

The reserved words given above cannot be used as identifiers within the PLASYD program. Their use will become clear in later sections. All reserved words other than '',:=,"",++and -- should be delimited before and after by either newlines, spaces or characters (char) other than letters and digits. The symbol := can be written as ←.

All characters (char) within the symbols [and] are ignored as comment, except within strings and character sequences where [and] are normal characters. For example:

```
X1 := 'AB[]' ;
[THIS IS A COMMENT]
```

is equivalent to writing:

```
X1 := 'AB[]' ;
```

3.

IDENTIFIERS, ACCUMULATORS AND CELLS

3.1 Syntax

```
xacc       ::= X0|X1|X2|X3|X4|X5|X6|X7
racc       ::= A1
xxacc      ::= X01|X12|X23|X34|X45|X56|X67|X70
rracc      ::= A12
modifer    ::= X1|X2|X3
identifier ::= letter|%|identifier letter|identifier digit|identifier %
```

3.2 Introduction

A PLASYD program consists of a series of declarations and statements. A statement corresponds to one or more computer instructions to be obeyed at object program run time. A declaration gives information to the computer mainly about storage layout and intercommunication.

Statements are basically of two kinds, control statements and assignment statements. Control statements can cause the normal sequence in which statements are to be obeyed to be altered. This can be used for example, to omit conditionally the execution of statements or to repeat a number of statements. Assignment statements are the means by which calculations are carried out. Their effect is to cause a value to be calculated and assigned to a variable. Variables are of two kinds called accumulators and cells. Accumulators correspond to the hardware accumulators in the 1900 series. Each cell corresponds to one or more words in the computer's main store.

Variables are divided into classes in PLASYD according to the type of data that they are used to hold and the mode of arithmetic that can be carried out on them. The various data types are dealt with later. Each variable has an identifier or name by which it is referenced in the program.

3.3 Accumulators

Accumulators have fixed identifiers assigned to them automatically by the system. These identifiers may not be used for any other purpose and are therefore similar to reserved words. They are:

(1) Integer Accumulators (xacc). The syntactic class xacc above defines the eight integer accumulators. They correspond to the 1900 series 24-bit fixed point accumulators. They can hold numbers, 24-bit patterns and addresses. Numbers can either be considered as signed numbers in the range -8388608 to 8388607 or unsigned numbers in the range 0 to 77777777 (octal). The bit pattern can be considered as four six-bit characters.

(2) Modifiers. The three fixed point accumulators X1, X2, X3 can be used as modifiers on the 1900 series and can therefore be used for equivalent special purposes within PLASYD.

(3) Real Accumulator (racc). The syntactic class racc above defines the 1900 floating point accumulator, A1, which is capable of holding a 48-bit floating point quantity.

(4) Long Integer Accumulator (xxacc). Each long integer accumulator is formed from two integer accumulators in an obvious way. For example, X23 is formed from X2 and X3. The long integer accumulator contains a 48-bit fixed point quantity. In general any alteration to the value of a long integer accumulator will destroy the values of its component integer accumulators, and vice versa. Consequently, altering the contents of X01 will alter X0 and X1 and will, therefore, destroy the values of X70 and X12 as well. It is, of course, possible to arrange computation on a long integer accumulator so that only the top or bottom half gets changed. For example, in X23, the least significant half is stored in X3.

(5) Long Real Accumulator (rracc). There is just one long real accumulator, A12, which is made up of the real accumulator, A1, and a mantissa extension for increasing the accuracy. Although available on the 1906A, the hardware is not available on many 1900's. Care should therefore be taken in using the long real accumulator if the program is to be transported. The long real accumulator can contain a 96-bit double length floating point number.

All the accumulators may be given additional names by using the synonym declaration to be defined later. For example:

```
ACC I SYN X2 ;
```

would allow fixed point accumulator X2 to also be called I.

3.4 Cells

Cells have no fixed names like accumulators. They may be assigned identifiers in the program by the relevant declarations. The form of identifiers is given by the syntactic class 'identifier' given above. Each identifier must be less than 32 characters in length and must not contain any layout characters such as space and newline as these will terminate the identifier. The identifiers defined for accumulators may not be used for other purposes. An identifier cannot coincide with a reserved word but a reserved word can be part of an identifier. Thus AND is not a legal identifier but ANDY is. Examples of possible cell identifiers are:

```
A   X   X8   %   A%B   AB%   %AB
AVERYLONGIDENTIFIER ANDY BEGI CHAR1
A1234567890B  PLASYD  ROOT1  ROOT2
```

If communication is desired between PLASYD and some other language then the global identifiers in PLASYD must conform to the limitations on identifiers in the other language.

The same identifier cannot be used for two different purposes in the same segment (a program consisting of a number of separated compiled segments could have local identifiers in different segments with the same name but different meaning).

4.

TYPES AND VALUES

4.1 Syntax

```
xvalue        ::= integer|MINUS integer|octalinteger|truncated|count|
                  charsequence|string
integer       ::= digit|integer digit
octaldigit    ::= 0|1|2|3|4|5|6|7
octalinteger  ::= #octaldigit|octalinteger octaldigit
truncated     ::= integer T integer|MINUS integer T integer
count         ::= integer CNT
charsequence  ::= 'ch'|'ch ch'|'ch ch ch'|'ch ch ch ch'
ch            ::= char|''|␣|"
string        ::= "stlist"
stlist        ::= st|stlist st
st            ::= char|" "|␣|'
rvalue        ::= real|MINUS real

real          ::= fraction|fraction & exponent|integer & exponent
fraction      ::= integer . digit|fraction digit
exponent      ::= integer|MINUS integer

xxvalue       ::= integer D|MINUS integer D|octalinteger D

rrvalue       ::= real L|MINUS real L
```

This section deals with the values that can be associated with cells and accumulators of the types INTEGER, REAL, LONG INTEGER and LONG REAL. PLASYD allows another type LOGICAL which is completely equivalent with INTEGER. The two types may be used interchangeably. However, LOGICAL is usually used in cases where no arithmetic is being carried out. For example, the individual bits of a variable might be used as a set of markers in which case it would be sensible to define it as LOGICAL. The compiler makes no distinction between cells of integer and logical type. In general, integer values are always associated with integer variables and so on. However, it is possible to associate integer or real values with long integer and long real variables respectively.

4.3 Integer Values

The 24-bit 1900 storage word which makes up an integer cell, or an integer accumulator, may be considered to hold:

(1) 24-bits considered as separate values of 0 or 1

(2) 4 six-bit characters

(3) an octal number in the range of 0 to 77777777 octal

(4) a decimal number in the range -8388608 to 8388607

Integer values (xvalue) are defined in a PLASYD program as:

(a) Integer. A positive or negative decimal integer in the range given above. Positive integers are written without a sign while negative integers are preceded by the reserved word MINUS. The individual digits making up an integer may not be separated by spaces. The integer number is held internally in 24-bit two's complement binary form. Examples are:

```
3671   29   0   4052   8388607
MINUS 6   MINUS 2732  MINUS 8388608
```

(b) Octal Integer. This consists of a sequence between 1 and 8 octal digits preceded by the symbol # . Spaces or other layout characters may occur between the symbol # and the first digit but nowhere else. The number will be stored in the right most octal positions of the word. Unspecified octal positions to the left are set to zero. The first two of the following examples therefore have the same value:

```
#00000770   # 770    #273    # 27
      #12345670     #1      # 00000000
```

(c) Truncated Numbers. These are mainly used for setting up addresses. There must be no spaces between the T and numbers on either side. If the number is considered as aTb then its value is the rightmost b bits of the number a. It will normally only be used with b = 12, 15 or 22 although other values are possible. If a is negative it can be used to set up negative addresses. Some examples with corresponding octal values are:

12T15	#00000014
1976215T22	#07423627
MINUS 12T15	#00077764
8000000T12	#00001000
MINUS 8000000T12	#00007000

(d) Count. This is an integer number in the range 0 to 511 which is to be held in the most significant 9 bits of the word with the remaining digits set to zero. No spaces are allowed between the number and the word CNT. Some examples with equivalent octal values are:

257CNT	#40100000
511CNT	#77700000
2CNT	#00200000

(e) Character Sequence. The character sequence (charsequence) consists of from one to four characters of the 1900 character set enclosed within the single quote marks ('). The single quote character itself is represented in the sequence by two consecutive single quotes which count as one of the four possible characters. A space can be used as one of the six-bit characters. The characters are held in the right most six-bit character positions of the integer variable with any unspecified character positions being set to zero. The six-bit values associated with the 1900 character set are given in Appendix 1. Some examples with corresponding octal values are as follows:

'%'	#00000025
'+23'	#00330203
'ABCD'	#41424344
'''A'''	#00274127
'2 3'	#00022003
'2 3 '	#02200320
''	#00000000

Note that the null string is allowed by the compiler.

(f) String. A string consists of one to four characters enclosed within the double quote marks. The double quote character itself is represented by two consecutive double quote marks. Unlike character sequences, the characters in a string are held in the left-most six-bit character positions of the integer variable with any unspecified character positions being set to the space character. Some examples with corresponding octal values are as follows:

" %"	#25202020
" "	#20202020
" "	#20202020
" A B C D"	#41424344

4.4 Real Values

A real variable consists of either the floating point accumulator A1 or a real cell consisting of two consecutive 24-bit words in main store. A real variable can hold a (generally normalised) floating point binary number in standard 1900 format. The numbers are held with an accuracy of about 11 decimal digits in the range -5.6×10^{76} to 5.6×10^{76}. The smallest number other than zero which can be represented is about 10^{-77}. Real values (rvalue) are defined in a PLASYD program as:

(a) Fractional Number. A fractional number (fraction) consists of an integer number followed by a decimal point and one or more integers. There should be no spaces between the integers and decimal point. The numbers may be signed. For example:

0.13261 321.67 215.0
MINUS 136.0 MINUS 0.141579

(b) Fractional Number with Exponent. Any fractional number may be followed by an & and an integer number. The integer is taken to be a power of ten by which the fractional number is multiplied. There must be no spaces between the fraction and the & symbol. Spaces can occur between the & and the exponent but are not necessary. The exponent may be signed. For example:

1.4&20 1.4&MINUS 20 1.4& MINUS 20
MINUS 0.5613&3 1.414& MINUS 6

(c) Integer Number with Exponent. An integer number followed by an exponent will also be considered a real number. Again no space is allowed between the integer and &. Spaces can occur between the & and exponent. For example:

3&6 7&MINUS 6 3& 5 43& MINUS 12
41107&25 MINUS 47& MINUS 3

4.5 Long Integer Values

A long integer variable consists of either two adjacent integer accumulators or two adjacent 24-bit words in main store. Long integers must be within the range -140737488355328 and 140737488355327. As an octal value, it may consist of up to sixteen octal digits. Long integer values (xxvalue) are defined in a PLASYD program as:

(a) Integer. A positive or negative decimal integer in the range given above and followed by the letter D. Neither the individual digits nor the letter D and preceding digit should be separated by spaces. For example:

27D MINUS 37221567854D

(b) Octal Integer. This consists of a sequence of 1 to 16 octal digits preceded by the symbol # . Spaces or other layout characters may occur between the # symbol and the first digit but not elsewhere. The number will be stored in the right-most octal positions of the word. Unspecified octal positions to the left are set to zero. The following two examples represent the same value:

#76767676767676D #0076767676767676D

4.6 Long Real Values

A long real variable can be manipulated on the 1906A and some 1900's when the necessary hardware is available. A long real variable consists of either the floating point accumulator and mantissa extension or, alternatively, four consecutive 24-bit words in main store. A long real variable can hold a (generally normalised) floating point binary number.

Long real values (rrvalue) are defined in a PLASYD program as:

(a) Real Number. Any of the methods (Fractional Number, Fractional Number with Exponent, Integer Number with Exponent) of defining a real number can be used for defining a long real number by following the number immediately with the letter L. No space should exist between the last digit and the letter L. Otherwise the constraints are similar to those for real values except that the range is as above. For example:

```
0.13261L                3.17854758432L
MINUS 1785.34L
1.4&80L          1.75& MINUS 20L
MINUS 3.1758477285&60L
3185642&50L
MINUS 777554& MINUS 40L
```

ADDRESSES AND STORAGE ALLOCATION ON THE 1900 RANGE

5.1 Upper and Lower Storage

Data storage on the 1900 range is complicated by the architecture which allows only the first 4096 locations of store to be accessed directly. The standard 1900 instructions have a 12-bit address field and a 2-bit modifier field. The modifier field indicates whether the address is unmodified or modified by the contents of X1, X2 or X3. To access a store location, other than the first 4096, requires the most significant part of the address to be stored in a modifier. For example, if the modifier field contains the value i ($\neq 0$) and the address field contains α then the actual address accessed is α + (Xi) where the brackets denote 'contents of'. The first 4096 locations are called 'lower' storage on the 1900 series while the remainder are called 'upper' storage. The standard method of accessing upper storage in 1900 software is to place in lower storage the base addresses for domains of upper storage not greater than 4096 locations in length. Assuming the base address is not already in a modifier, accessing an address in upper storage is achieved by loading the base address of the upper store domain into a modifier and placing the displacement in the modified instruction itself which does the accessing. To keep the number of lower store locations used in this way to a minimum, it is usual to divide upper storage into domains as close to 4096 words in length as possible.

5.2 Type Declarations

PLASYD declarations are defined in full in a later section. It is necessary, however, to define some basic declarations before individual PLASYD statements can be defined. The simplest form of the PLASYD declaration is:

```
celldec    ::= xcelldec|rcelldec|xxcelldec|rrcelldec
αcelldec   ::= αtype αitemlist;  {α =x|r|xx|rr}

xtype      ::= INTEGER | LOGICAL
rtype      ::= REAL
xxtype     ::= LONG INTEGER
rrtype     ::= LONG REAL
αitemlist  ::= αitem  |αitemlist, αitem { α= x|r|xx|rr}
αitem      ::= αidentifier | αidentifier (integer) {α=x|r|xx|rr}
αidentifier::= identifier {α=x|r|xx|rr}
```

Individual declarations are separated by semicolons.

The two forms of item are:

(a) identifier : this reserves sufficient storage for a single cell of the type being declared. For example:

```
INTEGER          A, B, C;
```

```
REAL            D, E, F;
```

allocates single 1900 words to A, B and C and two words each to D, E and F.

(b) identifier (integer) : this reserves sufficient storage for an array of cells of the type being declared. For example:

```
INTEGER         A(20),  B(20);
LONG REAL       D(50);
```

allocates 20 words to the arrays A and B and 200 words to the array D which consists of 50 LONG REAL cells each of four words in length.

5.3 Lower and Base Declarations

To specify that a set of declarations define cells to be placed in lower storage, they are placed between the basic symbols LOWER and LOWEND. All storage cells defined as lower can be accessed directly. For example:

```
LOWER
INTEGER     A, B;
REAL        C, D(10), E;
LOWEND;
```

defines a set of lower cells A, B, C, D(10) and E.

Cells that are not in lower storage cannot be directly accessed by the 1900. To enable such cells to be used, the upper storage may be divided into arbitrary sections called domains which must not exceed 4096 words. An upper cell can then be accessed by specifying the address of the first word of its domain (called the base of the cell) and the number of words (less than 4096) between this base and the cell required. This number is called the displacement of the cell in PLASYD. Declaration in PLASYD will define upper storage for cells unless lower storage is specified explicitly by the LOWER declaration given above. Upper storage is divided up into domains by the base declaration written:

```
BASE;
```

which starts a new domain. The first upper declaration will cause a new domain to be set up and successively declared items will be assigned store locations in that domain in order until the appearance of a base declaration, or until the next declaration would cause the size of the domain to exceed 4096 words. A new domain will then be started in the next available word. Whenever a new domain is started, one word of lower storage is used to store the address of the first word of the domain.

In PLASYD, the symbol £ is used to specify the base address of a domain. Thus £A is the base address of the domain containing the cell A. The symbol $ is used to specify the displacement of a cell from its base address. Thus $A defines the displacement of the cell A from the base of the domain. The actual address of A is denoted by @A so that @A =

£A + $A. For lower cells $A = @A and £A = 0

For example:

```
INTEGER A, B, C;
REAL D;
BASE;
REAL G, H;
INTEGER I, J, K;
```

defines two upper domains. The first is 5 words in length while the second is 7 (real cells take two words). The values of $D, $H and $K are 3, 2 and 6 respectively.

5.4 Identifier Classes

```
loweraidentifier   ::= identifier       {a = x|r|xx|rr}
upperaidentifier   ::= identifier       {a = x|r|xx|rr}

loweridentifier    ::= lowerxidentifier|lowerridentifier|
                       lowerxxidentifier|lowerrridentifier

upperidentifier    ::= upperxidentifier|upperridentifier|
                       upperxxidentifier|upperrridentifier
```

Identifiers are divided up into the classes, as defined above, depending on the cell's type defined in the declaration and whether it was allocated upper or lower storage.

6.

SIMPLE CELL DESIGNATION

6.1 Syntax

```
simpleαcell ::= lowerαidentifier|lowerαidentifier(integer)|
                lowerαidentifier(-integer)|(integer)|(modifier)|
                (modifier + integer)|αidentifier(modifier)|
                αidentifier(modifier+integer)|
                αidentifier(modifier-integer){α=x|r|xx|rr}
```

6.2 Addresses

This section describes the simpler formats for accessing the contents of storage cells. The syntax above defines an address of a storage location. The assignment statements to be defined later either deposit a new value into this location or access its contents. If the value of the 'integer' used in the syntax is I, the contents of the 'modifier' is X, and the displacement of the identifier is $ then the address specified in each case is:

lowerαidentifier	$
lowerαidentifier(integer)	$ + I
lowerαidentifier(-integer)	$ - I
(integer)	I
(modifier)	X
(modifier + integer)	X + I
αidentifier(modifier)	$ + X
αidentifier(modifier + integer)	$ + X + I
αidentifier(modifier - integer)	$ + X - I

The addresses calculated must have $0 \leq \$-I, I, \$+I < 4096$. For upper identifiers, it is usual for the modifier to contain the base address of the domain.

6.3 Specific Points

Specific points concerning the various types of cell designation can be illustrated using the following example. These declarations are assumed for all examples in the sections following.

```
LOWER
REAL LRX, LRA(20), LRY;
INTEGER LIX,LIA(10);
LONG INTEGER LIIX, LIIA(4);
LONG REAL LRRX, LRRA(2);
LOWEND;

BASE;
INTEGER UIA(12), UIX;
REAL URX, URA(10);
```

(a) lowerαidentifier : this can be referenced by specifying its identifier alone. The first element of an array is considered to be a simple cell as well. Therefore LRA specifies the address of the first word of the array LRA. Thus legal cell designations are:

LRX, LRA, LRY, LIX, LIA while both UIA and UIX are not as they are upper identifiers

(b) lowerαidentifier(integer), lowerαidentifier(-integer) : this designation is primarily used for accessing the individual elements of an array. Thus LIA(0), LIA(1), ... LIA(9) are the designations of the elements of the array LIA. Note that the elements are LIA(0) to LIA(9) and not LIA(1) to LIA(10). In the case of a real array, each element takes up two words of storage. Consequently the elements of the real array have designations LRA(0), LRA(2), LRA(4), ... LRA(38).

There is no check made on the size of the integer to ensure that the accessing is not outside the bounds of an array nor is there any reason why scalar identifiers should not be accessed in this way. Thus both LRX and LRX(0) designate the scalar LRX. Similarly LRX(2) is equivalent to LRA(0) which is equivalent to LRY(-40). All refer to the second real quantity, LRA(0), in the lower declaration. The address obtained by adding the fixed index to the identifier must always be in the range 0 to 4095.

The equivalencing of scalars and arrays in this way allows the programmer to declare effectively one array and to name individual elements of it. Thus LRX could be considered as a 22 element array whose second element is called LRA and last element LRY.

(c) (integer), (modifier), (modifier+integer) : the quantity in brackets defines the actual address of the cell being designated. Thus (30) is the thirtieth word of the core store, (X1) is the cell whose address is currently stored in X1 and (X1+3) is the cell 3 further on from the previous one. It is possible to access the integer accumulators in this way as they are also the first cells of the 1900 store. Thus (1) designates cell 1 which is also X1. In the case of a cell designated using a modifier, the address will be truncated to the lower 15 or 22 bits depending on the addressing mode in use. The value of the integer must be in the range 0 to 4095 but the value of the expression if a modifier is used may be any possible store location of the machine. By loading the modifier with an address this provides one method of accessing upper storage cells. For example if X1 contains the base address of the upper storage domain containing UIA and UIX (£UIA or @UIA) then (X1) designates UIA(0) and (X1+12) designates UIX.

(d) αidentifier(modifier), αidentifier (modifier+integer), αidentifier (modifier-integer) : for lower identifiers, this method of designation is primarily used for accessing individual elements of arrays. The modifier and integer between them define the number of words that the address is displaced from the address of the identifier. For example, if X1 contains 3 then LIA(X1) is equivalent to LIA(3) and LIA(X1+2) is equivalent to LIA(5). The address defined in this way

must be a lower address between 0 and 4095. One other limitation is that the part of the address other than that in the modifier should be positive. Therefore LRX(X1-8000) is not allowed even if X1 contains 8000.

For upper identifiers, this provides the main method of designation. If the modifier contains the address of the base of the domain (in our example X1 could contain £UIA) then the remainder of the address specifies the element of the domain. For example, if X1 contains £UIA, then UIA(X1) defines the first element of the array UIA and UIX(X1) defines the scalar UIX. This is because the identifier itself specifies the displacement from the base address. Similarly UIA(X1+12) also defines UIX. Note that UIA(3) is meaningless and upper identifiers must always be accessed using a modifier. One limitation is that the part of the address other than the modifier (that is the displacement specified by the identifier with the possible addition or subtraction of the integer) must be in the range 0 to 4095. Thus UIA(X1-1) or UIX(X1-13) are both illegal even if the current value of X1 is £UIA+10. Although the modifier usually contains the base of the domain, this is not essential and there is no reason why £UIA+1 should not be stored in X1. In this case UIA (X1) defines the element UIA(1) while UIA(X1+11) defines UIX.

6.4 Instructions Generated

PLASYD will normally generate a single 1906A order for each operator /operand pair when operands (cells) are defined as simplexcell or simplercell. For long integers and long reals, more than one instruction is often generated. A more complex mode of cell designation involving the SMO order will be described later. In general, the user can perform most of the functions he requires without needing the SMO designation.

7.

ASSIGNMENT STATEMENTS

7.1 Types of Assignment Statement

Assignment statements are the means by which storage cells and accumulators are given values. The form of a value assignment to a variable can vary from the simple constants defined in Section 4 to complicated expressions involving the current value of other variables. There is a distinction between cell and accumulator assignments. As the 1900 order code has few machine orders for doing arithmetic in cells, cell assignments are in general much more restrictive. In most cases the calculations are carried out using accumulators and the resulting value is then assigned to a storage cell. In general cell assignment statements are less efficient than accumulator assignments due to the number of store accesses involved.

Assignment statements may run over several lines if necessary, or several short statements may be placed on one line. Statements, like declarations are separated by semicolons. Assignments, in general, only alter the value of the accumulator or value to the left of the assignment operator. Cases where this is not so will be mentioned specifically. Care should be taken about where statements are split between lines. In particular, a wrong code may be generated if the split occurs immediately after the := symbol.

7.2 SMO Operands

The operands used in assignments can be the simple cell designators defined in the previous section or the more complex SMO designators. The syntax in the following chapters will include this more complex type which will not be defined in full until a later Section.

7.3 Overflow and Carry

The 1900 series computers do have a number of one-bit registers which are not directly addressable but do affect the operation of some instructions. The OVERFLOW or V register is set when the result of carrying out integer arithmetic in single length exceeds 23 bits plus sign. Once set, V will remain set until cleared. In the case of single length floating point operations, if the accumulator A1 has its exponent go out of range then the floating point overflow register FOVR will be set. Instructions exist on the 1900 range for testing and clearing overflow bits.

The CARRY register is a special one-bit register associated with multi length integer working. The integer arithmetic instructions on the 1900 will, in general, add the CARRY bit to the value of the operand before executing the instruction. However, as the basic instructions reset CARRY to zero, this effect can normally be ignored. When multi-length working is being carried out, special arithmetic instructions allow CARRY to be propagated though the multi-length integer.

8.
INTEGER ACCUMULATOR ASSIGNMENTS

8.1 Syntax

```
xassignment     ::= xacc:=xprimary |xacc:=address |xasgn + address |
                    xasgn-wordaddress |xacc:=monadic xprimary |
                    xacc:=CNT coperand |xassignment xarithmetic
                    xprimary |xassignment logical xprimary |xassignment
                    shift coperand
xasgn           ::= X0:=X0 |X1:=X1 |X2:=X2 |X3:=X3 |X4:=X4 |X5:=X5 |X6:=X6 |X7:=X7
xprimary        ::= xvalue |xacc |xcell
αcell           ::= simpleαcell |smoαcell      {α =x| r| xx| rr}
monadic         ::= NEG |EX |CH |CARRY |NEG CARRY
coperand        ::= integer |octalinteger |modifier |simplexcell
xarithmetic     ::= + |++ |- |-- |* |/
logical         ::= AND |OR |ER
shift           ::= SLL |SRL |SLA |SRA |SRAV |SLC |SRC
address         ::= wordaddress |CHAR integer |CHAR integer OF wordaddress |
                    @ identifier
wordaddress     ::= $ extendedcell |@ extendedcell |£ identifier
extendedcell    ::= cell |upperidentifier  (integer) |upperidentifier
                    (-integer)
cell            ::= xcell |rcell |xxcell |rrcell
```

8.2 Basic Integer Accumulator Assignment

xacc:= xprimary

The basic integer accumulator assignment sets the defined accumulator equal to the value of the primary specified. This is divided into three kinds:-

(1) Assigning a Value: one of the integer values defined in Section 4.3 is assigned to the integer accumulator specified. Examples are:

```
X1:=3671; X2:=MINUS 6; X3:= # 770; X4:=12T15; X5:=MINUS 80000T12;
X6:=257CNT; X7:='ABCD'; X3:= "A";
```

Assignment of an integer value which can be stored in 12 bits takes no extra storage. Integer values outside this range will cause the constant's value to be stored in lower and accessed from there. Several uses of the same value in one segment will result in only one lower storage cell being set aside.

(2) Assigning an Accumulator: in the case of integer accumulators it is possible to transfer values from one accumulator to another.

For example:

```
X1:=X3; X2:=X2; X5:=X7;
```

The code generated for X1:=X3 is equivalent to X1:=(3). In the case of an assignment to the same accumulator, this usually does not generate any code. However, code is generated if the assignment is split between two lines. Thus:

```
X2:=X2;
X2:=
X2;
```

would compile code in the second assignment but not in the first.

(3) Assigning a Cell: any of the simple cell designators defined in Section 6 can be used to assign the contents of the cell to an integer accumulator. For example:

```
X1:=LIX; X2:=LIA(3); X3:=LIA(-1); X4:=(3);
X5:=(X2); X6:=(X1+3); X7:=UIA(X1); X0:=UIA(X1+3);
X7:=UIX(X1-12); X6:=LIX(X1+2); X5:=LIA(X1-3).
```

When the type of cell is known, it must be integer.

8.3 Addresses

Integer cells and accumulators may be assigned values which are addresses. The use of variables with such values has been described in the section on cell designations. An address takes one of the following forms:

(a) Displacement ($ extendedcell): the address is defined as the displacement of the cell which is the number of words that the cell designator is displaced from the base of its domain. In the case of a lower cell, this is the address of the cell itself. For an upper identifier, the address is the base of its domain. If the cell designator does not consist of a simple identifier but contains a modifier and perhaps an integer then these will be added to the address in the normal way. Values of such addresses may be greater than 4095. In addition to normal cell designators, it is possible to specify an upper identifier with a fixed index. If the integer accumulator X1 contains the value X then some example addresses with their values are:

address	value
$LRX	@LRX
$LRA(4)	@LRX+5
$LRX(X1+3)	@LRX+X+3
$UIA	0
$UIA(2)	2
$UIX	12
$UIX(-2)	10

(b) Base (£identifier): the address is defined as the base of the domain containing the identifier. In the case of lower cells it is identically zero. Note that only identifiers are allowed and £LRX(3) is illegal. Some example addresses with values are:

address	value
£LRX	0
£LRA	0
£LIA	0
£UIA	@UIA
£UIX	@UIA

(c) Complete address (@extended cell): this is the complete address of a cell in words. It is equal to the displacement of a cell added to the base address. Thus @Y is always identically equal to £Y+$Y. Some examples are:

address	value
@LRX	@LRX
@LRX(X1+3)	@LRX+X+3
@UIA	@UIA
@UIA(2)	@UIA+2
@UIX	@UIA+12
@UIX(-2)	@UIA+10

As well as defining cell addresses in this way, this format can also be used to specify the address of a PLASYD label or procedure. These will be defined later.

(d) Character address. A 1900 word can be thought of as containing four 6-bit characters. In some 1900 instructions, the contents of a modifier or accumulator can be taken as an address with the top two bits defining the character position (0,1,2 or 3) and the remaining 22 bits defining the word address. In the simpler form, it defines a character address only, while in the more complex form it defines the word address as well. For example:

address	character value	word
CHAR 3	3	0
CHAR 2 of $UIX(-2)	2	10
CHAR 1 of @UIA(2)	1	@UIA+2
CHAR 6 of @UIA	2	@UIA+1

The compilation of addresses with the character position defined greater or equal to 4 tends to be inefficient and is not therefore advisable.

8.4 Address Assignments

```
xacc:=address
xasgn+address
xasgn-wordaddress
```

The simplest address assignment sets an integer accumulator equal to one of the addresses specified above. For example:

```
X1:=$LRX; X2:=£UIA; X3:=@UIX(-2):
X4:=CHAR 3; X5:=CHAR 1 OF @UIA(2).
```

The more complex forms allow an address to be added to the contents of an integer accumulator or a word address to be subtracted from the contents of the accumulator. Note that a character address cannot be subtracted from the accumulator. Some examples are:

```
X1:=X1+CHAR 3; X2:=X2+@UIX(-2);
X3:=X3+CHAR 1 OF @UIA(2);
X4:=X4-@UIX(-2); X5:=X5-£UIX;
```

It should be remembered that the character position of an address will normally cause the two most significant bits of the integer accumulator to be set. If the accumulator contains a large value then care should be taken to ensure that an addition does not cause overflow into these two bits.

Some comments on the code generated for address assignment are:

(a) Addition of just the CHAR part of an address will be done by repeated use of the BCHX instructions. Thus X1=X1+CHAR 7 will generate seven instructions.

(b) To assign, add or subtract an identifier base always takes one instruction.

(c) To assign, add or subtract a word displacement takes one instruction at least and more than one if a SMO index is used.

(d) To assign, add or subtract a cell address takes at least one instruction for a lower identifier and at least two for an upper identifier.

(e) If the address of the identifier used is not yet known, ie it is a label or a procedure which has not yet been declared, and an incremented form of address is used, then incorrect code will be generated. No error message or warning will be given.

(f) If the address of a word in upper storage is subtracted from an accumulator then incorrect code will normally be generated. No error message or warning will be given.

8.5 Monadic Operators

The primary on the right of the assignment may be preceded by a monadic operator. The effect on the assignment is as follows:

(a) NEG: the integer accumulator is assigned the negative of the primary value. For small constants (less than 12 bits) no lower storage is used and the code generated is faster than that using MINUS. Thus X1:=NEG 3 is preferred to X1:=MINUS 3.

(b) EX: the lower 9 bits of the primary specified are assigned to the integer accumulator. Constants will require lower storage. Consequently, X4:=EX 3 uses lower storage but X3:=3; X4:=EX X3 does not.

(c) CH: the result of the CH operator depends on whether the primary is defined using a modifier or not. In the unmodified form, the CH operator sets the specified accumulator equal to the lower 6 bits of the primary. If the primary is defined using a modifier, then the address is taken as a character address and the lower 6 bits of the integer accumulator are set equal to this character position. For example, if LIX contains the constant 'ABCD' then the value of X1 will be:

```
X1:=CH 33;                    'A'  that is  #00000041
X1:=CH LIX;                   'D'  that is  #00000044
X2:=0;
X1:=CH LIX(X2);               'A'  that is  #00000041
X2:=CHAR 2;
X1:=CH LIX(X2);               'C'  that is  #00000043
```

(d) CARRY: the integer accumulator is assigned the value of the primary without its most significant bit. The CARRY register is set if the most significant bit of the primary is set. As in all 1900 integer arithmetic, the value of the primary will be incremented by the value of the CARRY bit. For example if LIX contains 37777777 and CARRY is set, then X1:=CARRY LIX; will set X1 to 0 and will propagate the CARRY leaving the CARRY register set.

(e) NEG CARRY: the value of the primary is first negated and the operator CARRY is performed on the result.

The monadic operators do not cause additional machine instructions to be obeyed. In the case of CARRY and NEG, small constants do not require lower storage but with EX and CH they do.

8.6 Counter Assignments

```
xacc:=CNT coperand
```

The integer accumulators can be set up to contain a count in the bits 0 to 8 and a modifier in the remainder of the accumulator. Special branch instructions in the order code allow the modifier to be

incremented, the counter to be decremented and the branch to occur if the count is non-zero. As bits 0 and 1 are used to define a character position in a word, it is usual for the count to be set less than or equal to 127.

The effect of the counter assignment is to set the top 9 bits of the accumulator specified to the value of the 'coperand'. The remaining bits are set to zero. If an integer constant is used as the 'coperand' then its value must be less than 512. Similarly, an octal constant must be less than #1000. If the coperand is a cell designation then this will generate an additional order (which is a SMO). Typical examples are:

```
X1:=CNT 127; X1:=CNT #177;
X1:=CNT LIX; X1:=CNT LIA(X2);
X4:=CNT X2; X4:=CNT (X2).
```

It is more efficient to load a count by:

```
X1:=CNT 127;
```

than using:

```
X1:=127 CNT;
```

The latter generates a constant in lower storage.

8.7 General Integer Accumulator Assignment

```
xassignment xarithmetic xprimary
xassignment logical     xprimary
xassignment shift       coperand
```

So far, we have considered the simple examples of assigning a value to an integer accumulator. A general assignment statement can assign a value and then cause various arithmetic or logical operations to be performed on the contents of the accumulator. These operations are performed in the order specified within the statement, that is, strictly left to right and not in the normal arithmetical hierarchy order. For example:

```
X1:=X2+LIX*LIA(2)-X1;
```

is equivalent to:

```
X1:=X2; X1:=X1+LIX; X1:=X1*LIA(2); X1:=X1-X1;
```

If the same accumulator appears before and after the := on the same line of code then no code is generated for that operation. Note that the last operation in this example always forces the result to be zero! If the instruction is split between two lines immediately after := then the accumulator specified is loaded from store. This may be

of use in unsetting the CARRY register.
The three classes of operators are:

(1) Arithmetic: the possible operators are +, -, *, /, ++, --.

(a) +, - : these have their standard meanings.

(b) *,/ : these two operations do have the side effect of destroying the contents of a neighbouring accumulator (the neighbours of X7 are X6 and X0). In the case of multiplication, the next accumulator is set to zero. For division, the unrounded result is placed in the specified accumulator and the remainder in the preceding one. Thus:
X6:=X6*3 destroys X7 while
X6:=X6/3 destroys X5.

(c) ++,-- : these two operators are similar to + and - except that the top bit of the result is always reset to 0 and if CARRY has taken place then the CARRY register is set. The operators are useful when doing triple-length and multi-length working. For example, if a triple-length number is stored in LIA(1), LIA(2) and LIA(3) then the number can be loaded into X4, X5 and X6 and the constant 3 added to it by:

```
X6:=LIA(3)++3;
X5:=CARRY LIA(2);
X4:=LIA(1);
```

The number in X4, X5, X6 can have the number in LIA subtracted from it by:

```
X6:=X6--LIA(3);
X5:=X5--LIA(2);
X4:=X4-LIA(1);
```

(2) Logical ; the possible operators are AND, OR and ER which specify 'and', 'or' and 'non-equivalence' respectively. The accumulators and operators specified are taken as 24 bit quantities with the operations performed on corresponding bits.

(3) Shift : the possible operators are: SLL, SRL, SLA, SRA, SRAV, SLC, SRC. These are only allowed with the coperands specified in the Count Assignments. The meanings of the operators are:

SLL : Shift left logical. The top bits are lost and spaces created at the least significant end are filled with zeros.

SRL : Shift right logical. The least significant bits are lost and spaces created at the most significant end are filled with zeros.

SLA : Shift left arithmetic. The accumulator is regarded as a 24-bit signed number. The top bits are lost and spaces created at the least significant end are filled with zeros. The overflow register V is set if the

value of the most significant bit undergoes any change during the shifting. It can therefore be used to multiply positive and negative numbers by powers of 2.

SRA : Shift right arithmetic: Similar to SLA. The least significant bits are lost. In the case of SRA the sign bit is propagated at the most significant end of the word. The result is therefore still arithmetically correct and is equivalent to dividing by a power of 2.

SRAV: Special shift right arithmetic: The result is identical to SRA if the overflow register V is clear. If V is set, the inverse of the most significant bit is propagated at the most significant end of the word. Bits shifted beyond the right hand end are lost except that the value of the final bit so shifted is added back into the result to round it off. The SRAV allows the original number to be recovered after overflow. For example
X1:=X1 SLA 1 SRAV 1;
leaves X1 unchanged even if overflow took place. To take the average of two numbers in X1 and X2:-
X1:=X1+X2 SRAV 1 which works even if the sum overflows.

SLC : Shift left circular: The 24 bits are regarded as a pattern which is circulated to the left the number of places specified. As a bit leaves the most significant end it re-appears in the space at the least significant end.

SRC : Shift right circular: Similar to SLC but the pattern is circulated to the right. As a bit leaves the least significant end it re-appears in the space at the most significant end.

If X1 contains #76541234 then the results of the following operations are:

```
X1:=X1 SLL 3;     65412340
X1:=X1 SRL 3;     07654123
X1:=X1 SLA 3;     65412340        V is not set
X1:=X1 SRA 3;     77654124
X1:=X1 SLC 3;     65412347
X1:=X1 SRC 3;     47654123
```

As before, the possible operands are restricted to coperands. Some examples are:

```
X1:=X1 SRL 4; X2:=X2 SLL #11
X3:=LIX SLA X2; X4:=7 SLL LIA(X2-1);
```

From the syntax, it can be seen that there is complete freedom to mix operator/operand pairs of the types arithmetic, logical and shift. Thus:

```
X1:=X1 SRL 4 AND #77 OR CNT 127 + 4 - LIX;
```

9.

REAL ACCUMULATOR ASSIGNMENTS

9.1 Syntax

```
rassignment ::= A1:=rprimary|A1:=A1|A1:=NEG rprimary|
                rassignment FROM rprimary|rassignment UNDER rprimary|
                rassignment rarithmetic rprimary

rprimary    ::= rvalue|rcell

rarithmetic ::= +|-|*|/
```

9.2 Basic Real Accumulator Assignment

```
A1:=rprimary
A1:=A1
```

The basic real accumulator assignment sets the floating point accumulator (A1) equal to the value of the primary specified. The statement A1:=A1 does not generate any instructions and is really only of use in the construction of more complex assignments. There are basically two kinds of basic assignments:

(1) Assigning a value: one of the real values defined in Section 4.4 is assigned to the floating point accumulator. Examples are:

```
A1:=0.13261; A1:=321.67; A1:=MINUS 0.141579;
A1:=1.4&20; A1:=MINUS 0.5613 & MINUS 6;
A1:=3&6; A1:=43.& MINUS 12;
```

The type of accumulator and the primary must be identical. Thus A1:=0 is illegal and must be written A1:=0.0. Assignment of the value 0.0 to the real accumulator is more efficient than any other real assignment and does not require the number 0.0 to be stored. Assignment of all values other than 0.0 will cause two 24 bit words for each value to be assigned in lower storage to hold the value. Several uses of the same number in one segment will result in only one pair of lower storage cells being set aside.

(2) Assigning a cell: any of the simple cell designators defined in Section 6 can be used to assign the contents of the cell to the real accumulator. The type of the cell must be real if it is known. In the case where the type is indeterminate, it is assumed to be correct. Some examples:

```
A1:=LRX; A1:=LRA(4); A1:=LRY(-40);
A1:=(X1); A1:=(1); A1:=URA(X1+6);
```

9.3 Monadic Operators

The primary on the right of the assignment may be preceded by a monadic operator. The effect on the assignment is as follows:

(a) NEG: the real accumulator is assigned the negative of the primary value. This requires an extra instruction to be obeyed. For this reason it should not normally be used with values as primaries. That is use A1:=MINUS 3.0 rather than A1:=NEG 3.0. In the latter case 3.0 is stored in lower and is negated to A1 by two instructions, while in the second MINUS 3.0 is stored in lower and is loaded into A1 by one instruction.

9.4 General Real Accumulator Assignments

```
rassignment FROM rprimary
rassignment UNDER rprimary
rassignment rarithmetic rprimary
```

A general real assignment statement extends the basic assignment statement. The first part of the statement equivalent to the basic statement assigns a value to the real accumulator. The remaining terms of the statement then cause various arithmetic operations to be performed on the contents of the real accumulator. These operations are performed strictly from left to right. For example:

```
A1:=LRY+LRX*LRA(2);
```

is equivalent to

```
A1:=LRY; A1:=A1+LRX; A1:=A1*LRA(2);
```

A statement where the real accumulator appears before and after the := (on the same line) does not cause any code to be generated for that operation. If the statement is split between lines immediately after the := symbol, then incorrect code is generated. The possible operations are:

(1) +, -, *,/ : these have their usual meaning. Real arithmetic is rounded on the 1900 range. As A1 is not a primary, the statement A1:=LRX+A1 is illegal.

(2) FROM : this is the reverse subtraction operator. For example, A1:= LRX FROM LRY is equivalent to A1:=LRY-LRX. It can be useful in the form A1:=A1 FROM LRX as A1:=LRX-A1 is illegal.

(3) UNDER : this is the reverse divide operator. For example, A1:=LRX UNDER LRY is equivalent to A1:=LRY/LRX. Again it is useful when the first operand is the real accumulator.

Examples of real accumulator assignments are;

```
A1;= LRX * LRY FROM LRA(2) UNDER LRA(4) - URA(X1+2)/3.0;
A1;= NEG 3.145/LRX+LRY-3.2&7;
```

10.

LONG ACCUMULATOR ASSIGNMENTS

10.1 Syntax

```
xxassignment ::=xxacc:=xxprimary|xxacc:=NEG xxprimary|
               xxassignment xxarithmetic xxprimary|xxacc:=xprimary|
               xxassignment shift coperand

xxprimary    ::=xxvalue|xxacc|xxcell

xxarithmetic ::=+|-|++|--

rrassignment ::=A12:=rrprimary|A12:=NEG rrprimary|A12:=A12|
               rrassignment rrarithmetic rrprimary

rrprimary    ::=rrvalue|rrcell

rrarithmetic ::=+|-|*|/
```

10.2 Basic Long Integer Accumulator Assignments

Accumulator assignments for long integers are very similar to those for integers. The basic form allows a long integer value, accumulator or cell to be assigned to a long integer accumulator. Long integer values have been defined in Section 4.5. The code generated usually consists of two instructions per operation (one operation on the top accumulator of the pair while the other operates on the bottom).

Some typical examples are:

```
X12:=273D; X23:= #7777777777D; X34:=LIIX;
X45:=MINUS 32679D; X56:=LIIA(0); X67:=LIIA(X1+2);
```

Even for small integer constants, a pair of lower storage cells will be used to hold the value. Consequently X12:=273D; is less efficient in storage than X1:=0; X2:=273;

The monadic operator NEG can be used to assign the negative of the primary value to the accumulator. For example:

```
X70:=NEG LIIX; X01:=NEG X23;
```

Care should be taken that the same integer accumulator is not used on either side of the assignment. For example:

```
X56:=NEG X67;
```

will not give the result that would be expected as the code generated is equivalent to:

```
X6:=NEG CARRY X1;
X5:=NEG X6;
```

In the case of the same long integer accumulator on both sides of the := symbol, no code is generated if it occurs on the same line while the accumulators are loaded from store if the split occurs after the:=. Thus:

```
X12:=
X12+3D;
```

generates the code equivalent to:

```
X2:=(2); X1:=(1); X2:=X2++LIA(2); X1:=X1+LIA(1)
```

assuming LIA(1) and LIA(2) contained the constant 3 double length.

10.3 General Long Integer Accumulator Assignments

A general long integer accumulator assignment extends the simple form. As with integer assignments, the first part of the statement equivalent to the basic statement assigns a value to a long integer accumulator. The remaining terms of the statement then cause various arithmetic and shift operations to be performed on the contents of the long integer accumulator. These operations are performed strictly from left to right. The possible operations are similar to the arithmetic and shift operations defined for the integer accumulator assignments. The difference is that results apply to the 48 bit integer accumulator pair rather than a single integer accumulator. The full set of addition and subtraction operators are allowed including those which set CARRY(++ and --). Basically each operator/ operand generates two instructions. In the case of the shift operations, use is made of the double-length instructions available on the 1900 so that, in this case, only a single order is often required. Some examples are:

```
X67:=X67 SLA 2 - LIIA(2) SRL 3;
X12:=X67 - LIIX * LIIA(X3);
```

It is possible in some circumstances to use integer primaries in long integer assignments. Frequently the code generated is not what would be expected. As a general rule the address of the integer primary is assumed to be that of a long integer quantity so that the contents of the next address are also used. For example :

```
X67:=LIA(2);
```

does not set X67 equal to the value of LIA(2) but that contained in LIA(2) and LIA(3). Short integer quantities are compiled correcly so that X67:=3; gives the desired result. However, X67:=32670; accesses the addresses assigned to 32670 and also the following address which could contain anything. It is, therefore, dangerous

to mix long integer and integer primaries in a statement even though the compiler does not fault it. The result will almost certainly be different from that expected.

The PLASYD compiler does allow * and / to be used in long integer assignments. However, the results are not what the user could normally expect. Consequently, their use is not advised. Basically the multiply only operates on the top halves of the long integer quantities, while the divide takes the 48 bit quantity and divides it by the top 24 bits of the operand. It would be sensible for anyone attempting to use these operators to examine the code generated first.

10.4 Long Real Accumulator Assignments

Accumulator assignments for long reals are quite simple in form. The basic assignment allows the long real accumulator to be assigned a long real value or a long real primary. For example:

```
A12:=1.75L; A12:=3.97825L; A12:=MINUS 9.58& MINUS 25L;
A12:=LRRX; A12:=LRRA(2);
```

The monadic operator NEG can be used to assign the negative of the primary value to the long real accumulator. For example:

```
A12:=NEG 1.75L; A12:=NEG LRRX;
```

The general long real accumulator assignment extends the basic form. The first part, equivalent to the basic statement, assigns a value to the long real accumulator. The remaining terms of the statement then cause various arithmetic operations to be performed on the contents of the long real accumulator. These operations are performed strictly from left to right. For example:

```
A12:=A12+LRRX*LRRA(0)-LRRA(2)/LRRA(4);
A12:=LRRX*LRRX+LRRA;
```

11.

CELL ASSIGNMENTS

11.1 Syntax

```
xcellassignment ::=xcell:=xacc|xcell:=0|xcell:=storeop xacc|
                   xcell:=partop xacc|samexcellassign|
                   xcellassignment addop xcell|xcellassignment
                   logical xacc

rcellassignment ::=rcell:=A1|rcell:=0.0

xxcellassignment::=xxcell:=xxacc|xxcell:=0|xxcell:=NEG xxacc|
                   samexxcellassign|xxcellassignment addop xxacc

rrcellassignment::=rrcell:=A12

storeop          ::=NEG|CARRY|NEG CARRY
partop           ::=EX|SA|LA|CH
addop            ::=+|-|++|--

samexcellassign ::=xcell:=xcell
samexxcellassign::=xxcell:=xxcell
```

11.2 Introduction

As the 1900 series order code has few operations involving only operands in store and not using accumulators, the possible cell assignment statements are few. In the more complex statements involving several operators on the right hand side, it should be remembered that each operator/operand pair will cause a store access. Consequently, it is usually more efficient to use accumulator assignments especially if the right hand side of the statement is at all complex. For example:

```
X1:=X2+X3-X4++X5--X6 AND X7:
LIX:=X1;
```

is more efficient than:

```
LIX:=X2+X3-X4++X5--X6 AND X7;
```

Although the first method will generate one extra instruction, only one instruction will involve a store access. In the second method all six instructions will cause store accesses on the same cell. On the 1906A, this is certain to be less efficient.

In the syntax definitions for samexcellassign and samexxcellassign, the xcell and xxcell on the left and right hand sides of the assignment operator, :=, must be identical and on the same line.

11.3 Integer Cell Assignments

The two simple formats are the assignment of an integer accumulator or zero to a cell. For example:

```
LIX:=X0; LIA(1):=X1; LIA(-1):=X2; (4):=X3; (X1):=X2;
(X1+3):=X5; UIA(X1):=X6; UIA(X1+3):=X7; LIA(X1):=X0;
UIV(X1-5):=X2; LIX:=0; UIV(X1-2):=0;
```

Assignments of zero to cells use the special 1900 order for this operation. In the cases where the left hand side's type is not obvious, the type is assumed to be correct for the RHS. Thus (3):=X1 will assume that an integer cell assignment is required. In the potentially ambiguous situation of an assignment of zero to a cell, the cell type of the left hand side is assumed to be INTEGER and not LONG INTEGER. Thus (3):=0 sets only the contents of address 3 to zero.

The integer accumulator on the right of the assignment may be preceded by a monadic operator. The meanings of the operators are similar to those for integer cell assignments. The possible monadic operators are:

(a) NEG : this causes the negated value of the integer accumulator specified to be assigned to the cell.

(b) CARRY : the cell specified is assigned the value of the integer accumulator without its most significant bit. The CARRY register is set if the most significant bit of the primary is set. As in all 1900 integer arithmetic, the value assigned to the cell will be incremented by the value of the CARRY bit. So if X1 contains # 37777777 and CARRY is set then LIX:=CARRY X1; will set LIX to 0 and will propagate the CARRY leaving the CARRY register set.

(c) NEG CARRY : the value of the integer accumulator is first negated and the operator CARRY is performed on the result before the cell assignment.

(d) EX : this assigns the bottom 9 bits of the integer accumulator specified to the cell designated but does not alter the other 15 bits of the cell. Thus it is not strictly equivalent to EX in accumulator assignments where the remaining 15 bits are set to zero.

(e) SA : similar to EX except that 12 bits are deposited instead of 9. The remaining bits are unaltered.

(f) LA : similar to EX except that 15 bits are deposited instead of 9. The remaining bits are unaltered.

Assuming LIX contains # 07777777 and X1 contains # 33333333 then
LIX:=EX X1 sets LIX to # 07777333
LIX:=SA X1 sets LIX to # 07773333
LIX:=LA X1 sets LIX to # 07733333

(g) CH : sets the bottom character of the integer accumulator into the designated cell. Which character position is changed will depend on the form of the cell specified. If the cell is unmodified, the lower 6 bits of the cell will be changed. If the cell is defined using a modifier, then the address is taken as a character address and the assignment alters the bits in that character position of the cell. For example if X1 contains 'ABCD' and LIX contains '1234' then
LIX:=CH X1; will set LIX to '123D'
X2:=0; LIX(X2):=CH X1 will set LIX to 'D234'
X2:=CHAR 2; LIX(X2):=CH X1; will set LIX to '12D4'

Certain arithmetic and logical operations can be carried out on the values of integer cells without going via an accumulator. A more complex integer cell assignment takes one of the forms described above followed by any number of operator-accumulator pairs. It is also possible to modify the existing contents of a cell by the first part of the statement consisting of the same cell designation on either side of the := operator. This part of the statement must be on the same line. Thus LIX:=LIX+X1; is possible but LIX:=LIA+X1; is not allowed.

The possible operators are +,-,++,--, AND, OR, ER and these have the same meaning as given in Section 8.7. As for accumulator assignments all operations are carried out strictly from left to right. Examples of possible statements are:

```
LIX:=LIX+X1-X2++X3--X4 AND X5 OR X6 ER X7;
LIX:=X0+X1;
LIX(X2):=X3+X4;
(3):=(3)+X2;
```

11.4 Real Cell Assignments

The possible forms of real cell assignments are very limited with only the real accumulator and zero being possible right hand sides. For example:

```
LRX:=A1; (3):=A1; LRA(X1):=A1; URA(X1-2):=A1;
LRX:=0.0; URA(X1):=0.0;
```

Note that zero is the only constant that can be directly assigned to a cell. To assign the constant 3.0 to LRX requires:

```
A1:=3.0; LRX:=A1;
```

To assign zero to a cell, the left hand side must have its type known. There is a compiler error such that (3)=0.0, for example, is compiled incorrectly.

11.5 Long Cell Assignments

The only possible long real assignment is assigning the long real accumulator to a cell. For example:

```
LRRX:=A12; LRRA(X1-4):=A12;
```

The long integer cell assignments are similar to the corresponding integer cell assignments. The two simple forms allow a long accumulator or zero to be assigned to the designated cell. For example:

```
LIIX:=X12; LIIX:=0; (X1):=X12;
```

The only monadic operator allowed is NEG which assigns the negated value of the long integer accumulator specified to the cell. For example:

```
LIIX:=NEG X23;
```

More complex assignment statements can be generated by following any of these simple statements with any number of operator/long accumulator pairs. These are obeyed strictly from left to right. As with the integer and real cell assignments, it is possible to define the same cell on both the left and right of the := operator if the current contents of the cell are to be modified. For example:

```
LIIX:=LIIX+X12-X34++X56--X70;
LIIX:=0+X23;
LIIX:=NEG X12+X23;
LIIA(X1+2):=LIIA(X1+2)+X23-X45;
```

The possible operators are the same as for integer cell assignments. The difference is that operations are performed on 48 bit quantities instead of 24.

Although logical operations have been defined for double length integer quantities, the code generated is incorrect. These should not therefore be used.

12.

BLOCK STRUCTURE

12.1 Syntax

```
block     ::=BEGIN blockbody label? END|BEGIN declist blockbody label? END

blockbody::=statement;|blockbody statement;

declist   ::=dec|declist dec

dec       ::=accsyndec|cellsyndec|definedec|externaldec|
              nonplasyddec|proceduredec|puredec|storagedec
```

12.2 Block Structure

In the previous sections the most frequently used executable statements, the assignment statements, have been defined. These will normally be executed sequentially as written. However, PLASYD does contain control and conditional statements which can change the order in which statements are executed. For example, it is possible to skip a group of statements using the conditional statement. To make these instructions as powerful as possible, statements may be grouped together in a block which can then act as though it were a single statement. A block consists of the word BEGIN followed possibly by some declarations, then some executable statements and the word END, possibly preceded by a label (labels will be defined later). For example:

```
BEGIN
INTEGER I,J;
X1:=£J;
X4:=3;
J(X1):=X4;
I(X1):=X4;
END
```

The semicolon terminating the statement immediately preceding the END statement may be omitted.

A single PLASYD program will usually consist of a single block. In general, though, a program will consist of a number of blocks. As a block is a special case of a PLASYD statement, it can be part of another block so that blocks can be nested to any depth. For example:

```
BEGIN
   INTEGER I,J;
   .....
   BEGIN
      INTEGER K;
      .....
      BEGIN
         INTEGER L;
         X4:=I(X2)+K(X1)+L(X3);
      END;
      .....
   END;
   .....
   BEGIN
      INTEGER M,N;
      .....
   END;
   .....
END
```

Unlike Algol 60, identifiers must be unique within the complete segment being compiled. The block structure is therefore a means of allocating storage and reusing it. It does not have any effect on the identifiers allowed.

The only variables which can be used at any point in the program are those which have been declared in the current block or a surrounding block. In the example above, statements in the innermost block containing the assignment to X4 can access the variables whose names are I, J, K and L but cannot access the variables M and N.

There is no dynamic storage allocation in PLASYD. Variables are allocated storage space at compile time. An attempt is made to do this as efficiently as possible. Storage allocated to variables declared in low level blocks may be reused at a later point in a program. In the example above, the variables L and M would be allocated the same storage.

12.3 Label on END

A label may be placed before the END symbol of a block as follows:

```
LB:END
```

If a jump is made to this statement, the effect is as though a jump was made to a null statement immediately before the END statement. No null instruction will be generated in this case.

13.
PROCEDURES AND LABELS

13.1 Syntax

```
label                ::=gl? lidentifier:|ENTRY digit:
lidentifier          ::=identifier

proceduredec         ::=gl? PROCEDURE pidentifier (xacc ret?);statement;
pidentifier          ::=identifier
ret                  ::=,integer
gl                   ::=GLABEL

procedurestatement   ::=lidentifier|pidentifier
returnstatement      ::=RETURN|RETURN(integer)
gotostatement        ::=GOTO lidentifier|GOTO pidentifier|
                         GOTO lidentifier|GOTO pidentifier
datastatement        ::=DATA initial|DATA(initiallist)
obeystatement        ::=OBEY xcell
```

13.2 Labels

Normally a PLASYD program will be entered at the first statement following the initial declaration and the following statements will be obeyed in the order in which they appear, intervening declarations being skipped. If it is required to alter the order in which statements are obeyed, a control statement must be used. The simplest form of control statement is a jump to a different part of the program. In order to designate a statement to which control is to be passed, the statement must be labelled. The simplest way of doing this is to precede the statement by an identifier and a colon. The identifier must be unique within the program or section of program being compiled. It does, therefore, differ from the more usual Algol definition where the label definition acts as a declaration of that identifier as a label in the current block. For example:

```
BEGIN
   INTEGER I;
   BEGIN
      INTEGER J;
      .....
     L:.....
   END;
   BEGIN
      INTEGER K;
      ......
     L:.....
   END;
END
```

This program is not allowed in PLASYD although it would be allowed in Algol.

There is no objection to a statement having several labels attached to it. Some examples are:

```
A:B:C:OLE:X3:=B(£B);
L:X1:=X1+6;
NEXTCH:B:=X4-X2;
DB4:X1:=$NEST;
```

Accessing of labels from other separately compiled segments is described in Chapter 19.

13.3 GOTO Statement

The simplest way to transfer control unconditionally is the GOTO statement. Note that the word GOTO can be written as two words, GO TO if the user desires. The effect of this statement is to cause the statement designated by the label identifier and all statements following it to be obeyed in sequence until another control statement is reached. For example:

```
BEGIN
   INTEGER I, J;
   GOTO L;
   X0:=3;
 L:X1:=2;
   GOTO K;
   X2:=4;
 K:X3:=2;
END;
```

This program will set X1 and X3 to 2, but will not alter the contents of X0 and X2.

The syntax also allows the GOTO statement to be used as a means of entering a procedure. This will be discussed in the section 13.5 concerned with procedure calling.

13.4 Procedure Declarations

A procedure is a named group of PLASYD statements which may be obeyed at any point in a program by executing a procedure statement. After they have been obeyed control will normally return to the statement following the call. A procedure declaration specifies the group of statements that are to form the procedure and assigns a name to them. It also specifies an integer accumulator called the link. When the procedure is called from another part of the program the address of the next instruction to the call will be placed in this link

accumulator. Procedure declarations, like other declarations, must appear at the head of a block. The scope of the procedure name is, however, similar to that for a label. The procedure name may not be used elsewhere within the program. It is possible to call a procedure at a point in the program before the declaration if required, and even to call it from outside the block in which it is declared. Communication between the rest of the program and a procedure can be via any of the variables declared in the outer block or the accumulators. There is, therefore, little reason for defining procedures at any level other than the outer block level. It is possible to define procedures as separately compiled segments which can be accessed via global variables. Full details of these will be given later.

A simple example of a procedure declaration is:

```
PROCEDURE ZERO (X1);
BEGIN
A1:=0.0;
X0:=0;
END;
```

The identifier ZERO is the name by which this procedure will be known. The link accumulator in this case is X1. Consequently, a call of the procedure ZERO will set X1 to point to the next statement after the procedure statement which made the call. The result of calling this procedure will be to set the real accumulator and the first integer accumulator to zero. At the end of the procedure control will transfer back to the instruction specified by X1. The body of the procedure declaration is a single statement which will normally be a block. However, it can be a single statement.
For example:

```
PROCEDURE ZERO (X1);
X0:=0;
```

is allowable.

13.5 Procedure Calls

A procedure may be called at any point in the program by executing a procedure statement. This is a simple statement consisting merely of the procedure identifier. For example:

```
ZERO;
```

would cause the procedure defined in the previous section to be entered. The effect of the procedure statement is to store the address of the next instruction in the link accumulator and then to transfer control to the statement that forms the body of the procedure. This will usually be a block. When this statement has been executed, providing the link accumulator has not be changed, control will normally pass to the statement following the call and proceed in normal sequence from there.

If the link accumulator needs to be used for other purposes in the body of the procedure, its entry value should be stored and then recovered before exit. Exit from the procedure body is achieved:

1) After executing the statement immediately before the final END if the procedure body is a block.

2) Jumping to a label attached to the final END.

3) By executing the RETURN statement.

For example:

```
PROCEDURE ZERO(X1);
BEGIN
A1:=0.0;
END;

PROCEDURE ZERO(X1);
BEGIN
A1:=0.0;
GOTO LAST;
.....
LAST:END;

PROCEDURE ZERO(X1)
BEGIN
A1:=0.0;
RETURN
.....
END;
```

All three procedures set the real accumulator to zero and exit.

13.6 Parameters and the OBEY Statement

One method of passing data between the calling program and the procedure is by using the accumulators and variables declared in the outer block. However, there is no standard method of passing parameters and the programmer may use any method he likes. The standard method adopted in PLAN programs is to follow the procedure call by statements to load the integer accumulator X3 with the addresses of the parameters.

Some means must be provided to access this statement. The OBEY statement causes the 1900 machine code instruction stored in the integer cell specified to be obeyed. If it does not cause a change of control, then, after execution, control will return to the statement following the OBEY. Consider the following program:

```
BEGIN
REAL F,G;

PROCEDURE SQUARE(X1);
BEGIN
OBEY(X1);
A1:=(X3)*(X3);
X1:=X1+1;
END;

X2:=£F;
A1:=3.0;
F(X2):=A1;

SQUARE;
X3:=@F;

RET:G(X2):=A1;
END
```

Before calling the procedure SQUARE, the variable F is set equal to 3.0. The procedure SQUARE executes the statement stored in the position pointed at by the link accumulator. This is the action of the OBEY statement. The next statement sets the real accumulator to the square of the argument. Before leaving the procedure SQUARE, the integer accumulator X1 is increased by 1 so that the link now points at the statement labelled with RET. On returning from the procedure SQUARE, G is set equal to the value of the real accumulator (9.0 in this case).

13.7 Return from Procedure

The statement X1:=X1+1 in the previous section can be omitted if the procedure SQUARE is defined with the second integer argument. The second argument causes the control to be returned to the address of the instruction following the procedure call incremented by the value of the integer.

For example:

```
PROCEDURE SQUARE (X1,1);
```

would mean that the incrementing of X1 would not now be necessary. In general, the integer value will be equal to the number of arguments to the procedure. This will ensure that the instruction following the arguments is next obeyed on return from the procedure.

An alternative method of specifying a different return address from the instruction following the procedure call, is to exit from the procedure using the RETURN statement with an integer argument. This has the same effect as the second argument in the procedure declaration. If a procedure is declared with the second argument n and a RETURN

statement with an argument m then the RETURN takes precedence and control is returned to m instructions past the one following the procedure call. If RETURN has no arguments, control is returned to n instructions past the one following the procedure call.
For example:

```
  PROCEDURE SQUARE (X1,2);
  BEGIN
  ........
A:RETURN;
  ........
B:RETURN(3);
  ........
C:END;
```

The return address in the three different exit positions is:

A:X1+2
B:X1+3
C:X1+2

13.8 Data Statement

```
datastatement   ::=DATA initial|DATA(initiallist)
initiallist     ::=initial|initial*integer|initiallist,initial|
                   initiallist, initial*integer
initial         ::=xinitial|rinitial|xxinitial|rrinitial
αinitial        ::=αvalue  {α=r|xx|rr}
xinitial        ::=xvalue|string|fixedaddress|functionstatement|
                   count+fixedaddress|count+integer|count+truncated|
                   count+octalinteger
string          ::="stlist"
stlist          ::=st|stlist st
st              ::=char|" "|  |'

fixedaddress    ::=basicfixed|CHAR integer|CHAR integer OF basicfixed|
                   @identifier|@lidentifier
basicfixed      ::=@fixedcell|$fixedcell|£identifier
fixedcell       ::=xfixedcell|xxfixedcell|rfixedcell|rrfixedcell
αfixedcell      ::=αidentifier|αidentifier(integer)|αidentifier(-integer)
                   {α=x|r|xx|rr}
```

The DATA statement provides an alternative means of passing arguments to procedures. Its effect is to store the initial values at that point in the program. In its simplest form, the DATA statement allows constant values of all types to be stored. For example:

```
DATA 3;
DATA (3.7,1,57D,27.3L);
```

Particular constants may be repeated a number of times using the initial*integer form.

For example:

```
DATA (3.7,1*5,2.3);
```

This will cause the constants 3.7,1,1,1,1,1,2.3 to be stored at that point in the program. The integer following the * indicates the number of times the constant is to be repeated. As well as constants, the following integer values are allowed:-

(a) string : this is a sequence of characters enclosed in double quotes ("). The double quote symbol is itself represented by two double quotes in succession. The characters are inserted in sequence, four per 1900 word and spaces are added at the end if necessary to bring the total number of characters to a multiple of four. Note that 'A' and "A" are not the same. "A" is stored as 41202020 while 'A' is stored as 00000041. Thus "A" is equivalent to 'A ', while 'A' is equivalent to "000A".

(b) fixedaddress : these fixed addresses define a position in store. They comprise a subset of the addresses defined in Section 8. A fixedaddress is one that can be completely evaluated at compile time. It may not, therefore, be modified.

(c) count : a fixed address or integer value may be preceded by count and the symbol +. The count will be placed in the top 9 bits of the integer cell and the remainder, which must be expressable in the lower 15 bits or less, will be placed in the remainder of the word. For example:

```
6CNT+3,17CNT+@I,5CNT+$J(3),
12CNT+MINUS3, 5CNT+27T2,7CNT+#22
```

(d) function : this is a means of inserting a specific 1900 machine instruction at this point. Full details are given later. Examples might be:

```
!LDN(X1,4) and !FAD(3,B).
```

The previous example of the procedure SQUARE could be modified so that the value to be squared is contained in the location following the procedure call as follows:

```
BEGIN
REAL G;

PROCEDURE SQUARE(X1,2);
BEGIN
A1:=(X1)*(X1);
END;

SQUARE;
DATA(3.0);
X2:=£G;
G(X2):=A1;
END;
```

Note that as the data item is a real quantity, it will take up two locations and consequently the return address must be incremented by 2 using the second argument in the procedure declaration. A more complicated example might be:

```
XXX;
DATA(1,3.0,22D,3.5L,"ABCDE");
```

Here the return address could have to be incremented by 11(1+2+2+4+2).

13.9 Recursion and GOTO Exit

Procedures should not directly or indirectly call themselves since recursion is not catered for directly by the PLASYD system. If recursion is necessary the user must make his own arrangements to stack the link unless return down the recursion chain is not needed. In this case the final procedure must be left by a GOTO jump to a label external to the procedure.

Any procedure, whether recursive or not, can be left by a GOTO statement. Care must be taken in the case when the label jumped to is in a block parallel to the one in which the procedure is declared. As storage is reallocated, values of variables could be lost. This also applies in the case where a procedure is called from a block not enclosed by the one in which the procedure is declared. It is good practice only to call procedures from the one in which the procedure is declared or a block enclosed by this procedure. Consider, for example:

```
BEGIN
  INTEGER I;

U:BEGIN
      INTEGER J,K;
      PROCEDURE FRED(X1);
      BEGIN
         X2:=£J;
         J(X2):=0;
         .........
         GOTO LB;
         .........
         RETURN;
         .........
      END;
      .........
    A:FRED;
      .........
  END;
  .........
V:BEGIN
     INTEGER L,M;
     .........
   B:FRED;
     .........
  LB:.........
  END;
END
```

Here the procedure FRED is defined within the block labelled U and called from both this block and the parallel block labelled V. Exit from the procedure is either to the label LB or a standard return. If we first consider the call of FRED labelled A, the standard return will have the variable J available and set to zero. If the procedure is left by the GOTO then the return address will be in block V and the variable J will now not be available and the fact that it had been set to zero would not be known. If the procedure had been called from the statement labelled B in block V then either the normal return or the GOTO exit will be back to the block V. However, the blocks U and V may well share storage for the variables J,K and L,M. Consequently, the updating of the value of J by the procedure FRED may well have altered the value of the variable L.

In order to allow recursion of PLASYD procedures, the user must set up a stack to hold the contents of the link accumulators. A simple system could be organised where all procedures use the same link accumulator and the only procedure exits must be standard returns and not GOTO jumps. In this case a single stack will be all that is required. For example:

```
BEGIN
  LOWER
  INTEGER STACK(100),STACKPTR=1;
  LOWEND;

  PROCEDURE A(X1);
  BEGIN
     X2:=STACKPTR;
     STACK(X2):=X1;
     STACKPTR:=X2+1;

     .........

     X2:=STACKPTR-1;
     X1:=STACK(X2);
     STACKPTR:=X2;
  END;
  ........
END
```

If all procedures have the same statements at the head and return position as procedure A, then they may be called recursively up to a maximum depth of 100.

13.10 The Equivalence of Labels and Procedures

Labels and procedures are to a large extent interchangeable in PLASYD. A procedure call always sets up the link value while a GOTO jump does not. It is possible both to enter a procedure by a GOTO jump or to enter at a label within a procedure by a procedure call.

(a) Jump to procedure:

The first statement of the body of a procedure may be jumped to by a GOTO statement. No link will be stored and if the end of the procedure

body is reached or a RETURN statement is executed then the next statement obeyed will be determined by the contents of the link accumulator at that point. The main use of the facility is, therefore, when a string of procedures is to be obeyed with no return from each required.

Another place in which this facility can be of use is when a procedure calls a subprocedure and, on return from this subprocedure, immediately leaves the main procedure. In this case a GOTO the subprocedure will, when the return is reached, exit to the place from which the main procedure was called assuming both procedures use the same link. This is more efficient than a standard procedure call to the subprocedure. For example:

PROCEDURE CALL

```
BEGIN
  PROCEDURE MAIN(X1);
  BEGIN;
  SUB;
  END;
  PROCEDURE SUB(X3);
  BEGIN
  X2:=0;
  END;
 MAIN;
END
```

GOTO CALL

```
BEGIN
  PROCEDURE MAIN(X1);
  BEGIN
  GOTO SUB;
  END;
  PROCEDURE SUB(X1);
  BEGIN
  X2:=0;
  END;
  MAIN;
END
```

(b) Calling a label:

A label declared in the body of a procedure may be treated as a procedure with the same link as the procedure itself. This can be used to provide alternative entry points to a procedure. For example:

```
PROCEDURE SIN(X1);
BEGIN
........
COS:.....
.....
END;
```

Here both SIN; and COS; are statements that can appear in the program. The return action will be the same in both cases.

14.

CONDITIONAL AND CONTROL STATEMENTS

14.1 Syntax

```
casestatement       ::=CASE modifier OF statementlist CASEND
statementlist       ::=statement;|statementlist statement;

ifstatement         ::=IF logicalexpression THEN simplestatement ELSE
                        statement|IF logicalexpression THEN statement

statement           ::=simplestatement|complexstatement
simplestatement     ::=assignment|cellassignment|functionstatement|
                        nonplasydstatement|procedurestatement|
                        casestatement|block|gotostatement|returnstatement|
                        datastatement|obeystatement|NULL|
                        label simplestatement

complexstatement    ::=ifstatement|forstatement|whilestatement|
                        includestatement|label complexstatement

assignment          ::=xassignment|rassignment|xxassignment|rrassignment
cellassignment      ::=xcellassignment|rcellassignment|xxcellassignment|
                        rrcellassignment

logicalexpression   ::=andexpression|orexpression|condition
andexpression       ::=condition AND condition|andexpression AND condition
orexpression        ::=condition OR condition|orexpression OR condition

condition           ::=relationex|OVERFLOW|FOVERFLOW|CARRY|NOT OVERFLOW|
                        NOT FOVERFLOW|NOT CARRY
relationex          ::=xrelationex|rrelationex|xxrelationex|rrrelationex

αrelationex         ::=αacc relation αprimary {α=r|xx|rr}
xrelationex         ::=xacc relation xprimary|xacc relation partaddress|
                        xacc charel xprimary
partaddress         ::=£cellidentifier|$cellidentifier
cellidentifier      ::=upperidentifier|loweridentifier
relation            ::=<|>|<=|>=|NOT=|=
charel              ::=<CH|>CH|<=CH|>=CH

forstatement        ::=FOR xassignment STEP increment UNTIL xprimary DO
                        statement
increment           ::=xprimary|-xprimary|CHAR

whilestatement      ::=WHILE logicalexpression DO statement|DO statement
```

14.2 CASE Statement

This causes one only of a number of statements to be obeyed according to the value of a modifier. It is important that each of the statements generates only one machine instruction. For example:

```
CASE X1 OF
A1:=A1+LRX;
A1:=A1-LRX;
A1:=A1*LRX;
A1:=A1/LRX;
GOTO LB;
X1:=NEG LIX;
LIX:=0;
URX(X2):=A1;
LIX:=LIX+X2;
CASEND;
```

The effect of the statement is to cause the i'th instruction of the sequence to be obeyed where the sequence is numbered starting at zero and i is the value of the modifier. In the example above, if X1 had the value 3 then the statement A1:=A1/LRX would be obeyed.

Unless the statement obeyed is a control jump of some kind, the next statement obeyed is the one following the CASEND. The modifier is destroyed by this instruction. For example, X1 will not still have the value 3 after the instruction above has been executed.

The CASE statement is implemented by storing the statements in storage, adding the base of these statements to the modifier and then executing an OBEY instruction with the modifier giving the instruction address.

Care should be taken that the statements in the CASE only generate single machine code instructions. This does mean that any instruction containing a SMO index is not allowed. Some examples of statements generating a single instruction are:

1. Accumulator assignments whose right hand side consists of a single primary or whose right hand consists of the left hand side, an operator and a primary. In general long assignments are not allowed nor is the operator * allowed with integer operands. This is because it generates two instructions.

2. A real or integer cell assignment whose right hand side consists either of an accumulator or the left hand side together with an operator and a primary. The assignment of zero to a real cell is not allowed as this generates two instructions.

3. A procedure statement.
4. A GOTO statement.
5. An OBEY statement.
6. A NULL statement.
7. A function statement.

If in doubt the SWITCH(0) facility will indicate which instructions have been generated.

If the modifier value specifies a non-existent instruction the effect is undefined.

14.3 IF Statement

The IF statement has basically two different formats similar to Algol. In the simple form:

IF logical expression THEN statement

the statement following the THEN is only obeyed if the logical expression has a value TRUE. Control is then passed to the statement following the IF statement assuming the one following the THEN is not some kind of control jump. In the more complex form:

IF logicalexpression THEN simplestatement ELSE statement

the simple statement following the THEN is executed if the logical expression has a value TRUE and the statement following ELSE is executed if the logical expression has a value FALSE. In either case, assuming the statements obeyed were not control jumps, control then passes to the statement following the IF statement. Thus one or other of the statements is obeyed but not both. Note that only the subclass of simple statements is allowed after the THEN. This is to remove the standard ambiguity of the dangling ELSE. For example in the statement:

```
IF X1=0 THEN IF X2=0 THEN GOTO L1 ELSE GOTO L2
```

the ELSE is associated with the second of the IF statements and could have been written:

```
IF X1=0 THEN
BEGIN
IF X2=0 THEN GOTO L1 ELSE GOTO L2;
END;
```

The statement brackets, BEGIN and END, can always be introduced in this way to avoid ambiguity.

In its simplest form the logical expression is either a relation or one of the status conditions. The status conditions are set to true if:

OVERFLOW	the overflow indicator is set (testing overflow clears it).
FOVERFLOW	the real (floating point) overflow indicator is set.
CARRY	the carry bit is set (if it is part of a multiple condition it should be the first item).
NOT OVERFLOW	the overflow indicator is not set.
NOT FOVERFLOW	the real overflow indicator is not set.
NOT CARRY	the carry bit is not set.

The simple relations compare some primary with the current value of an accumulator. These are quite self explanatory, being the standard set of comparisons. For example:

```
A1>0.0, A12<3.0L, X1=3, X2>=3, X12<=7D, X4 NOT=3
```

Two additional types of relation are permissible between integer accumulators and variables. The first of these:

```
xacc relation partaddress
```

allows comparisons to be made between the contents of an accumulator and the base or displacement of a call. For example:

```
X1>£LIX, X2=$UIA, X3<=£URA
```

The second relation:

```
xacc charel xprimary
```

allows special character comparisons to be made. The effect is to test the value of the integer accumulator considered as four six-bit characters against the primary similarly considered. CH comparisons are more efficient than normal relations. They can be used in place of arithmetic comparisons providing both operands are of the same sign. If they are of different signs, a negative number is unfortunately treated as larger than a positive one. The comparison is assumed to be a CH one if the primary is a character sequence used with a normal relation. Thus:

```
X1>='ABCD'
```

is taken as identical to:

```
X1>=CH 'ABCD'
```

The CH is part of the comparison operator and should not therefore be separated by spaces from the remainder.

In its more complex form, the logical expression may consist of a number of relations all of which must be satisfied (andexpression), or only one need be satisfied (orexpression). For example:

```
X1>3 AND A1>URX(X2) AND NOT OVERFLOW
CARRY OR X1<CH LIX OR A1<URA(X1+3)
```

Note that AND's and OR's cannot be mixed so that to jump to label LB if X1 is zero and either X2 or X3 are also zero requires a statement of the form:

```
IF X1=0 THEN BEGIN IF X2=0 OR X3=0 THEN GOTO LB; END;
```

14.4 FOR Statement

Although loops can be written using an IF statement and a GOTO statement, the FOR statement provides an easier method of causing a group of statements to be obeyed a specified number of times. For example:

```
FOR X1:=LIX STEP LIA UNTIL 10 DO
LRX(X1):=0;
```

is equivalent to:

```
   X1:=LIX;
AG:LRX(X1):=0;
   IF X1=10 THEN GOTO ND;
   X1:=X1+LIA;
   GOTO AG;
ND:NULL;
```

First the accumulator assignment is made followed by the execution of the statement. Then the value of the accumulator is checked for equality with the upper limit. If it does not equal it then the value of the accumulator is incremented and the statement executed again. The loop continues until the exact value of the limit is reached. Thus the statements:

```
FOR X1:=3 STEP 2 UNTIL 8 DO X7:=X7+3;
FOR X1:=3 STEP 1 UNTIL 2 DO X7:=X7+3;
```

are both infinite loops. In any case where there is doubt that exact agreement will occur, a WHILE statement should be used.

The integer accumulator assignment initially can be any of the allowed forms (including address assignment). The increment can be any integer primary possibly preceded by a - sign. Some examples are:

```
FOR X1:=LIX(X2+2)+LIA(2) STEP #770 UNTIL UIA(X2) DO .....
FOR X1:=10+LIX STEP -1 UNTIL 0 DO.....
FOR X1:=X1+$LRX STEP 1 UNTIL $LRA(4) DO.....
```

It is more efficient to have a limit of zero than any other primary.

The increment may also be the symbol CHAR. This will cause the value of the accumulator, treated as a character address, to be incremented by one character address at each iteration of the loop. Note that a BCHX order will be used to do the incrementing and consequently, if the compilation is in 15AM mode, the count position of the accumulator will be altered. The limit value must therefore take account of this.

The limit must be an integer primary so if the final value is a complicated expression, this should be calculated first and assigned to an accumulator or store location.

Some examples are:

```
FOR X3:=LIX STEP 1 UNTIL LIA(1) DO
BEGIN
   UIA(X1+1):=X3;
   (X3):=1;
END;

X7:=£IUA+6+X1;
LIX:=X7;
FOR X2:=CHAR 1 OF £UIA STEP CHAR UNTIL LIX DO
BEGIN
   X6:=CH UIA(X2);
   IF X6>'Z' THEN GOTO EXTRA;
END;
```

This example only has meaning in 22AM mode.

14.5 WHILE Statement

This is similar in effect to a FOR statement in that it allows a statement to be executed repeatedly but in this case execution ceases when a condition specified by the programmer no longer holds. For example:

```
WHILE A1<LRX DO
BEGIN
   X1:=X1+1;
   A1:=A1+LRA(1);
END;
```

This is equivalent to:

```
LBL:IF A1<LRX THEN
BEGIN
   X1:=X1+1;
   A1:=A1+LRA(1);
   GOTO LBL;
END;
```

In the general form where the logical expression is present, the condition is tested first and if it is TRUE the statement is executed and the test repeated. If the condition no longer holds, control passes to the next statement in the sequence.

The WHILE statement is more flexible than the FOR statement as it can be used to construct loops without an exact end. Also the loop index can be a real accumulator. Thus :

```
FOR X1:=3 STEP 2 UNTIL 2 DO ..... and
FOR X1:=3 STEP 2 UNTIL 10 DO .....
```

are both illegal FOR loops in that they will be repeated for ever.

However, similar WHILE loops will both terminate:

```
X1:=3; A1:=3.0;
WHILE X1<2 DO BEGIN ..... X1:=X1+2; END;
WHILE A1<10.0 DO BEGIN .. A1:=A1+2.0; END;
```

Conditions involving zero are more efficient than others.

There is a special form of WHILE statement for the repeated execution of a block from which there is some conditional exit. This has the simple form:

```
DO statement;
```

which is equivalent to

```
AG: statement; GOTO AG;
```

For example:

```
DO
BEGIN
  X1:=X1+1;
  X7:=LIA(X1);
  LIA(X1-1):=X7;
  IF X1> 9 THEN GOTO EXT;
END;
```

15.

FUNCTIONS

15.1 Syntax

```
functionstatement   ::=simplef|oneargf|twoargf
simplef             ::=!fidsimple
fidsimple           ::=NULL|SUSAR|LFPZ
oneargf             ::=!fidone  (nparam)

fidone              ::=BRN|BVS|BVSR|BVC|BVCR|BCS|BCC|BVCI|SUSTY|DISTY|
                        DELTY|SUSWT|DISP|DEL|OBEY|MODE|FIX|SFPZ|SMO|
                        STOZ|ACT|RMS

twoargf             ::=!fidtwo (xparam, nparam) |!fidtwo (,nparam)

fidtwo              ::=SFP|SUSBY|REL|DIS|CONT|SUSDP|ALLOT|PERI|SUSMA|
                        AUTO|SUSIN|GIVE|RRQ|LDX|ADX|NGX|SBX|LDXC|ADXC|
                        NGXC|SBXC|STO|ADS|NGS|SBS|STOC|ADSC|NGSC|SBSC|
                        ANDX|ORX|ERX|LDCH|LDEX|TXU|TXL|ANDS|ORS|ERS|
                        DCH|DEX|DSA|DLA|MPY|MPR|MPA|CDB|DVD|DVR|DVS|
                        CBD|BZE|BNZ|BPZ|BNG|BUX|BDX|BCHX|BCT|CALL|EXIT|
                        BFP|LDN|ADN|NGN|SBN|LDNC|ADNC|NGNC|SBNC|SLC|SLL|
                        SLA|SRC|SRL|SRA|SRAV|NORM|MVCH|SLC2|SLL2|SLA2|
                        SRC2|SRL2|SRA2|SRAV2|ANDN|ORN|ERN|LDCT|MOVE|SUM|
                        FLOAT|FAD|FSB|FMPY|FDVD|LFP

xparam              ::=octaldigit|xacc|xxacc

nparam              ::=simplecell|integer|octalinteger|charsequence|
                        pidentifier|lidentifier

simplecell          ::=xcell|rcell|xxcell|rrcell
```

15.2 PLAN Instructions in PLASYD

A function statement is the way that a machine code instruction can be compiled into a PLASYD program. The type of machine instruction is defined by the PLAN mnemonic preceded by ! symbol. The identifier of the mnemonic is not a reserved word and can be used as a normal identifier at any other part of the program. The identifier will have none, one or two arguments depending on the type of instruction. The two arguments which may be involved are the nparam which corresponds to the address and modifier fields of the instruction and the xparam which corresponds to the index field. If the xparam field has value zero then it can be omitted.

Therefore

!FAD(,LRX) !FAD(0,LRX) have the same meaning. The possible settings for the two parameters are:

(a) nparam. A function which is not a branch instruction may have as an nparam an unsigned integer number or octal value which does not exceed 12 bits (10 bits in the case of shifts) or a one or two character sequence. The binary representation of any of these quantities will be placed in the N field of the machine order. Alternatively the nparam may be a simplecell in which case the address of the cell will be placed in the N field of the instruction.

For branch instructions the N field may be either a label or the name of a procedure. The address of the statement so designated will be placed in the N field of the machine instruction. Alternatively, an integer or octal integer will be interpreted as an absolute address.

(b) xparam. An xparam may be an octal digit or an accumulator identifier. If it is an accumulator identifier, the corresponding octal number is assumed. For example:

```
!LDX(X7,LIX)      !LDX(7,LIX)
```

are the same.

Some typical examples of functions are:

```
!CDB(X3,UIV(X1)); !STO(0,7); !FAD(3,LRX);
!DISP('%%'); !BVCI(LBL); !FSB(,URA(X1+2)); !NULL;
!SLC2(X12,LIX);
```

A complete set of mnemonics with the meaning of the orders is given in the Appendix. The two functions !ACT and !RMS have only an nparam as X7 is assumed.

16.

CELL DECLARATIONS

16.1 Syntax

```
celldec        ::=xcelldec |rcelldec |xxcelldec |rrcelldec
αcelldec       ::=αtype αitemlist;  {α=x |r |xx |rr}
xtype          ::=INTEGER |LOGICAL
rtype          ::=REAL
xxtype         ::=LONG INTEGER
rrtype         ::=LONG REAL

αitemlist      ::=αitem |αitemlist,αitem {α=x |r |xx |rr}
αitem          ::=αidentifier |αidentifier=αinitial |αidentifier(integer)|
                  αidentifier(integer)=(αinitialList)   {α=x |r |xx |rr}

αinitialList   ::=αinitial |αinitial*integer |αinitiallist,αinitial |
                  αinitiallist,αinitial*integer  {α=x |r |xx |rr}

αidentifier    ::=identifier {α=x |r |xx |rr}
```

16.2 Naming of Cells

Before any cell is used it must be given an identifier as a name and the compiler must know its type. This is done by the cell declaration. As has been defined in Chapter 12, all declarations must appear at the beginning of a block before any executable statements. Each declaration consists of the symbol denoting the type followed by a number of items which will be given consecutive storage locations. Each item is separated from the next by a comma. Items can take one of the following forms:

(a) αidentifier: this specifies that sufficient storage is allocated for one item of the type specified to which the αidentifier given will be attached. INTEGER cells take up a single 1900 24-bit word. REAL and LONG INTEGER take up two 24-bit words while LONG REAL requires four 24-bit words. Examples are:

```
INTEGER A, B, C;
REAL D, E, F;
LONG REAL G;
LONG INTEGER ALONGIDENTIFIER, ANOTHERLONGONE;
```

If this set of statements appeared in a program then 17 1900 words would be allocated.

(b) αidentifier (integer): an identifier followed by an integer, N, in parentheses denotes an array of cells, N in number, for which storage has been allocated. The first cell may be referred to by the αidentifier

as though it were a simple identifier of the first type. The remaining members of the array are accessed by specifying it in one of the cell designations given in Chapter 6.
For example:

```
INTEGER AR(20),BR(10);
REAL CR(40);
LONG REAL DR(10);
```

The integer cells will use 30 1900 words, the real cells 80 and the long real 40. Note that the first element of the array AR can be accessed as AR(0) and the last by AR(19). Also there is no reason why AR(20) should not be used when referring to the cell BR(0).

(c) αidentifier=αinitial: this form is similar to (a) except that as well as giving the cell a name and defining its type, an initial value is allocated to the variable. In the first type (a), the initial value will be undefined. As variables at levels other than the outer block level share storage, initialisation of cell values only makes sense in the outer block level. Consequently initialisation is only allowed at the outer block level. The possible initial values of each type are the same as for the DATA statement described in Section 13.8. For example:

```
REAL A=1.7, B=1.38&MINUS 7, C=MINUS 37.5;
INTEGER D=7, E=#22, F='ABCD';
LONG REAL G=1.75L;
LONG INTEGER H=275D;
INTEGER I=@LIX, J=$LIX(3), K=£LIA(-1);
INTEGER L=CHAR 3,M=CHAR 2 OF @LIX;
INTEGER N=@SQUARE, M=@LBL;
```

The last example assumes SQUARE has been defined as a procedure and LBL as a label.

```
INTEGER P=6CNT+3, Q=17CNT+@LIX, R=5CNT+27T2;
INTEGER S=!FAD(3,LRX); T="ABCD";
```

(d) αidentifier(integer)=(αinitiallist): this form is similar to (b) except that as well as defining and naming an array of cells, it also sets initial values to some of the elements starting from the beginning. The αinitiallist is similar to the list defined for the DATA statement except that all items in the list must be of the same type. The form αinitial*integer is used to set the same value to a number of consecutive elements in the array (the integer defines the number to be assigned this initial value). The string, if used, can set the value of a number of consecutive cells. For example:

```
INTEGER AR(20)=(1*12,2*8), BR(10)=(5*7);
REAL CR(40)=(1.7*25,2.9,1.9*14);
LONG REAL DR(10)=(2.7L*10);
INTEGER ER(5)=("ABCDEFGHIJKLMNOPQRST")
INTEGER FR(2)=("A""B' DEF")
INTEGER GR(5)=(1,@LIX,CHAR3,@SQUARE,!FAD(3,LIX))
```

Note that in the first declaration only the elements BR(0) to BR(6) are set to 5. The remainder have no defined value initially.

The cell type INTEGER can also be written LOGICAL. The two are completely interchangeable. They have been included to aid in the differentiation between 24-bit quantities used as numbers upon which arithmetic operations are performed and 24-bit quantities used as a set of individual bits. However,no check is made that INTEGER declarations are used if arithmetic is to be performed and LOGICAL declarations are used if logical oprations are to be performed.

17.

SYNONYM DECLARATIONS

17.1 Syntax

```
cellsyndec   ::=xcellsyndec|rcellsyndec|xxcellsyndec|rrcellsyndec
αcellsyndec  ::=αtype αsynlist;  {α=x|r|xx|rr}
αsynlist     ::=αsyn|αsynlist, αsyn  {α=x|r|xx|rr}
αsyn         ::=αidentifier SYN fixedcell|αidentifier SYN (integer)|
                 αidentifier SYN (integer)=αinitial  {α=x|r|xx|rr}

accsyndec    ::=xaccsyndec|raccsyndec|xxaccsyndec|rraccsyndec
αaccsyndec   ::=ACC αaccsynlist;  {α=x|r|xx|rr}
αaccsynlist  ::=αaccsyn|αaccsynlist, αaccsyn  {α=x|r|xx|rr}
αaccsyn      ::=αidentifier SYN αacc  {α=x|r|xx|rr}
αfixedcell   ::=αidentifier|αidentifier(integer)|αidentifier(-integer)
                 {α=x|r|xx|rr}
```

17.2 Cell Synonyms

It is sometimes convenient to refer to a cell by more than one name, to give individual array elements names of their own or to treat a real array as two integer cells on occasions. It often increases the readability of a program if the same cell is used for more than one purpose.

A cell synonym declaration consists of the type of the new cells being declared followed by a list of individual synonyms. Each synonym consists of the identifier chosen for the new cell followed by the word SYN and the designator of the cell which is to be given the extra identifier. For example:

```
BASE;
INTEGER A(20),B,C(10);
REAL D(10),E,F;
INTEGER NEWB SYN B,C3 SYN C(2),H SYN C(-3);
INTEGER J SYN (3),K SYN (30)=#77000;
INTEGER L SYN E,M SYN E(1);
LONG REAL T SYN D;
```

In the first example NEWB is the name also given to the cell B. The cell designator must be a fixed cell which means that it can either be an identifier or an identifier followed by a fixed index. The second example sets the name C3 to the third item of the C array, C(2). If an index is used then it should not be such that it designates a cell in a different domain to the identifier. Thus:

```
INTEGER N SYN A(-1),P SYN A(4100);
```

are not allowed. Note that only fixed indexes are allowed.

For example:

```
INTEGER Q SYN A(X1);
```

is illegal. The identifier following the SYN must have already been declared by a normal or synonym declaration.
For example:

```
INTEGER A; INTEGER B SYN A; INTEGER C SYN B;
```

is allowed.

There is no reason why a cell of one type should not be given a synonym of a different type. In the last example, the integer identifiers L and M refer to the top and bottom halves of the cell E.

Cell synonyms may also be used to give names to absolute store locations. In this case only initial values can also be assigned as long as they do not affect the loading process. The examples above assign the name J to store location 3 which is X3. The second example gives store location 30 the name K and initialises it to have the value #77000.

17.3 Accumulator Synonyms

To assign extra identifiers to accumulators, the accumulator synonym declaration must be used. This is similar to a cell synonym declaration but the type is replaced by the symbol ACC and only accumulator identifier may follow SYN. Examples are:

```
ACC I SYN X1, J SYN X2, IJ SYN X3, SUM SYN A1, S SYN A12;
```

The new identifiers can then be used whenever the corresponding fixed accumulator identifier could be used. For example:

```
J:=LIX; SUM:=SUM+URX(X1+2);
```

18.

STORAGE ALLOCATION

18.1 Syntax

```
simpledeclist  ::=celldec|basedec|simpledeclist celldec|
                   simpledeclist basedec

basedec        ::=BASE;

lowerdec       ::=LOWER lwdeclist LOWEND;

lwdeclist      ::=celldec|lowerglobaldec|lwdeclist celldec|
                   lwdeclist lowerglobaldec

celldeclist    ::=celldec|celldeclist celldec
```

18.2 Upper Storage

Most PLASYD cells will be allocated upper storage. That is some address other than the first 4096 words of storage. Variables placed in upper storage cannot be directly addressed and must be accessed by indirect means. In earlier sections the method of accessing upper storage using a modifier has been described. Any set of declarations in a block head defined as a simpledeclist will be allocated upper storage. To enable such cells to be used, the upper storage is divided into arbitrary sections called domains which must not exceed 4096 words. An upper cell is then accessed by specifying the address of the first word in its domain called the base for that cell and the number of words (which must be less than 4096) between this base and the cell required. This number is called the displacement of the cell from its base. Given a cell identifier A, PLASYD defines the three addresses:

£A = base address of A, that is the address of the first word of the domain in which A appears.

$A = the displacement of A from its base.

@A = the actual address of A, @A=£A+$A.

The first upper declaration will cause a new domain to be set up and successively declared items will be assigned store locations in that domain in order until the next declaration would cause the size of the domain to exceed 4096 words. A new domain will then be started in the next available word. Each new block starts a new domain except that the block at the head of a procedure body is assumed to be part of the enclosing block's declaration.

If a single array item is more than 4096 words a new base will be started coinciding with its first word, and another new base will be started immediately after it. Any cells above the 4096'th will have to be addressed by a specially set up modifier.

If more than 4096 words are declared then a new base is defined and a comment to this effect is made on the program listing. Each base uses a word in lower storage to hold the address of the base of that domain. Each time a new domain is started, a word of lower storage is allocated at that point.

For example:

```
BEGIN
  INTEGER I(4095),J,K,L;
  INTEGER M(5000);
  INTEGER N;
  ..........
  BEGIN
    INTEGER P(4095),Q,R,S;
    ..........
    BEGIN
      INTEGER T,U,V(5000);
      PROCEDURE FRED(X1);
      BEGIN
        INTEGER PI,PJ(4000),PK(1000),PL(5000),PM;
        ..........
        END;
      END;
      ..........
      BEGIN
        INTEGER W,X(4000),Y(1000);
        ..........
      END;
  ..........
  END;
  ..........
  BEGIN
    INTEGER Z,A,B(4000),C(1000),D;
    PROCEDURE DICK(X1);
    BEGIN
      INTEGER E,F(5000);
    END;
  ..........
  END;
END
```

would allocate the following bases and domains

Cell	Lower Storage	Base Address	Displacement
I	0	ß	0
J	0	ß	4095
K	1	ß+4096	0
L	1	ß+4096	1
M	2	ß+4098	0
N	3	ß+9098	0
P	4	ß+9099	0
Q	4	ß+9099	4095
R	5	ß+13195	0

Cell	Lower Storage	Base Address	Displacement
S	5	β+13195	1
T	6	β+13197	0
U	6	β+13197	1
V	7	β+13199	0
PI	8	β+18199	0
PJ	8	β+18199	1
PK	9	β+22200	0
PM	9	β+22200	1000
W	6	β+13197	0
X	6	β+13197	1
Y	10	β+17198	0
Z	4	β+9099	0
A	4	β+9099	1
B	4	β+9099	2
C	11	β+13101	0
D	11	β+13101	1000
E	11	β+13101	1001
F	12	β+14103	0

Notice that 12 lower storage cells have been used to hold base addresses. The storage for variables P, Q, R and S is reused for the variables Z, A, B, C. Similarly, T, U, V, and W, X, Y share storage.

18.3 BASE Declaration

The user may not like the arbitrary way in which new domains have been allocated. The base declaration forces a new domain to be started at this point. By using the BASE statement, certain cells can be forced in the same domain if that is required. For example:

```
BEGIN
  INTEGER I(4095), J,K,L;
END
```

would normally allocate I and J in one domain with K and L in a separate domain. J can be forced into the same domain as K and L as follows

```
BEGIN
  INTEGER I(4095);
  BASE;
  INTEGER J,K,L;
END
```

18.4 Lower Storage

Although most cells are likely to be allocated upper storage, it may be advantageous to allocate lower storage to some cells. Such cells can be addressed directly and therefore do not have the overhead associated with upper cells where the base of the domain must be loaded into a modifier. A set of declarations are defined as having storage allocated in the lower area by placing the declarations between the symbols LOWER and LOWEND; to form a lower declaration. In the syntax given in Section 18.1 it can be seen that global declarations can also be defined as requiring lower storage. Details of global declarations are given later. Consider:

```
BEGIN
  INTEGER I,J;
  LOWER
  INTEGER K,L;
  REAL M;
  LOWEND;
  REAL N;
  LOWER
  INTEGER P=3;
  LOWEND;
  REAL Q;
END
```

Here I, J, N and Q are allocated upper storage while K, L, M and P are allocated lower storage.

Lower storage is used for purposes other than the holding of cells declared as lower. Consequently, less than 4096 words will be available. Lower storage is allocated as follows:-

a) locations 0 to 7 are the integer accumulators
b) locations 8 to 95 are reserved locations
c) constants are allocated space in lower storage
d) a location is allocated for each domain of upper storage defined and contains the base address of the domain.

19.

SUBCOMPILATION AND GLOBAL STORAGE

19.1 Syntax

```
segment                 ::=program|proceduredec
program                 ::=statement
globaldec               ::=GLOBAL gldeclist GLOBEND;
gldeclist               ::=blockidentifier : simpledeclist|
                            gldeclist blockidentifier:simplediclist
label                   ::=gl? lidentifier:|ENTRY digit:
proceduredec            ::=gl? PROCEDURE pidentifier(xacc ret?);
                            statement;
gl                      ::=GLABEL
externaldec             ::=EXTERNAL externallist;
externallist            ::=spec|externallist, spec
spec                    ::=pidentifier|lidentifier|pidentifier(xacc)|
                            lidentifier(xacc)
blockidentifier         ::=identifier
topglobaldec            ::=TOPGLOBAL blockidentifier:simpledeclist GLOBEND;
lowerglobaldec          ::=GLOBAL lowergldeclist GLOBEND;
lowergldeclist          ::=blockidentifier:celldeclist|
                            lowergldeclist blockidentifier:celldeclist
puredec                 ::=PURE storagedeclist PUREND;
storagedeclist          ::=storagedec|storagedeclist storagedec
storagedec              ::=celldec|basedec|lowerdec|globaldec|topglobaldec|
                            lowerglobaldec
nonplasyddec            ::=NONPLASYD externallist;
nonplasydstatement      ::=pidentifier|lidentifier|pidentifier(integer)|
                            lidentifier(integer)
```

19.2 Subcompilation

So far PLASYD programs have been considered as though they had to be compiled in one run. The program, apart from the trivial case of a single statement, would consist of a block as shown in the examples of earlier sections. It is however possible to split a program into a number of segments which may be compiled separately and then consolidated to form a program.

Each segment must be either a master segment which takes the form of a block, or a procedure which takes the form of a procedure declaration. Any complete program must contain at most one master segment and zero or more procedure segments.

Unless specifically declared, all identifiers will be local to the segment in which they are declared. The same identifier may be used for different purposes in different segments. Communication of variables between segments is provided by the GLOBAL declaration. Indicating which procedure calls refer to other segments and the link accumulator involved, is provided by the EXTERNAL declaration.

19.3 Global Names

Since identifiers are normally local to the block in which they are declared or at most, global to the segment being compiled, a special declaration is needed to indicate that a procedure name or label may be accessed from a separately compiled segment. This is done by preceding the label definition or the procedure declaration by the symbol GLABEL (global label).

For example:

```
GLABEL OUT: X:=X+X3;

GLABEL PROCEDURE FRED(X1);
X:=X+X3;
```

The GLABEL qualifier will ensure that the compiling system will keep a record of the identifier concerned for use in other segments. The name of a separately compiled procedure is automatically a global label and the symbol GLABEL is not required in this case.

A global label or procedure name may be called or GOTO'd from any external segment in the same way as a local label or procedure name could. However each separately compiled segment must inform the compiler of the external global labels that it is using and, if it is to call any of them as procedures then the link accumulator to be used must be specified. This is done by the EXTERNAL declaration. For example:

```
EXTERNAL OUT,IN,FRED(X1),IGNATIUS(X4);
```

The EXTERNAL declaration specifies which identifiers out of the set of global labels will be used by this segment. If an identifier does not appear in an EXTERNAL declaration then it can be used as a local identifier in spite of the fact that it may be declared as a global label in another segment of the program. This interchangeability between labels and procedures applies to global identifiers in the same way as it does with local ones. Labels of procedures must be declared as EXTERNAL before they are used.

An example of a three segment program might be:

```
[SEGEMENT 1]
BEGIN
EXTERNAL FRED,DICK(X1),HARRY(X1);
PROCEDURE LCL(X1);X0:=0;
GLABEL PROCEDURE TOM(X1);
BEGIN
  X6:=0;
  GOTO ND;
END;

HARRY;
GOTO FRED;
GLABEL SID:X4:=0;
GOTO DICK;
ND:X7:=0;
END
```

```
[SEGMENT 2]
PROCEDURE HARRY(X1);
BEGIN
  X2:=0;
END;

[SEGMENT 3]
PROCEDURE DICK(X1);
BEGIN
   EXTERNAL SID,TOM;

   X5:=0;
   GOTO TOM;

GLABEL FRED:X3:=0;
      GOTO SID;
END;
```

This three segment program sets the integer accumulators X2 to X7 to zero in order. Notice how procedure DICK is called using a GOTO statement. The EXTERNAL declaration may indicate a link accumulator for a global label even if it is not used. The procedures HARRY and DICK are assumed to be global as they have been compiled as separate segments.

A procedure or label defined as global may also be used locally.

19.4 Entry Points

Every program must contain at least one entry point at which the program can be started. This is done by an entry label and this may precede any statement in the same way as a label. For example:

```
BEGIN
  INTEGER I,J,K;

  X1:=0;
ENTRY 1:X2:=0;
ENTRY 9:X3:=0;
END
```

The particular entry point needed during any particular run of the program is specified to the operating system by reference to this digit. A maximum of 10 entry points are permitted for the program. Each entry point is distinguished by the numbers 0 to 9. It is necessary to insert a space between the symbol ENTRY and the following digit.

19.5 Global Variables

The simplest method of communication between a set of segments is to have a set of variables which are global or common to them all. Accumulators are automatically defined as being common to all segments. Cells are defined as common by using the GLOBAL declaration which consists of a set of cell declarations enclosed between the words GLOBAL and GLOBEND. Immediately following the word GLOBAL and before the first declaration, an identifier name must be given followed by a colon. This name is the one given to the global storage area defined by the GLOBAL declaration.

For example:

```
GLOBAL
ABC:
REAL A(10),B,C;
INTEGER I,J,K;
GLOBEND;
```

defines a global area consisting of 27 locations which have been given the name ABC. Several global areas can be defined in each segment either by defining each separately or by enclosing them all in the same brackets. For example:

```
GLOBAL
DEF:
INTEGER I,J;
GLOBEND;

GLOBAL
GHI:
INTEGER K,L;
GLOBEND;
```

defines two global areas with names DEF and GHI. These could equally well have been defined by:

```
GLOBAL
DEF:
INTEGER I,J;
GHI:
INTEGER K,L;
GLOBEND;
```

All segments having a global declaration with the same name can access the cells defined in the declaration. For example, a second segment might have a global name ABC defined as follows:

```
GLOBAL
ABC:
REAL X(5),Y(6),Z;
INTEGER M(3),N(3);
GLOBEND;
```

This defines a global area of 30 locations with the name ABC. The actual length of the area allocated to ABC is the maximum of all the global declarations with that name. If the two declarations above for ABC were the only ones that appeared, then a global area of length 30 would be allocated. It is important to notice that the name itself is all that need be the same. The individual declarations themselves may be quite different. Locations are allocated to variables as they appear in the declaration. In the example the first location of ABC has the name A(0) in the first segment and X(0) in the second. As the second declaration defines an area which is 3 locations longer than the first, the array N does not have any equivalent in the first segment. Examples of cells that do share the same location are:

```
[A(4), X(4)]   [A(5), Y(0)]   [B, Y(5)]   [C,Z]

[I, M(0)]   [K, M(2)]
```

The declarations above have been defining global areas in upper storage. It is possible to include BASE declarations within the set of upper cell declarations if they are required. GLOBAL declarations may also define areas of lower storage by enclosing the cell declarations between the brackets LOWER and LOWEND. For example:

```
LOWER
GLOBAL
ABC:
REAL X(5)
GLOBEND;
LOWEND;
```

defines an area of lower storage which is 10 locations in length and has the name ABC. Users should ensure that all global declarations with the same name in a program are either all upper declarations or all lower. It is possible to have the GLOBAL declaration outside the LOWER.

Cells defined as global may be initialised in the same way as any other cell. However, it is sensible to define the initial values in only one segment for a particular global declaration. If two segments have different initial values set for the same cell in a global declaration, then the actual value taken is undefined.

The names given to global areas must be distinct from those given to cells, procedures or labels in that segment. For example:

```
GLOBAL
ABC:
INTEGER ABC;
GLOBEND;
```

is not allowed.

It is sometimes necessary to define one global area which is located at the end of the allocated storage and can be of any length up to the limit available on the computer or allowed by the operating system. This is defined in PLASYD by a special TOPGLOBAL declaration similar in form to the standard GLOBAL declaration. In all segments comprising the program, there should only be one global area defined as TOPGLOBAL. If this global area is declared in several segments, then each declaration should be given as TOPGLOBAL. For example:

```
TOPGLOBAL
ABC:
INTEGER I(1);
GLOBEND;
```

defines ABC as TOPGLOBAL. Although only one storage location has been defined in ABC, as it is the last area allocated, the user may effectively let the array I be of any length. This is particularly useful for stack organisation where the actual depth of the stack is unknown until run time. The method of implementation of TOPGLOBAL is different under GEORGE 3 and GEORGE 4. Sparse programs under GEORGE 4 have TOPGLOBAL allocated at a high quire boundary (512K) while GEORGE 3 and GEORGE 4 dense programs allocate it immediately following the other allocated storage areas.

19.6 PURE Declaration

An area of storage can be designated as PURE by enclosing the declarations defining the area inside the brackets PURE and PUREND. For example:

```
PURE
LOWER
INTEGER A=1, B=2;
LOWEND;
GLOBAL FRED:
REAL C= 3.0;
GLOBEND;
GLOBAL XYZ:
LOWER
INTEGER D=5, E=7;
LOWEND;
GLOBEND;
PUREND;
```

This defines three distinct areas of storage, the lower area holding A and B; the upper global area called FRED and the lower global area called XYZ. Each area is defined as PURE which means that at consolidation time the areas will be marked as not requiring to be written to. On a paged 1906A, this does mean that the relevant pages can be marked as read only and this will stop any unexpected writing. Unfortunately, the current algorithm adopted by the consolidator is only to separate out pure and impure parts if it does not cause any additional 1024 word pages to be defined. For example, if the total impure lower storage was 500 words and the pure lower storage defined was 50 words, then both would be put in the same page. However, if both areas had been 700 words long, then two pages would be needed and the consolidator would then separate the pure and impure parts. As can be seen from the example, there is no point in defining pure storage which has not been initialised. Its main use is for storing preset tables.

19.7 NONPLASYD Declaration and Statement

The NONPLASYD declaration and statement allow a PLASYD program to call segments with multiple unlabelled entry points. The declaration has a similar form to the EXTERNAL statement and approximately the same meaning except that an entry to the label or procedure may be made at an address displaced from the entry point itself. Unlike the standard procedure or GOTO statement, the NONPLASYD version follows the identifier with an integer which defines the displacement from the entry point where the procedure is to be entered. For example:

```
JACK(3);
```

will enter the separately compiled procedure JACK at an instruction three after the entry point for JACK. Similarly:

```
GOTO JACK(4);
```

jumps to procedure JACK at a position four instructions past the entry point.

The bracketed integer may be omitted, in which case it is assumed to be zero. Thus:

```
JACK(0);
JACK;
```

are equivalent procedure calls.

DEFINE DECLARATIONS, CONDITIONAL COMPILATION AND INCLUDE STATEMENTS

20.1 Syntax

```
definestat    ::=DEFINE deflist;
deflist       ::=identifier=defexpr|deflist, identifier=defexpr
defexpr       ::=defprim|defprim aop defbasic
defprim       ::=defbasic|$fixedcellidentifier|@fcd-@fcd|
                  fcellidentifier-fcellidentifier|@instruction
                  identifier-@instruction identifier
aop           ::=+|-|*|/
defbasic      ::=integer|identifier (where identifier has previously
                  been defined)
fcd           ::=identifier|identifier(integer)|identifier(-integer)

conditcomp    ::=?integer(section of program)?integer
                  (where the two integers are the same and in the
                  range 1-10 inclusive.  There must be no spaces
                  between the ?, the integer and the parentheses).

includestat   ::=INCLUDE(subfilename)  |INCLUDE(subfilename, SL)

subfilename       ::=identifier
instridentifier   ::=lidentifier|pidentifier
```

20.2 Defined Identifiers

The define statement provides a means of associating an absolute integer value with an identifier at compile time. An identifier defined in this way may then be used at any point at which the corresponding integer value would be valid.
NB. The identfier does not label a storage cell, and cannot therefore be assigned to or altered in any way by the program. For example, consider the following section of code:

```
LOWER INTEGER TABLE1(1Ø)=(Ø*1Ø),TABLE2(1Ø);
    --------------------
FOR X1:=ØSTEP 1 UNTIL 9 DO
BEGIN X6:=TABLE1(X1);TABLE2(X1):=TABLE2(X1)+X6;END;
```

If the size of the tables is later altered, at least four alterations must be made to the program code. If, however, the code was:

```
DEFINE N=1Ø,P=N-1;
LOWER INTEGER TABLE1(N)=(Ø*N),TABLE2(N):
    --------------------
FOR X1:=ØSTEP 1 UNTIL P DO -----
```

only the DEFINE statement would need to be altered.

The forms of expression which may be used in DEFINE statements are:

(a) A literal value or a previously defined identifier, for example:
DEFINE A=3,B=#1ØØCNT,C='XYZ',D=B;

(b) An absolute expression consisting of:
(i) the displacement of a fixed cell, for example:
DEFINE E=$TABLE1(1),F=$TABLE2;
the value given is the displacement of the cell considered as a number of words.
(ii) the difference between the bases of two identifiers or address of two fixed cells, for example:
DEFINE G=£TABLE2-£TABLE1;H=@TABLE2-@TABLE1(3);
each pair of cells must be both in upper or both in lower storage, also both must be in the same common block or neither must be in common.
(iii) the difference between the addresses of two instruction identifiers, for example:
DEFINE I=@SIN-@COS;
where SIN and COS are either procedure or label identifiers.
All identifiers used in types (i), (ii) and (iii) must have been previously declared. The value given in types (ii) and (iii) is the difference between the two addresses considered as a number of words.

(c) Any of the forms given in (a) and (b) followed by any number of operator/operand pairs. The permitted operators are:
+ - * /
and the operands must be literals or previously defined identifiers. The operations are carried out from left to right, and all intermediate and final results must be single length integer values, for example:
DEFINE J=A+1/2*'?';
In this case, if (A+1)/2 is not an exact integer it will be rounded down.

20.3 Compile Time Control over Code Generated

Sections of code can be included in a program, and either compiled or ignored according to the setting of bits in the switch word.
For example, the line of code:

```
?1( X4:=A ?2( +B ?3( -C )?3 )?2 ?3( *D )?3 ; )?1
```

will have the following effect:

Switch 1	Switch 2	Switch 3	Code Compiled
off	on/off	on/off	none
on	off	off	X4:=A;
on	off	on	X4:=A*D;
on	on	off	X4:=A+B;
on	on	on	X4:=A+B-C*D;

Note that the scopes of conditional brackets may be nested and may even overlap, but that in the case of overlapping scopes the effect achieved may not be that intended.

For example, it is permissible to write:

```
?1(JOE;?2(SID;)?1FRED;)?2
```

but the effect is as follows:

Switch 1	Switch 2	Code Compiled
off	off	FRED;
off	on	FRED;
on	off	JOE;
on	on	JOE;SID;FRED;

20.4 Use of Text-form Libraries

The use of the INCLUDE statement, which must appear on a line by itself, causes the contents of the specified subfile to be compiled as if they appeared at that point in the program. The contents of the subfile may be any number of statements or declarations, or even parts of statements. These may not include further INCLUDE statements. If the parameter SL is given, the subfile code is shortlisted, if not it is listed as if it were part of the main program file except that the lines will not be numbered.

A situation in which the INCLUDE facility would be useful is where a large global declaration is required in several segments. In this case the global declaration could be placed in a subfile and included wherever required. If INCLUDE statements are to be used in a program then a MACROFILE or PUBLICFILE directive must be given, and the file containing the text to be included must be assigned to the compiler. Note, if the macrofile is empty this will cause the compiler to fail.

Source text can be placed in subfile form in direct access files by use of the utility program #XMED, which is described in the Compiling Systems Manual. Note that there is no subfile parameter which specifies the language PLASYD. The parameters *FORTRAN, *EMA, or *ALGOL can be used instead. The program #XMED may be found in a file entitled :SUBLIB.PROGRAM XMED, and an example of a simple job using it is given in Appendix 4.

21.

COMPILER DIRECTIVES

21.1 Syntax

```
file          ::= filename|filename (integer)
subfile       ::= subfilename|subfilename (integer)
subdescrip    ::= file. subfile|file|. subfile
```

The integer is, in each case, the generation number of the file or subfile.

```
sendto        ::= SENDTO (file. subfile)|SEND TO (file. subfile)
addto         ::= ADDTO (file. subfile)|ADD TO (file. subfile)

dumpon        ::= DUMPON (file)|DUMP ON (file)

libfile       ::= LIBRARY (file. subfile)
semifile      ::= SEMICOMPILED (file. subfile)

nameline      ::= NAME (name)
```

where 'name' is the four character name of the program

```
trusted       ::= TRUSTED (integer)
priority      ::= PRIORITY (integer)
progmode      ::= PROGRAM MODE (pmodedescrip)
pmodedescrip  ::= mdα, mdβ, mdδ|mdα, mdβ|mdα
```

where α β δ = a b c in any combination

```
mda           ::= 15AM|22AM
mdb           ::= DBM|EBM
mdc           ::= 15CH|22CH|MIXAM

updown        ::= UPPER, LOWER|LOWER, UPPER|UPPER|LOWER

progeven      ::= PROGEVEN (updown)
ignore        ::= IGNORE (updown)

segments      ::= SEGMENTS

namelist      ::= name, namelist|name
overlay       ::= OVERLAY (integer, integer) namelist
overcommon    ::= OVERCOMMON (integer) namelist

segmode       ::= SEGMENT MODE (smodedescrip)
smodedescrip  ::= mdα, mdβ|mdα
```

ere α β = d e in either order

```
mdd           ::= 15AM|22AM|MIXAM
mde           ::= DBM|EBM|MIXBM
```

```
segeven        ::= SEGEVEN (updown)| SEGEVEN
smomacro       ::= SMOMACRO
compilesmo     ::= COMPILESMO

localseg       ::= LOCAL SEGMENTS
globalseg      ::= GLOBAL SEGMENTS
local          ::= LOCAL  localtail
localtail      ::= ALL |: namelist |ALL BUT:namelist

macrofile      ::= MACROFILE (file)
publicfile     ::= PUBLICFILE (file)

readfrom       ::= READ FROM (subdescrip)

listlevel      ::= LIST|SHORTLIST|NOLIST
tabset         ::= TAB integer
switch         ::= SWITCH(numberlist)
numberlist     ::= integer, numberlist|integer
```

21.2 Program Description

Program description directives occur before the first segment of the program and may not be repeated unless this is specifically permitted in the description of the individual directive. Their effect lasts until the end of compilation. The directives described in the present chapter will be adequate for the majority of programs. Descriptions of less commonly used directives will be given in an appendix.

21.2.1 SENDTO

This directive is used to specify the output file and subfile to which the semicompiled generated by the PLASYD compiler, is to be sent, and there must be one such directive. The filename given is irrelevant when the compiler is run under GEORGE.

21.2.2 SEMICOMPILED and LIBRARY

The SEMICOMPILED and LIBRARY directives specify files containing previously compiled segments, which are to be consolidated with the segments currently being compiled. If a SEMICOMPILED directive is used, all the segments in the named file will be consolidated. If LIBRARY, only those segments which have been referred to by earlier segments will be consolidated. There may be any number of LIBRARY and SEMICOMPILED directives, subject to the consolidator limitation of twelve input files. This number includes the file currently being processed. The filename specified in the SEMICOMPILED and LIBRARY directives is irrelevant.

21.2.3 NAME

This directive specifies the four character program name of the binary to be produced from the source currently being compiled.

The characters must be alphanumeric and the first must be a letter. More than four characters may be given up to a maximum of twelve, but only the first four are significant. If this directive is omitted, a default name of XXXX is assumed. The program name serves no useful function, but it is printed on the compiler and consolidator listing and may therefore be of some use as a form of comment.

21.2.4 PROGRAM MODE

The address and branch modes in which a program is to be run are specified by means of a PROGRAM MODE directive. The address mode may be specified as 15AM or as 22AM. If it is not specified 15AM is assumed. The branch mode may be specified as DBM or EBM. If it is not specified, DBM is assumed. The consolidator address checking mode may be specified as 15CH, 22CH, or MIXAM, the latter meaning no checks. If it is not specified the specified or assumed address mode parameter is used. If the directive is omitted completely, the assumed settings are:

PROGRAM MODE (15AM, DBM,15CH)

15AM corresponds to the FORTRAN directive COMPACT DATA and 22AM to EXTENDED DATA. DBM corresponds to the FORTRAN directive COMPACT PROGRAM and EBM to EXTENDED PROGRAM.

21.3 Segment Description

SEGMENT DESCRIPTION directives are placed before any segment of a multisegment program or before the lone segment if there is only one. They may be repeated before subsequent segments, and unless stated to the contrary their effect lasts until they are reset by another directive of the same type, or until the end of the compilation.

21.3.1 MACROFILE and PUBLICFILE

These directives are used to specify the file containing library source texts, to be used by INCLUDE statements. If both MACROFILE and PUBLICFILE directives are given, the compiler will search for the required subfile in the macrofile before searching the publicfile. The filenames specified are irrelevant under GEORGE, as the operating system will provide the files required.

21.3.2 LOCAL SEGMENTS and GLOBAL SEGMENTS, LOCAL Directive

The LOCAL SEGMENTS directive informs the compiler that global labels and procedure names in subsequent segments will only be referenced from the overlay in which they appear. The GLOBAL SEGMENTS directive specifies that subsequent global labels and procedure names may be referenced from other overlays. The default setting is GLOBAL SEGMENTS and if this is not altered a substantial amount of overlay changing code will be incorporated, which will not be required in a non-overlaid program. The LOCAL directive specifies which of the global labels

in a segment following are to be referenced only from the same overlay and which external labels referenced from that segment are in the same overlay. There are three forms of this directive:

LOCAL ALL;	all labels in the same overlay
LOCAL: namelist;	specified labels in the same overlay
LOCAL ALL BUT: namelist;	labels other than those specified are in the same overlay.

If a LOCAL SEGMENTS directive is in force, all global labels must be declared local to their overlay. This may conveniently be done by preceding each segment with a LOCAL ALL directive. Note that a LOCAL directive only refers to the segment immediately following.

21.4 Program Source and Listing Control

The following directives are all optional and their effects last until they are reset by a directive of similar type, unless specified differently.

21.4.1 Compiler Listing Levels

There are three directives which may be used to specify the amount of output required from the compiler. The directive NOLIST will suppress all compiler output. SHORTLIST will cause those lines to which the compiler has added error messages or comments to be listed. LIST will produce full listing of the source code, with any error messages and comments, plus a detailed list of the core store required by each segment. The default setting is LIST.

21.4.2. Switch Bit Settings

If it is desired to use the conditional compilation facility as described in section 20.3, the appropriate bits in the switch word may be set by means of the SWITCH directive. If several bits are to be set at the same time, this may be done by giving a list of numbers within one SWITCH directive or by giving several directives. For example the following sequences will both set 2, 4 and 6 of the switch word.

(a)
```
SWITCH (2)
SWITCH (4)
SWITCH (6)
```

(b)
```
SWITCH (2,4,6)
```

Note that bits 0 to 10 may be used for controlling conditional compilation, but the bit 0 is also used to indicate that a listing of the semicompiled code produced by the compiler is required. Note also that the effect of switch directives only lasts until the end of the segment they precede.

22.

FORTRAN/PLASYD MIXED PROGRAMMING

22.1 Introduction

One of the main uses of PLASYD will be for inserting procedures into predominantly FORTRAN programs to improve the efficiency of the total system. It will be necessary to call each language from the other and it must be possible to pass values backwards and forwards between the two languages. To do this the interface between the two languages must be defined.

Creation of mixed language programs is relatively simple using the TASK system [3] which allows users to compile and run complex binary programs in a simple and efficient manner. For example if the following files are defined:
MAINF , a FORTRAN main program
SUBRF , a FORTRAN subroutine
SUBRP , a PLASYD subroutine

then the notation:

MAINF→SUBRP

can be used to define a program consisting of a PLASYD subroutine called by a FORTRAN main program. In its simplest form, the TASK system automatically inserts the necessary steering lines for consolidation in FORTRAN. However any PLASYD segments should contain a directive SENDTO (A.B) where both A and B are any identifiers (the TASK system will not use these names so that any unused name can be allocated). Some examples of TASK calls from a MOP console are:

(1) MAINF → SUBRF

```
TASK FORTRAN,*CR MAINF,*CR SUBRF
```

(2) MAINF → SUBRP

```
TASK PLASYD,*CR SUBRP,NOCONS,-
TASK FORTRAN,*CR MAINF,LINK
```

(3) MAINF → SUBRP → SUBRF

```
TASK PLASYD,*CR SUBRP,NOCONS,-
TASK FORTRAN,*CR MAINF,*CR SUBRF,LINK
```

In the mixed language programs, the first TASK call will compile the PLASYD subroutine leaving the semicompiled output in a workfile which is then consolidated with the FORTRAN routines by the LINK parameter. The FORTRAN library is scanned automatically so that library routines specified here will be added without any additional directives being necessary.

22.2 FORTRAN Subroutine Linkage

The method of subroutine linkage used in FORTRAN is quite similar to that defined for PLASYD in Chapter 13. Unlike PLASYD, the link accumulator must always be X1. It is always set to point to the instruction immediately following the call instruction itself. If the subroutine has no arguments then this will be the next executable instruction; otherwise it will point at the first of a set of instructio which define the arguments to be passed to the subroutine. Each argumen has a single instruction associated with it which loads information about the argument into X3. This will be, in most cases, just the addres of the argument. In some cases it does have certain marker bits set to differentiate between types of argument when more than one type is possible.

The simple FORTRAN subroutine call:

```
      CALL FRED
```

is exactly equivalent to the PLASYD procedure call:

```
FRED;
```

where the procedure FRED has been declared as:

```
PROCEDURE FRED(X1);
.........
```

If the FORTRAN CALL statement has I arguments then it is important to remember that the return address should be I instructions past the address given by X1. When setting up a PLASYD subroutine to be called from FORTRAN, the simplest way to ensure that the correct return positio is obtained is by including the second argument in the PLASYD procedure declaration. For example, a PLASYD procedure FRED which is to be called from FORTRAN with 3 arguments could be written:

```
PROCEDURE FRED (X1,3);
BEGIN
.........
RETURN (3);
.........
RETURN;
.........
END;
```

An exit from the PLASYD procedure by either of the RETURN statements or via the final END will return control to the correct position in the FORTRAN program (see Section 13.7 for further details).

It is possible that the same PLASYD procedure may be called from a number of places in a FORTRAN program with a different number of argumen in each place. If this is the case then there must be some way of indicating to the PLASYD procedure how many arguments are present so that the correct return position can be estimated. A standard technique is to have the first argument indicating how many arguments are to follow.

As indicated above, the code generated for a FORTRAN call involving arguments such as:

CALL FRED (J,B,E)

will have the general form:

```
FRED;
An instruction setting X3 to information concerning J
An instruction setting X3 to information concerning B
An instruction setting X3 to information concerning E
Next executable instruction
```

The possible arguments appearing in the CALL statement include scalars, constants, arithmetic expressions and arrays of type REAL, INTEGER, LOGICAL, COMPLEX and DOUBLE PRECISION. In addition, labels and subroutine/function names can be used. The standard method of getting at the information associated with an argument is to do an OBEY of the relevant instruction following the CALL statement. For example, if the routine FRED has been called, then:

```
OBEY (X1);    will set X3 to information concerning J, the first argument
OBEY (X1+1);  will set X3 to information concerning B, the second argument
OBEY (X1+2);  will set X3 to information concerning E, the third argument
```

In the following sections, the particular form of the information contained in X3 will be described.

The major difference between function and subroutine calls in FORTRAN is that the called routine must return a value in the case of function calls. In the 1900 compilers, this value is returned in a subset of accumulators. The particular accumulators used depends on the type of the function as follows:

TYPE OF FUNCTION	RESULT VALUE RETURNED IN
INTEGER	X6
LOGICAL	X6
REAL	A1
DOUBLE PRECISION	A1 (first two words) X4, X5 (last two words)
COMPLEX	A1 for real part X4, X5 for imaginary part

If functions are written in PLASYD which perform non-standard operations such as input/output of data then it is necessary to specify the function as ABNORMAL in the Program Description Segment.

22.3 Scalars, Constants and Arithmetic Expressions as Arguments

If the N+1 th argument of a subroutine call is either a constant, scalar or arithmetic expression then:

```
OBEY (X1+N);
```

will always set X3 to the address of the argument. The contents of this address will be the value of the argument. In the one case of a text constant, the value of X3 will be the address of the text constant, together with the #20000000 bit set.

The PLASYD equivalent of the FORTRAN subroutine:

```
      SUBROUTINE FRED (J,B,E)
      LOGICAL B
      J = 3+J
      B = .TRUE.
      E = E+4.0
      RETURN
      END
```

would be:

```
PROCEDURE FRED (X1,3);
BEGIN
OBEY (X1);                          [sets X3 to address of J]
X0 := (X3) +3;
(X3) := X0;
OBEY (X1+1);                        [sets X3 to address of B]
X0 := 1;                            [set X0 to value true]
(X3) := X0;
OBEY (X1+2);                        [sets X3 to address of E]
A1 := (X3) +4.0;
(X3) := A1;
END;
```

The LOGICAL variable is set to 1 to indicate .TRUE. and would have been set to 0 if the value .FALSE. had been required. For each type of argument, if X3 is set to α by the OBEY instruction then the form of the argument is as follows:

TYPE	SIZE IN WORDS	ARGUMENT
REAL	2	α,α+1 contains floating point number in standard PLASYD form
INTEGER	2	α contains signed binary integer in standard PLASYD INTEGER form while α+1 is set to zero
LOGICAL	2	α contains 0 for .FALSE. and 1 for .TRUE. while α+1 is set to zero
DOUBLE PRECISION	4	α,α+1 contains floating poin number in same format as for REAL while α+2,α+3 contains remaining part of mantissa. The whole is equivalent to t PLASYD LONG REAL format.

COMPLEX	4	α,α+1 contains floating point number in REAL format which is the real part of the complex number. α+2,α+3 contains floating point number which is the imaginary part of the complex number.

It is possible to define a compressed compilation mode for integer and logical quantities in FORTRAN. These are:

INTEGER (COMPRESS)	1	α contains integer in PLASYD form
LOGICAL (COMPRESS)	1	α contains 0 for .FALSE. and 1 for .TRUE.

In general this compressed mode is not used.

Arguments which are scalars may have a value passed back out of the routine as shown in the example above. If, however, the actual arguments are constant this can be quite dangerous as it will mean that the value of the constant in the calling routine has been overwritten.

22.4 Array Arguments

In a FORTRAN program, arrays are stored sequentially with the individual elements in consecutive storage locations. The number of locations required for each array element, say β, is the same as for the scalars defined above. The individual elements are stored in order such that the first subscript varies the most rapidly. For example:

```
    DIMENSION A(2,3)
```

defines an array of SIX elements stored in the order:

A(1,1) , A(2,1) , A(1,2) , A(2,2) , A(1,3) , A(2,3)

If the address of A(1,1) is α then the addresses of the other elements are α+β, α+2β, α+3β ..... For example, the address of A(1,2) is α+2β. If A is a real array then the address is α+4 while if it is a complex array then the address is α+8.

When passing array arguments to subroutines there are basically three different situations:-

(T1) the called routine expects a scalar argument and the actual argument is an array element. For example:

```
    SUBROUTINE FRED (A)
    A=3.0
    RETURN
    END
```

with a call of form:

```
    CALL FRED (B(3))
```

(T2) the called routine expects an array argument and the actual argument is an array name. For example:

```
SUBROUTINE FRED (A)
DIMENSION A(20)
A(1)=3.0
RETURN
END
```

with a call of form:

```
CALL FRED (B)
where B is an array
```

(T3) the called routine expects an array argument and the actual argument is an array element. For example the call of routine FRED in (T2) could have been:

```
CALL FRED (B(3))
```

As the calling routine has no knowledge of the argument expected, the form of the argument passed in cases (T1) and (T3) is identical. In both cases the address of B(3) is passed with, in addition, the #40000000 bit set. As this bit defines the character position in a word, the standard operations of loading and storing the contents of the address specified by X3 can be done without removing the bit. If more complex operations are to be done then it is good practice to remove the bit first. In some of the earlier1900 compilers, the bit #20000000 was used so that it is safer in fact always to remove the top two bits. For example a routine of type (T1) which needs to set the real array element argument to 3.0 would be:

```
PROCEDURE FRED (X1,1);
BEGIN
OBEY (X1);
A1 :=3.0;
(X3) :=A1;
END;
```

If the top bits are to be removed, the OBEY order could be followed by:

```
X3 :=X3 AND #17777777;
```

In cases (T2) and (T3), if the routine FRED called is in PLASYD, the most likely information required is the address of the first element. In case (T2) this is the address of the first element of B while in case (T3) it is the address of B(3) in the examples given above. In the case (T2) the information passed to X3 by the OBEY order is the address of the array header. This contains information such as the dimensionality of the array and the limits of each subscript. The PLASYD routine can therefore differentiate between the two types (T2) and (T3) by checking for the top two bits being set. If either is set then X3 contains the address of the first element; otherwise the value of X3 is the address of the array header. Array headers may have a variety of forms. However, in all cases the first location of the array is stored in the second word of the array header. A PLASYD routine equivalent to the example in (T2) would therefore be:

```
ROCEDURE FRED (X1,1);
EGIN
BEY (X1);
2 := X3 AND #60000000;
F X2=0 THEN X3 := (X3+1);
1 := 3.0;
X3) := A1;
ND;
```

f the PLASYD routine needs to access more information about the array
n calls of type (T2), then this can be obtained from the array header
sing the routine GETAH in the FORTRAN and PLAN libraries. The routine
ETAH requires two arguments. The first argument must be the address
f an area of store in the PLASYD procedure into which the information
s to be put. For an N-dimensional array, the length of this area must
e N+4 locations. The procedure FRED above, assuming it is only used
ith one and two dimensional array names as arguments, might have the
orm:

```
ROCEDURE FRED (X1,1);
EGIN
XTERNAL GETAH (X1);
OWER
NTEGER AHDR, INFOBLK(6);
OWEND;
BEY (X1);
HDR := X3;
ETAH;
3 := AHDR;
3 := @INFOBLK;
......
```

his procedure would set the array INFOBLK as follows:

```
NFOBLK(0)   =   the number of dimensions of the array
IFOBLK(1)   =   the number of words , β , per any element
IFOBLK(2)   =   the base address of the array, ie address of B(0,0)
IFOBLK(3)   =   the address of the first element of the array
IFOBLK(4)   =   the address of the first word past the end of the array
IFOBLK(5)   =   the first partial product P(1) = D(1)
```

ere D(I) is the number of elements in the Ith dimension.

the procedure had been called with an N-dimensional array name as
gument then:

```
OBLK(6) = the second partial product P(2) = D(1)*D(2)
....
OBLK(N+3) = the last partial product P(N-1) = D(1)*D(2)*...*D(N-1)
```

address of the arbitrary array element B(I1,I2,....IN)can be
culated from the above information by:

```
OBLK(2) + I1 + I2*P(1) + I3*P(2) + ... + IN*P(N-1)
```

library containing GETAH is automatically scanned by the TASK
em so that there is no need to specify it explicitly.

22.5 Subroutine and Function Names as Arguments

It is possible to have, in FORTRAN, a CALL of the form:

```
CALL FRED (JBC)
```

where JBC is the name of either a function or subroutine. For example the subroutine FRED might be:

```
SUBROUTINE FRED (XYZ)
CALL XYZ
RETURN
END
```

The call of FRED is therefore equivalent to calling JBC in the example above. The argument to the routine call in this case is an instruction which sets X3 to an instruction containing a jump to the routine JBC. Assuming the subroutine FRED above is replaced by a PLASYD procedure then this would have the form:

```
PROCEDURE FRED (X1,1);
BEGIN
LOWER
INTEGER LINK;
LOWEND;

LINK := X1;
OBEY (X1);        [set X3 to address of jump instruction]
X1 := @RET;       [set X1 to correct return address]
OBEY (3);         [executes jump to subroutine]
RET:
LINK := X1;
END;
```

If the routine called has arguments then a set of instructions following the label RET must be added which sets X3 to the required information for each.

22.6 Labels as Arguments

It is possible to define labels as arguments to FORTRAN subroutines in the 1900 dialect. For example:

```
SUBROUTINE FRED (*,*,X,Y,Z,*)
......
RETURN 1
......
RETURN 2
......
RETURN 3
......
RETURN
END
```

defines a subroutine with three label arguments. Statements of the

form RETURN i will return control to the ith label in the argument list. In the actual call to the subroutine FRED, labels are indicated by preceding them with an & symbol. For example:

```
    CALL FRED (&100, &2, X, U, V, &30)
```

indicates that control may return to labels 100, 2 and 30 as well as the instruction immediately following the CALL statement. Thus:

```
    RETURN 3
```

will return control to label 30 in the example.

The method of compilation adopted on the 1900 is that the CALL statement itself is equivalent to:

```
    CALL FRED (X, U, V)
```

and it is followed by the equivalent of:

```
    GOTO (1, 100, 2, 30) , X6
1   CONTINUE
```

where X6 is the integer accumulator. Before returning to the calling routine, the subroutine must set X6 to 0 for the normal return or 1,2,3 for the first, second or third label respectively.

For example, the FORTRAN subroutine:

```
    SUBROUTINE JBE (*,I,J,*,*)
    IF (I .EQ. 0) RETURN
    IF (J) 1, 2, 3
1   RETURN 1
2   RETURN 2
3   RETURN 3
    END
```

would be written in PLASYD as:

```
PROCEDURE JBE (X1,2);
BEGIN
OBEY (X1);
X0 := (X3);
X6 := 0;
IF X0=0 THEN RETURN;
OBEY (X1+1);
X0 := (X3);
X6 := 3;
IF X0 < 0 THEN X6 := 1;
IF X0 = 0 THEN X6 := 2;
RETURN;
END;
```

22.7 Passing Arguments via COMMON

A second method of passing arguments between one routine and another is using COMMON storage in FORTRAN. There is a direct equivalence between FORTRAN named common areas and PLASYD global areas (see Section 19.5). For example a FORTRAN routine having the declaration:

```
      LOGICAL B
      COMMON/XYZ/A,B,I,C(5),J(7,5)
```

defines an area of storage called XYZ consisting of 86 locations. The integer and logical quantities in FORTRAN are normally allocated two words of storage with the second not being used. A PLASYD procedure could communicate with this FORTRAN routine by defining a GLOBAL area as follows:

```
GLOBAL XYZ:
REAL A;
INTEGER B, BRUBBISH, I, IRUBBISH;
REAL C(5);
INTEGER J(70);
GLOBEND;
```

The fact that two words of storage are allocated for integer and logical variables in FORTRAN does mean that the equivalences are not immediately obvious. For example, the array element J(1,1) in FORTRAN is equivalent to both J(0) and J(1) in PLASYD where the actual integer value will be put in J(0) and J(1) will be set to zero. Similarly J(2,1) is equivalent to J(2) and J(3). and so on. It should also be noted that whereas the FORTRAN declaration for C defines elements C(1) to C(5), the PLASYD declaration defines variables C(0) to C(4).

Unlabelled FORTRAN COMMON is equivalent to a PLASYD global area with the name %. Thus the following two declarations are equivalent:

```
      COMMON A
GLOBAL %: REAL A; GLOBEND;
```

It may be necessary in some instances to provide communication between PLASYD procedures which is distinct from any named common areas defined in the FORTRAN part of the program (this is likely for library systems defined in PLASYD). In order to avoid name clashes, the PLASYD global areas should be given with the % symbol somewhere in the middle of the name. This will ensure that no clashes occur.

To generate more efficient code in the PLASYD procedures, it may be desirable to have the global area defined as lower storage. If this is done in the PLASYD procedure, then it will force the equivalent FORTRAN COMMON area to reside in lower to preserve the equivalence. Although the code in the FORTRAN routines will not take advantage of this, it may well be worth doing if the PLASYD procedures are heavily used.

If the user requires one COMMON area to be of undefined SIZE then this can be allowed by defining the equivalent PLASYD global area as TOPGLOBAL. Under GEORGE 3 this will ensure that this particular COMMON area is loaded last so that it can overrun its upper bound without overwriting other areas. The same effect is achieved under GEORGE 4 by placing the COMMON area at a high virtual store address for sparse programs.

22.8 Calling FORTRAN routines from PLASYD

From the previous sections it can be seen that the FORTRAN calling sequence is similar to the PLASYD one. The main point to remember is that the procedure call must be followed by the set of argument instructions which are single orders setting X3 basically to the address of the argument. There may be certain marker bits added in the case of text constants and array elements. Due to the complexity of the array headers, it is recommended that passage of an array to a FORTRAN routine is achieved by specifying an array element (type T3 of Section 22.4) rather than the array name itself.

The individual instructions setting X3 should not depend on them being executed in a standard order nor should they assume that the instruction is only OBEYed once. It is usual for the instruction to be OBEYed twice by the FORTRAN routine.

Although it is usual for the instructions setting X3 to be of the simple form:

```
X3 := @A;
```

where A is the argument, there is no necessity for this. For example, the instruction could be a subroutine call as long as any links are stored and restored correctly. A call of the FORTRAN subroutine FRED could be:

```
FRED;
GET1;
GET2;
GET3;
```

The routine FRED expects three arguments and the addresses of these are calculated by the subroutines GET1, GET2 and GET3 respectively.

As the FORTRAN subroutine will be compiled separately, there is a need to have an external declaration of the form:

```
EXTERNAL FRED (X1);
```

to ensure that the correct link, X1, is used.

22.9 Tracing

So far it has been assumed that the user is not interested in the fact that the PLASYD procedure has been entered for tracing purposes. It is as though the PLASYD procedure is part of the FORTRAN subroutine that called it. In order to provide a correct tracing information it is necessary to call a library procedure FPROLOG at the start of the PLASYD procedure and FEPILOG at the end. As well as setting up the correct tracing information, these routines also pass arguments from the FORTRAN routine to the PLASYD procedure and check for overflow. For example, a PLASYD procedure might have the form:

```
PROCEDURE FRED (X1;);
BEGIN
INTEGER TRC(4);
FPROLOG;
DATA ("FREDPLAS",3,@TRC);
.......
.......
FEPILOG;
DATA(@TRC);
END;
```

The arguments following the call to FPROLOG consist of an 8 character text string defining the TRACE name of the procedure, the number of arguments and the address of an array into which the arguments may be transferred. The example call given above expects the procedure FRED to be called with 3 arguments. The array TRC will be set as follows:

```
TRC(0)  contains link information to be used by FEPILOG
TRC(1)  contains address of first argument
TRC(2)  contains address of second argument
TRC(3)  contains address of third argument
```

The bits of the word not used in the address may or may not be zero. The routine will also handle a variable number of arguments if, in place of the second argument, a negative value is given. In this case the number given defines the maximum number of arguments excluding the first that can be passed. The first argument must define how many arguments are to follow. For example if FRED is called by:

```
      CALL FRED (2, A, B)
```

where 2 indicates that two arguments are to follow, and if it assumed that no call of FRED will take place with more than 5 arguments other than the first then the form of the prologue will be:

```
PROCEDURE FRED (X1);
BEGIN
INTEGER TRC(7);
FPROLOG;
DATA ("FREDPLAS", - 5, @TRC);
```

In this case the array TRC is set as follows:

```
TRC(0)  contains link information to be used by FEPILOG
TRC(1)  contains the first argument, i.e. the number of arguments following
TRC(2)  contains address of first argument
TRC(3)  contains address of second argument
.......
```

Note that in both cases the array TRC must be one greater than the maximum number of arguments expected including the count if it appears.

The routines FPROLOG and FEPILOG use X3 as a link accumulator. Both routines check for overflow and will give FORTRAN execution error 50. The call of FPROLOG should be the first instruction obeyed in the procedure so that correct tracing is given on such errors.

22.10 Peripheral Usage

One of the main uses of a PLASYD procedure may be to handle non-standard input/output requests. The user should remember that input/output operations for disc and tape will be buffered in FORTRAN and it is therefore dangerous to mix requests for the same stream from PLASYD and FORTRAN. In the case of cards and lineprinter output, there should not be any trouble in mixing, for example, reads from the same stream by routines in both languages.

23.

ALGOL/PLASYD MIXED PROGRAMMING

23.1 Introduction

It is indicated in Chapter 22 that PLASYD segments may be used in FORTRAN programs, and vice-versa. Similarly, it is possible to use PLASYD segments in an Algol program: it is not, however, generally possible to use Algol segments in a PLASYD program.

For example, if the following filestore files exist:

MAINA an Algol main program, which calls the subroutine SUBRP

SUBRP a PLASYD subroutine

then to compile and consolidate this program, the following TASK system calls are required:

```
TASK PLASYD, *CR SUBRP, NOCONS, -
TASK ALGOL,  *CR MAINA, LINK
```

23.2 General Points

The PLASYD segments must be declared in the Algol program: this declaration consists of the appropriate equivalent Algol procedure heading, followed by the basic symbol:

```
external;
```

Thus, a real procedure (that is, a function producing a real result) with two arguments called by value, one integer and the other logical (or Boolean in Algol 60 notation) would require the declaration:

```
real procedure fred (A,B);
value A,B;
integer A;
Boolean B;
external;
```

The PLASYD segments may use all Algol parameter types other than Algol procedures, and all the parameters other than expressions (which must be called by value) may be called either by name or by value.

The user should beware of mixing address and branch modes: either the PLASYD should be in the same address and branch modes as the Algol, or, preferably, the programmer should ensure that his coding is branch and address mode compatible (this corresponds to omitting the PLASYD statement SEGMENT MODE, or to using the form SEGMENT MODE (MIXAM, MIXBM)): in general, to accomplish this latter aim, the instructions BUX and BDX should not be used, and the instruction

BCHX should only be used when the count field (which will not be present in 22AM programs) is not significant. Such PLASYD constructions as

```
X2 := X2 + CHAR 1;
```

satisfy this.

If the PLASYD segment is to be a function, then the result should be left in a standard place, namely the accumulator X6 if it is of type integer or Boolean, or the floating point accumulator if it is of type real.

Note that in Algol, integer and Boolean (logical) variables always occupy one word each, and real variables always occupy two words. There are no double precision or complex types.

23.3 Link Accumulator, Picking up Scalar Parameters

The accumulator used for the link address is always X1, and the user should ensure that this is saved on entry and restored before exit.

The call instruction (in the Algol segment) is followed by n instructions (where n is the number of parameters), the ith of which enables the PLASYD segment to access the ith parameter by performing an OBEY on that instruction.

There are two pitfalls to beware of, otherwise incorrect parameter values may be obtained:

(i) if a parameter has been declared as being called by name, the contents of all accumulators, except X1, may be destroyed when the OBEY instruction for that parameter is executed. Hence, if the contents of any of the other accumulators are significant, they should be stored away before picking up the parameter.

(ii) if any of the parameters have been declared as being called by value, then the value of the corresponding actual parameter should be obtained and stored before any parameter called by name is accessed.

The PLASYD program sequence required to access a parameter depends on the type of the parameter, and is detailed below for each parameter type.

Integer

Integer variables are held in one word in 1900 Algol.

The OBEY instruction will give, in X3, the address of the word containing the parameter: thus the following will load the value of the third parameter of a procedure to X4:

```
BEGIN
   LOWER INTEGER LINK;
   LOWEND;
   LINK := X1;
   OBEY (X1+2);     [(X1+2) GIVES 3RD PARAMETER]
   X4 := (X3);
```

If the parameter is called by name, then its value may be returned by the PLASYD before exit, thus:

```
   X1 := LINK;       [RESTORE LINK]
   OBEY (X1+2);
   X6 := VALUE;      [LOAD AND RETURN VALUE]
   (X3) := X6;
```

Boolean

Boolean variables are held in one word: the value true is indicated by bit 23 of the word being a 1 (ie the word has the contents #00000001) and false by bit 23 of the word being 0 (ie the word has the contents #00000000).

Boolean parameters are picked up and returned in the same way as integer parameters.

Real

Real values are held in floating point form and occupy two words. The picking up of real parameters is similar to the picking up of integers, except that after the OBEY instruction, X3 will contain the address of the first of the two words containing the parameter. The LFP order may then be used to load the value to the floating point accumulator:

```
BEGIN
   LOWER INTEGER LINK; REAL REALNO;
   LOWEND;
   LINK := X1;
   OBEY (X1+1);     [OBTAIN 2ND PARAMETER]
   A1 := (X3);      [THIS GIVES LFP      0(3)]
```

Similarly, to return a real value to a parameter called by name :

```
   X1 := LINK;
   OBEY (X1+1);
   A1 := REALNO;
   (X3) := A1;
```

23.4 Array Parameters

The elements of an array are stored in consecutive units of core store; each unit is one word for integer and Boolean elements, two words for real elements. The order of the elements is such that the leftmost subscript varies the most rapidly, and successive subscripts vary less rapidly; thus the array :

```
A[1:2,1:2,4:5];
```

would be stored in the 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].

Information about an array is held in an array header, which is stored separately from the array. The format of these array headers is not documented; however, there is a routine, GETAH, available to translate these array headers into the information block defined below, and another routine, GENAH, will generate an array header from such an information block. The routines GETAH and GENAH are included in the SRA4 and GROATS libraries.

The information block produced by GETAH and required by GENAH has the following structure:

word 0 number of dimensions (n)

word 1 number of 1900 words per array 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-1)th partial product

Thus for an n-dimensional array, this block is of length $n+4$ words. If the array is one dimensional, only words 0-4 are present, and of these, probably word 3 (the address of the first word in the array) is the most useful.

This information block, and the method of obtaining it, applies identically for real, integer and Boolean arrays.

To use GETAH, it is first necessary to pick up the parameter corresponding to the array : this will give in X3 the address of the first word of the array header. This address has to be stored, and then used as a parameter to GETAH, as demonstrated in the following example (this assumes the array is two dimensional, and hence the information block will require 6 words):

```
BEGIN
   EXTERNAL GETAH (X1);
   LOWER
        INTEGER ADDRESS, LINK, AH(6);
   LOWEND;
   LINK := X1;
   OBEY (X1);   [IF  ARRAY IS FIRST PARAMETER]
   ADDRESS := X3;   [STORE ADDRESS OF ARRAY HEADER]
   GETAH;
   X3 := ADDRESS;   [NOTE TWO INSTRUCTIONS TO LOAD ADDRESSES]
   X3 := @AH;
```

This will produce the required information block in the area AH.

Similarly, if the user wishes to return an array header set up for a PLASYD data area, he should proceed as follows: firstly, set up the information block (as defined above) in the area AH (the length of AH should be appropriate for the number of dimensions the array is to have). Secondly, in the Algol segment, there should be declared an array of the same type and number of dimensions: it should, however, be small in size, as the array will be lost, for example:

```
real array  AR[1:1, 1:1];
```

Then, if this array is passed to the PLASYD segment as, say, the first parameter, the array header for the PLASYD area may be returned in place of the Algol array header by use of the routine GENAH, thus:

```
OBEY (X1);     [X1 CONTAINS LINK ADDRESS]
ADDRESS : = X3;
GENAH;
X3 : = ADDRESS;
X3 : = @AH;
```

Henceforward, all references to the array in the Algol segments will actually refer to the PLASYD data area.

23.5 Miscellaneous Parameter Types

Strings

The OBEY instruction for a string parameter will place, in X3, the address of the first word of a two word string header. This string header contains, in the first word, the number of characters in the string, and in the second word, the character address of the first character in the string. The string header may well be in a different area of core to the characters in the string.

In the following example, the string is the third parameter to the procedure:

```
BEGIN
     LOWER INTEGER LINK, STRINGCOUNT,
                   CHARADDRESS;
     LOWEND;
     LINK := X1;
      .
      .
      .
      .
     X1 := LINK;
     OBEY (X1+2);  [OBTAIN 3RD PARAMETER]
     X6 := (X3);   STRINGCOUNT : = X6;   [OBTAIN COUNT]
     X1 := (X3+1); CHARADDRESS : = X1;   [OBTAIN ADDRESS]
      .
      .
      .
      .
      .
```

Labels

The subroutine GOTOLAB may be used to jump to a label in the Algol segment from within a PLASYD segment. It is called as follows:

```
EXTERNAL GOTOLAB(X1);
  .
  .
  .
  .

GOTOLAB;
X3 := @M;
X3 := @N;
```

where M is the word containing the link address of the PLASYD segment, and N is a word containing the position of the label in the PLASYD procedure's declaration, i.e. the value n if the label is the nth parameter.

The routine GOTOLAB is included in both the Algol and the GROATS libraries. It uses (destroying the previous contents of) accumulators 1, 2 and 3.

In the following example, the procedure 'fred' has the declaration

```
procedure fred (I, J, LABEL);
integer I, J;
label  LABEL;
external;
```

The section of this segment associated with the label parameter would be of the form:

```
BEGIN
    LOWER INTEGER LINK, I, J, LABELNO = 3;
        [THE LABEL IS THE THIRD PARAMETER]

    LOWEND;
    EXTERNAL GOTOLAB(X1);
    LINK := X1;
      .
      .
      .
      .
      .
      .
      .
    GOTOLAB;                    [JUMP TO LABEL ]
    X3 := @LINK;
    X3 := @LABELNO;
      .
      .
      .
      .
      .
      .
      .
    X1 := LINK;                 [ NORMAL EXIT]
END;
```

Switches

The subroutine GOTOSW may be used to jump from a PLASYD segment to a switch in the Algol program. It is called as follows:

```
EXTERNAL GOTOSW(X1);
      .
      .
      .
      .
      .
GOTOSW;
X3 := @M;
X3 := @N;
X3 := @P;
```

where M is the word containing the link address of the PLASYD segment, N is a word containing the position of the switch in the PLASYD procedure's declaration (i.e. the value n if the switch is the nth parameter) and P is a word containing the switch subscript.

The routine GOTOSW is included in both the Algol and the GROATS libraries. It uses (and destroys the contents of) accumulators 1, 2, 3 and 6.

In the following example, the procedure 'chas' is assumed to have the declaration:

```
procedure  chas (A,SW1);
real  A;
switch  SW1;
external;
```

The section of the PLASYD segment associated with the label parameter would be of the form:

```
BEGIN
     LOWER INTEGER LINK, POSITION = 2, SUBSCRIPT;
         [THE SWITCH IS THE SECOND PARAMETER]
     LOWEND;
     EXTERNAL GOTOSW(X1);
     LINK := X1;
       .
       .
       .
       .
       .
       .
     SUBSCRIPT := X6;  [THE VALUE OF THE SWITCH SUBSCRIPT HAS
       .                  BEEN CALCULATED IN X6]
       .
       .
       .
       .
       .


     GOTOSW;
     X3 := @ LINK;
     X3 := @ POSITION;
     X3 := @ SUBSCRIPT;
       .
       .
       .
       .
       .
       .
       .
       .

      X1 := LINK;      [NORMAL RETURN]

END;
```

Procedures

It is possible to pass procedures of type external (i.e. not written in Algol) to a PLASYD procedure as parameters. Such a procedure is called from a PLASYD segment, by use of the subroutine PROC. The calling sequence for PROC is:

```
EXTERNAL  PROC(X1);
 .
 .
 .
 .
 .
PROC;
X3 :=  @M;
X3 :=  @N;
X3 :=  @ARG1;
 .
 .
 .
 .
 .
X3 :=  @ARGN;
 .
 .
 .
 .
```

where M is the address of the word containing the link address of the PLASYD segment, N is a word containing the position of the procedure in the PLASYD segment's declaration (ie the value n if the switch is the nth parameter) and ARG1, ARG2,....., ARGN are the arguments to the procedure to be called.

The routine PROC is included in both the Algol and the GROATS libraries. It uses (and destroys the contents of) accumulators 1, 2, 3 and 6.

In the following example, the procedure 'george' is assumed to have the declaration:

```
procedure  george(A,B,P);
real  A,B;
procedure  P;
external;
```

(The procedure P has one argument.)

The section of the PLASYD segment concerned with the procedure P would be of the form:

```
BEGIN
     LOWER INTEGER LINK, POSITION =3, ARG;
           [THE PROCEDURE P IS THE THIRD PARAMETER]
     LOWEND;
     EXTERNAL PROC(X1);
     LINK := X1;
       .
       .
       .
       .                          [THE ARGUMENT FOR P IS EVALUATED
       .                           AND STORED IN ARG]
       .
       .
       .
       .

     PROC;
     X3 := @LINK;
     X3 := @POSITION;
     X3 := @ARG;
       .
       .
       .
END;
```

Following the call to PROC, the procedure P will return to the point in this (PLASYD) procedure immediately following the parameters for PROC.

23.6 Accessing Algol Variables

The simple variables declared in the outer block of a master segment of an Algol program are stored in the lower common area ALGOL60 in the order in which they are declared, and may be accessed directly by a PLASYD segment. Arrays, and all variables at other levels of block nesting are stored elsewhere (usually on the stack) and are not available to the PLASYD segment except when used as parameters to procedure calls. The area ALGOL60 is initialised by the master segment at the start of an Algol program, and hence the variables in this area may not be given preset values by the PLASYD segments.

An Algol program uses a stack, which is the top common area (so that it can be extended if necessary). If a PLASYD segment requires an extensible area, then declaring it to be TOPGLOBAL and consolidating the segment into an Algol program will not work, since the consolidator, being offered two top common areas, will treat the second of them (normally the PLASYD TOPGLOBAL area) as an ordinary upper common variable area.

If the program is sparse, there is little difficulty: the PLASYD segment may be written with the extensible area missing, and when it is required, increase the active core size by 1K words (beware - the top bit of the accumulator will be set for GIVE/3 and GIVE/4 extracodes in a sparse program) and start addressing at some unused address, such as 1M = 1048576 (the Algol stack starts at address 512K = 524288) : provided the PLASYD was written using a variable to hold the base address of this area, then no problems need arise.

In the case of a dense program, again one method of proceeding is to write the PLASYD using a variable for the base address of the extensible area, which otherwise does not exist. Then when the PLASYD requires this area, it should increase the core size by 1K words and set the previous core size into the variable holding the base address. To ensure that the Algol stack will not be extended across the PLASYD area, an Algol program description statement

'SPACE' *n*

with a sufficiently large value of *n* should be used.

24.

USEFUL LIBRARY ROUTINES

24.1 Introduction

If PLASYD programs are run using the TASK system then the PLAN library is automatically searched for any external procedures that have not been included explicitly. Full details of the library procedures available are given in the relevant ICL manuals. Additional library routines appear in ACL's 1906A User Notes. In this chapter a few procedures are listed which should be particularly useful.

24.2 MONITOR

The aim of this routine is to print out areas of store during the running of the PLASYD program. The user may insert MONITOR requests at various points in his program. Whether or not printing actually takes place will depend on the current settings of the switches corresponding to bits 0 to 9 of word 30 of each program. The link used by the procedure MONITOR is X1 so that a declaration:

```
EXTERNAL MONITOR(X1);
```

should appear at the head of the program somewhere in the declarations.

A typical call of MONITOR would be as follows:

```
LINK:=X1;
MONITOR;
DATA(1CNT+502,50CNT+@Z);
```

Before the call of MONITOR the user must insert an instruction which stores the integer accumulator X1 into a lower variable. The areas of store printed depend on the parameters immediately following the call to MONITOR. These take the general form:

```
DATA(nCNT+m, cCNT+a, .....);
```

The parameters have the following meaning:

n = an integer in the range 0 to 511 which gives the number of cCNT+a parameters following.

m = usually a three digit number where the first digit defines the switch which will determine whether printing will take place and the remaining two digits can be used to identify the point in the source program from which the output comes. It is usual to have the m values for different calls of MONITOR unique. In the example given above, the number 502 identifies this MONITOR call and the output from it will only be printed if switch 5 is set on.

If a 1000 is added to the value of m then the contents of the integer accumulators which are normally output at the head of the MONITOR print will be omitted.

c = number of words of store to be printed out. The count must be less than 512.

a = address of first word of store to be printed. Using a number of parameters of this type, the one MONITOR can print several different areas of store.

The format of the printout is as follows:

(1) The heading
MONITOR PRINT m

(2) The contents of the integer accumulators X0 to X7 unless m is greater or equal to 1000

(3) The contents of the areas of store are printed one word per line. Each word is printed in a variety of formats preceded by its address. The formats are:

(a) The word in PLAN instruction form (F X A)
(b) The word as four characters
(c) The word as 8 octal digits preceded by #
(d) The word as a signed, zero-suppressed decimal integer
(e) The word as a signed decimal fraction.

Zero words are not printed; in their place **** is printed. Printing continues from the next non-zero word. Two newlines are output before and after the MONITOR printout with single newlines between the individual areas requested. The procedure is 338 words long including data areas used. None of the values of accumulators are disturbed by the procedure. If only the integer accumulators themselves are required, this can be achieved by setting n = 0. For example, arguments:

```
DATA(0CNT+504);
```

would just print the integer accumulators.

The best method of using MONITOR is to set up debug prints in groups controlled by different switch values. If the program has the name XYZ then it can be run from a MOP console by:

```
TASK PLASYD,*CR XYZ,#??(ON 5),#??(ON 3),#LP0
```

This would cause all MONITOR printing controlled by switches 5 and 3 to be printed. Modifying the TASK command to include additional ON commands would allow other printing to take place.

The MONITOR procedure only works in 15 bit address mode and direct branch mode. If programs require monitor printing in any other mode then the procedure MONITORX should be used. Output goes to the stream *LP0.

24.3 MONITORX

This procedure is very similar in format to MONITOR which was described in Section 24.2. The major differences are that it will work in any mode and the addresses to be printed are only listed in octal. Thus a valid call of MONITORX is:

```
LINK :=X1;
MONITORX;
DATA(1CNT+502, 50CNT+@Z);
```

The format of the printout is:

(1) The heading
MONITOR PRINT m

(2) The contents of X0 to X7 are printed on a single line followed by the letter V if V is set.

(3) The contents of the areas of store are printed in octal (as 8 octal digits) with eight words per line. Each line is preceded by the address of the first word in decimal. If a zero word occurs somewhere in a line it is printed. If the first word is zero then **** is printed. Printing in octal continues on a newline with the next non-zero word.

The other major difference between MONITOR and MONITORX is that there is a second form of the parameter defining the area to be output. As well as cCNT+a it is also possible to write a parameter as two separate words, for example:

```
DATA(1CNT+502, 50, @Z);
```

This is used when c is greater than 511 or the address 'a' is greater than or equal to 32K.

The procedure is 255 words long including data areas.

24.4 MONITORI

This routine is similar to MONITOR and MONITORX except that it is designed to print only those locations which have been changed in value since the last call of MONITORI for a specific point in the program. The format of the calls to MONITORI is the same as for MONITOR and the parameters have basically the same meaning.

The first time any MONITORI call is reached in the program the contents of the 8 integer accumulators will be printed unless 1000 has been added to the monitor identifying parameter. Subsequently, when the MONITORI call is reached, the accumulators will only be printed if their contents have changed since last being printed. Similarly, the contents of the areas specified will be output in full the first time a particular monitor point is reached. However, on

reaching the monitor point identified by 'm' subsequently, only those locations whose contents have changed since the last print of this monitor point will be printed.

The final values of all locations in the monitor prints can be obtained by the inclusion of a monitor point whose first parameter after the identifying parameter is zero. In this case, the print out of store locations will be the same as if all previous different monitor points were repeated at this point in the program (only those locations that have changed will be printed).

The format of the printout for each location is as follows:

(a) location address as a decimal number
(b) contents in character form
(c) contents in octal form
(d) contents as a zero suppressed decimal integer
(e) if contents of location are zero then only printed as four zero characters. A sequence of zero locations is not condensed to one print as in MONITORX and MONITOR.

Two monitor points with the same identifying parameter but different parameter lists will be regarded as being the same point if and only if the locations to be monitored at the second point are a leading subset of the locations to be monitored at the first point.

As the last locations printed have to be stated for comparison purposes, the user must set up a scratch area on disc. Also the output is defined to go to *LP7. A suitable TASK command to run a program in file FRED containing MONITORI commands would be:

```
TASK PLASYD,*CR FRED,#LP7,#??(CE!(*ED,BUCK!,KWORO)),-
#??(AS *ED7,!(WRITE)),#??(ON 5)
```

if monitor switch 5 was required on.

Alternatively the scratch file assignment could be put in a macro file giving:

```
TASK PLASYD,*CR FRED,#LP7,MAC
```

where a macro FRED-MAC was:

```
CE!(*ED,BUCK!,KWORO)
AS *ED7,!(WRITE)
ON5
```

The routine requires 610 words of storage including data areas.

24.5 Simple Input/Output Routines

It is often necessary for the user to do his own handling of either data input or output. Below are simple procedures for reading and writing lines of data.

```
PROCEDURE INLINE(X1);
BEGIN
LOWER
INTEGER INCTRL(4)=(3CNT,0,80,@INPUTBF);
INTEGER INPUTBF(20);
LOWEND;

!PERI(1,INCTRL);
X0 :=INCTRL(1);
IF X0<0 THEN !SUSBY(0,3);
END;

PROCEDURE OUTLINE(X1);
BEGIN
LOWER
INTEGER OUTCTRL(4) = (4CNT,0,80,@OUTPUTBF);
INTEGER OUTPUTBF(20);
LOWEND;
!PERI(0,OUTCTRL);
X0:=OUTCTRL(1);
IF X0<0 THEN !SUSBY(0,4);
END;
```

The first argument of the PERI in the procedure INLINE defines that input will be from unit number 1 and the control INCTRL indicates that the input is from the card reader. Thus the procedure INLINE will read the next card from *CR and place the 80 characters read into the locations INPUTBF(0) to INPUTBF(19), four characters to a word.

Similarly the procedure OUTLINE will pass the 80 characters contained in the buffer OUTPUTBF(0) to OUTPUTBF(19) to the stream *CP0.

24.6 Conversion of Integers to Characters

The following routine will convert a binary integer into a decimal or octal character representation, and place the characters in a buffer specified by the calling routine. The integer is placed in X4 before the call, and the address of the buffer in X2. Note that the address supplied is the address of the first character after the number in the buffer. The decimal conversion routine will deal correctly with positive or negative numbers, preceding them with a space or minus sign respectively. Note also that a number consisting of a one in the sign bit and zeroes elsewhere cannot be converted by the routine, and is therefore trapped as a special case. The use of the routine should be made clear by the following examples.

(1) Convert the value held in COUNT to decimal characters, and place in BUFFER and BUFFER(1). Leading zeroes are to be replaced with spaces.

```
X4:= '     '; BUFFER:= X4; BUFFER(1):= X4;
X2:= @ BUFFER(2); X4:= COUNT; OUTDECIMAL;
```

(2) Convert the value held in COUNT to octal characters, and place in BUFFER (3) and BUFFER(4). All eight digits are to be printed.

```
BUFFER(3):= 0 BUFFER(4):= 0 X2:= @BUFFER(5);
X4:= COUNT; OUTOCTAL;
```

(3) Convert the value held in TOTAL to decimal characters, and place in BUFFER(6) and BUFFER(7). TOTAL is known to be in the range 10000 to 99999, and it is desired to place the digits in character positions 1 - 3 of BUFFER(6) and 0 - 1 of BUFFER(7). Character 0 of BUFFER(6) will contain a space for the sign, and character 2 of BUFFER(7) is to contain a semicolon.

```
X4:= ' '; BUFFER(7):= X4; X2:=@BUFFER(7) + CHAR 2;
X4:= TOTAL; OUTDECIMAL;
```

```
LOCAL ALL;
PROCEDURE OUTDECIMAL (X0);
BEGIN
  LOWER
   LOGICAL STRING(2) = ("-8388608");
  LOWEND;

  [CALL WITH BINARY NUMBER IN X4]
  [X2 =ADDRESS OF CHARACTER AFTER NUMBER IN OUTPUT BUFFER]

  X1 := ' ';  X7:=10;
  IF X4 < 0 THEN BEGIN X4:= NEG X4; X1:= '-';
  IF X4 < 0 THEN GOTO MAXNEG;

  LOOP: X2:= X2 SLC 2 - 1 SRC 2; !DVS(3,X7);  (X2):= CH X3;
  IF X4 NOT= 0 THEN GOTO LOOP;
  X2:= X2 SLC 2 - 1 SRC 2; (X2):= CH X1;  RETURN;

  MAXNEG: X2:=X2-2; X1:= @STRING; !MVCH(1,8);  RETURN;

GLABEL OUTOCTAL: X2:= X2 SLC 2 - 1 SRC 2; X45:= X45 SRL 3;

  X5:= X5 SRL 21; (X2):=  CH X5;
  IF X4 NOT= 0 THEN GOTO OUTOCTAL;
END;
```

25.

Use of TASK macro to Compile PLASYD PROGRAMS

25.1 Beginners' Guide to TASK

This chapter is not intended to give a full description of the TASK macro, for which see the relevant manual, but to enable the user to compile and test a PLASYD program using, as far as possible, the default options of the TASK macro.

(a) TASK PLASYD, *CR PROG 1

This call of the TASK macro may be included in a job deck or macro definition, or may be typed on a MOP console. In the latter case the macro will run a background job and return control to the user. This job will send broadcasts back to the user to inform him of its progress, and will send its output to the lineprinter. It will attempt to compile the source file PROG 1 using the PLASYD compiler, and if this is successful, will attempt to consolidate the semicompiled. If this in turn is successful it will enter the binary. At this point the binary will have LP0 assigned to a workfile, which will be listed on the lineprinter. If the program requires data, this may be placed in a file called PROG 1-DAT, (sourcefile name plus -DAT,) which will automatically be assigned to CR0.

(b) TASK PLAS, *CR PROG 2, SAVE, NORUN

This is similar to (a), except that the resultant binary (if any) will be saved in a file called PROG 2-BIN, and will not be run. Note the abbreviation PLAS.

(c) TASK PLAS, *CR PROG 3, COMP, NOCONS

This call will produce a semicompiled file called PROG 3-SEM, and will not attempt to consolidate it. This form can be used if the sourcefile contains subroutines which are to be used by a FORTRAN program, for example. If the FORTRAN sourcefile is called MYPROG, it may now be compiled and tested, with the PLASYD subroutines, by the call.

TASK FORTRAN, *CR MYPROG, SEMI PROG 3-SEM, LINK

25.2 Use of Direct Access Sourcefiles

If the PLASYD facility PUBLICFILE is used, the required direct access file may be presented to the compiler by including the TASK parameter

*??(AS *DA12, PUBSOURCE)

where PUBSOURCE is the filename. If MACROFILE is used, the parameter is

*??(AS *DA13, *filename*)

and for READ FROM it is

*??(AS *DA14, *filename*)

Most users, however, should never need to use these facilities, as program source-code is more conveniently held in GEORGE serial files than in direct access files.

26 .

SMO CELL DESIGNATION

26.1 Syntax

```
smoαcell  ::=  (simplesmo) |(simplesmo+integer) |
               (simplesmo+modifier) |(simplesmo+modifier+integer)|
               αidentifier(simplesmo) | identifier(simplesmo+integer) |
               αidentifier(simplesmo+modifier) |
               αidentifier(simplesmo+modifier+integer) {α=x |r}
```

(but see Section 26.4)

```
simplesmo ::=  simpleαcell |£upperidentifier|blockidentifier
```

26.2 Addresses

This section describes the more complex formats for accessing the contents of storage cells. The syntax above defines an address of a storage location. The assignment statements defined earlier either deposit a new value into this location or access its contents. Unlike the 'simple α cell' designators, it is necessary to generate more than one instruction for each appearance of a 'smoαcell' in a statement. On some 1900 computers (including a 1906A) there is a special SMO hardware instruction which causes the next instruction obeyed to have its address modified by the contents of the word specified in the SMO instruction. On these computers it is therefore reasonably efficient to use SMO designators. On other computers in the 1900 range, a macro instruction has to be obeyed which is equivalent to between 3 and 5 instructions. It is recommended that SMO cell designation should only be used when the same effect cannot be produced using simple cell designators.

The three different forms of the SIMPLESMO define a value as follows:

(a) simplexcell : the value is the contents of the simplexcell

(b) £upperidentifier : the value is the base address of the domain containing the upper identifier specified

(c) block identifier : the value is the address of the first word of the global storage area with this name

If the value defined by the simplesmo is V, the value of the 'integer' defined in the syntax is I, the contents of the 'modifier' is X, and the displacement of the identifier is $ then the address specified in each case is:

(simplesmo)	V
(simplesmo + integer)	V+I
(simplesmo + modifier)	V+X
(simplesmo + modifier + integer)	V+X+I
αidentifier (simplesmo)	$+V
αidentifier (simplesmo + integer)	$+V+I
αidentifier (simplesmo + modifier)	$+V+X
αidentifier (simplesmo + modifier + integer)	$+V+X+I

The addresses calculated must have 0≤ $-I, I, $+I <4096

26.3 Some Example Assignment Statements

Consider the declarations:

```
LOWER
INTEGER LAA=@LB, LB=1, LC=@UA;
LOWEND;
BASE;
INTEGER UA=3, UB=@UA, UC=11;
GLOBAL FRED:
INTEGER D=5,E=9;
GLOBEND;
```

Then the following integer assignments statements produce the result given:

```
(a)  X2 := (LAA);          sets X2to 1, the  contents of @LB

(b)  X2 := (LAA(2))        sets X2 to 3, the contents of @UA

(c)  X1 := @LAA;
     X2 := ((X1));         sets X2 to 1

(d)  X3 := 2;
     X2 := (LAA(X3));      sets X2 to 3

(e)  X3 := £UB;
     X2 := (UB(X3))        sets X2 to 3

(f)  X3 := 1;
     X2 := (LAA(X3-1));    sets X2 to 1

(g)  X3 := 1;
     X2 := (LAA(X3+1)+2); sets X2 to 11

(h)  X2 := LAA(LAA(1)+1); sets X2 to @UA

(i)  X2 := E(FRED)          sets X2 to 9

(j)  X2 := UA(£UA)          sets X2 to 3
```

As the integer accumulators can also be accessed as the first 8 locations of store, it is possible to use integer accumulators other than X1, X2 and X3 as modifiers by specifying their store addresses. For example, X6 can be referred to as (6) so that in the same way X3 can be used as a modifier in U(X3), X6 can be used in U((6)). Double modification can be achieved by using designators of form ((6)+X2). To use an index consisting of contents of X1 and X2, the designators ((1)+X2) or ((2)+X1) can be used.

26.4 Errors in PLASYD Compiler

There are currently two errors in the PLASYD compiler involving SMO designators. Using the alternatives given below:

```
smoαcell     ::= (simplesmo)
simplesmo    ::= simplexcell
simplexcell  ::= loweraidentifier (-integer)
```

it should be possible to write statements of the form:

```
X2 := (LC(-2));
```

Statements of this type currently give a compiler error and generate incorrect code.

Similarly:

```
smoαcell     ::= (simplesmo +integer)
simplesmo    ::= simplexcell
simplexcell  ::= αidentifier (modifier -integer)
```

should allow a statement of the form:

```
X2 := (LAA(X3-1)+1);
```

This also generates incorrect code and gives a compiler error.

Apart from these two cases, the correct code is generated for the syntax given at the head of the chapter.

27.

COMPILER OUTPUT

27.1 Source Listing

The PLASYD compiler produces a full listing of the program's source text during the compilation unless the user inhibits it by inserting the NOLIST or SHORTLIST directive at the head of the program or segment being compiled. Each source line appears on the left hand side of the output listing. To the right of the source line can appear:

(a) the instructions generated
(b) any errors found

Only the first 72 characters of each PLASYD source line will be recognised and printed by the compiler. Care should be taken to ensure that no source text appears in higher character positions as it will be ignored and will appear not to exist from the listing.

If the directive:
SWITCH(0)
has been added at the head of the segment being compiled, then each line of text will be followed by the machine code instructions generated. The possible fields printed are:

```
   P   T   F   X   M   N   S      0
  66  PR  117  0   3   1  LP   04770001
```

where:
P = position in decimal of this instruction relative to start of segment

T = type of instruction generated. PR indicates an instruction in the program while LT defines a literal to be stored

F = function code of instruction

X = integer accumulator

M = modifier

N = address field

S = type of address in the instruction. The type PR indicates a program address relative to start of segment, LT indicates a literal and LP a lower preset address

0 = octal form of instruction generated.

Normally the instructions will be printed in ascending numerical order of PR. However, in the more complex statements, such as IF and FOR, it is possible for the skeleton instruction to be generated without the address and later reprinted when the address came available.

27.2 Error Diagnostics

At the right hand side of each program line will be indicated any errors that have occurred in the line. If more than one error has occurred, these will be printed one after the other on separate lines. For example:

```
INTEGER P(4000), Q(4000);
                                                    COMMENT 11
```

The types of diagnostics printed are:

(a) ERROR, the error will cause compilation to fail. It will be patched up so that compilation can continue to the end of the program in order that other errors may also be recognised. The complete set of errors are listed in Appendix I together with any assunptions made in order to patch up the program.

(b) COMMENT, although not an error this message will point out an unusual situation of which the user should be aware.

In the SHORTLIST form of listing, only those lines having errors will be printed together with the error messages.

At the end of the listing, the message:

```
ERRORS IN COMPILATION
```

will be printed if any errors have occurred.

27.3 Symbol Table Listing

If the full listing of the source has been generated, then a storage map will also be generated while the other two modes do not generate the map. By setting SWITCH(21), the reverse of the standard option associated with the listing can be defined.

The storage map consists of the following sections:

(a) SEGMENT
If the segment is a procedure, details are given in the form:

```
segmentname          link          n WORDS
```

If the segment is a master segment, details are:

```
MASTER SEGMENT OF programname  n WORDS
```

(b) GLABEL PROCEDURES
Details are given in the form:

```
procedurename        link          entry W
```

where W indicates the word of the segment at which the procedure starts

(c) INTERNAL PROCEDURES
Details are similar to (b)

(d) EXTERNAL
Details are given in form:

nameofitem link

(e) NONPLASYD
Details are similar to (d)

(f) DEFINEES
Details are given in the form:

nameofitem valueofitem

The value is written as 8 octal digits preceded by #

(g) LITERALS n WORDS

The first line is followed by n lines of form:

numberofliteral value r charvalue

The literals are listed in order of definition. The value of the literal is first specified as 8 octal digits and then as four characters. The field 'r' defines the appropriate two character relativiser if it is needed.

(h) GLABELS
Details are given in the form:

m glabelname

where m is the word of the segment at which the GLABEL occurs. The GLABELS are listed in ascending order.

(i) LABELS
Details as for (h)

(j) LOWER PRESET n WORDS
Details are given in the form:

m (t) symbolicname

where m is the position of the item, t is the type of the item (I = integer LI = long integer, R = real, LR = long real)

(k) PURE LOWER PRESET n WORDS
Details as for (j)

(l) UPPER PRESET n WORDS
Details as for (j)

(m) PURE UPPER PRESET n WORDS
Details as for (j)

(n) LOWER GLOBAL blockname n WORDS
Details as for (j). This format is repeated for each lower global block

(p) PURE LOWER GLOBAL blockname n WORDS
Details as for (n)

(q) UPPER GLOBAL blockname n WORDS
Details as for (n)

(r) PURE UPPER GLOBAL blockname n WORDS
Details as for (n)

(s) SYNONYMS OF ABSOLUTE ADDRESSES
Details are given in the form:

m (t) synonymname

where m is the absolute location and t is the type as in (j)

(t) ACCUMULATOR SYNONYMS
Details are given in the form:

accumulatorname synonymname

The absence of any items from (b) onwards is indicated by NONE following the name of the item.

PLAN INSTRUCTIONS NOT PROVIDED FOR IN PLASYD

There are a number of instructions in the 1900 order code which are found useful by programmers writing in PLASYD, but which can only be used by writing the PLAN instruction explicitly. The following sections describe some of the more commonly used of such instructions.

28.1 Counter-modifiers

Several 1900 orders permit an accumulator to be regarded as being composed of two sections:- an address field, occupying B9-23‡, (the lower 15 bits,) and a counter field, occupying B0-8. The following instructions act upon these fields.

```
!BUX(X,label);
```

Increment the address field of accumulator X by one, then decrement the counter field by one, and if this is not now zero branch to the address indicated by 'label'.‡

```
!BDX(X,label);
```

Similar to BUX except that the address field is incremented by two.

```
!BCHX(X,label);
```

For this instruction, the accumulator is considered to consist of a word address field in B9-23‡, a counter field in B2-8, and a character address field in B0-1. The instruction increments the character address by one, with overflow being added into the word address, then decrements the counter and branches if it is not now zero.

```
!BCT(X,label);
```

Decrement the address field of accumulator X by one, and branch to 'label' if it is not now zero.

‡If the program is running in 22AM, the address field occupies B2-23 of the accumulator, and there is no counter field. The instructions BUX, BDX and BCHX therefore update the address field as described, and branch unconditionally to the address indicated by 'label'. In this address mode the BCT instruction can be used to update a counter in a different accumulator.

28.2 Program Termination and Display Messages

There is a group of six instructions which can be used to send messages into the monitoring file of the job in which the program is being run, and optionally to halt or delete the binary.

The instructions are:

!DISP(*chars*);

Send the message DISPLAY: *chars* to the monitoring file.‡

!DISTY(*index*);

Send the message DISPLAY: *string* to the monitoring file.‡

!SUSWT(*chars*);

Send the message HALTED: *chars* to the monitoring file and suspend the program.

!SUSTY(*index*);

Send the message HALTED: *string* to the monitoring file and suspend the program.

!DEL(*chars*);

Send the message DELETED: *chars* to the monitoring file and delete the program.

!DELTY(*index*);

Send the message DELETED: *string* to the monitoring file and delete the program.

‡*chars* is a two character message. The instructions using this form may be written in various ways, thus if X1 contains the value 2, the following instructions will all have the same effect.

!DISP('A2'); !DISP(4102); !DISP(2114); !DISP((X1+2112));

index is the address of either:

(a) a word whose address field contains the address of *string*, and whose counter field contains the length of *string* in characters, or

(b) a pair of words, the first of which contains the length of *string*, and the second the address. This form is only allowed in 22AM.

string is a sequence of not more than forty characters, beginning in the first character position of a word.

For example, to delete a program with the message:

DELETED : IT WORKED!

the following code can be used:

```
LOWER LOGICAL MESSAGE(4)=
(1OCNT+@MESSAGE+1, "IT WORKED!");
......
LOWEND;
......
!DELTY(MESSAGE);
```

28.3 Number Conversion and Block Movement

A pair of instructions are provided for converting between binary numbers and decimal characters. They operate on a binary value held in a long integer accumulator, and a character representation of the number in store.

```
!CDB(X,address);          for example   !CDB(4,(X2));
```

If the character pointed to by the address in X2 (in the example) is numeric, the long integer value in X45 will be multiplied by ten, and the character will be added to X5. If not, the carry register will be set and no other action taken.

```
!CBD(X,address);          for example   !CBD(4,(X2));
```

For the purposes of this instruction, the value held in X45 (in the example) is considered to be a double length binary fraction. This value is multiplied by ten, and the integral part of the product is stored in the character position indicated by the address in X2. If zero-suppression is set, non-significant zeroes in the resultant number are replaced by zeroes. Zero-suppression may be set by obeying the instruction:

```
!MODE(1);
```

It is unset automatically when a non-zero digit is generated by a CBD order, or by program control with the instruction:

```
!MODE(0);
```

There are two instructions for transferring information from one part of store to another. They operate on a pair of accumulators, which contain the source and destination addresses, and the size of the transfer is indicated by the address field of the instruction.

```
!MOVE(X,N);
```

Move N words from the address specified in X to the address specified in X+1.‡

Example 1: to move ten words from INPUT to OUTPUT.

```
X4:=@INPUT; X5:=@OUTPUT; !MOVE(4,10);
```

Example 2: to set to zero a block of words starting at FIRST, whose length is held in the variable LENGTH.

```
X4:=@FIRST; X5:=@FIRST(1); FIRST:=0; X3:=LENGTH; !MOVE(4,(X3));
```

or alternatively:

```
X4:=@FIRST; X5:=X4+1; FIRST:=0; !MOVE(4,(LENGTH));
```

```
!MVCH(X,N);
```

Move N characters from the character address specified in X to that specified in X+1.‡

Example 3: to move 17 characters from BUFA to BUFB.

```
X4:=@BUFA; X5:=@BUFB; !MVCH(4,17);
```

‡ *The count of words or characters to be transferred must be less than 512. A count of zero will cause 512 words or characters to be transferred. In the case of the MOVE instruction, the addresses in the pair of accumulators used remain unchanged after execution, unless they are in the area to which the transfer was made. In the case of the MVCH instruction, the addresses are updated to point to the character after the last character transferred. (It is therefore possible to concatenate several strings of characters without resetting the destination-address accumulator.)*

28.4 PERI, GIVE and SUSBY

These instructions will not be described in detail here as they are described in the GEORGE manual. The use of PERI and SUSBY instructions may be seen in the sample program in Appendix 7, which reads from a simulated card reader, and writes to a simulated lineprinter. A brief description of the simpler GIVE instructions follows.

```
!GIVE(4,n);
```

A pair of accumulators is specified, in this case X4 and X5. The address field is an integer indicating the variety of the instruction.

n = 1, place the current date in X45 in the form dd/mm/yy, for example 10/09/73.

n = 2, place the current time in X45 in the form hh/mm/ss, for example 17/05/25.

n = 3, place the current size of the program in X4, B0 is set if the program is sparse.

n = 4, the program size is altered to that specified in X4, it will be sparse if B0 is set.

APPENDIX 1

ERRORS AND COMMENTS

A1.1 Errors

Error Number	Meaning	Corrective action taken
0	; or END scanned and STACK broken	STACK is discarded until correct symbol appears.
1	END not matched by BEGIN on STACK	STACK is discarded until BEGIN appears or STACK terminates.
2	BASE statement in LOWER declaration	The statement is ignored.
3	**** read as source	END's forced in to match BEGIN's on stack
4	Type declaration not followed by identifier list	Skip to a recognisable symbol.
5	Illegal symbol in identifier list	Skip to a recognisable symbol.
6	Incorrect array length (not absolute const. or defined)	Array length 1 is assumed.
7	Identifier declared twice	Identifier ignored
8	Incorrect structure of LOCAL (ALL) (BUT)	The statement is ignored.
9	Illegal symbol in label list	The statement is ignored.
10	Unmatched bracketing	Assume)
11	Identifier already declared	Skip to ;
12	Illegal source construction	Skip to ;
13		
14	LOWEND or GLOBEND missing	No action
15	PURE storage not initialised	Sets to zero
16		
17	Too many items in an array	Surplus items ignored

Error Number	Meaning	Corrective action taken
18	$ of a lower item used in a DEFINE statement	Skip to , or ;
19	LOWEND without corresponding LOWER	LOWEND ignored
20	DEFINE not followed by identifier	Skip to , or ;
21	Identifier after DEFINE already declared	Skip to , or ;
22	Identifier not followed by = in DEFINE statement	Skip to , or ;
23	Separator missing in initial value list	, assumed
24	GLOBEND without matching GLOBAL	GLOBEND ignored
25	Impossible error (Stack not free for begin)	Error ignored
26	LOWER not followed by αtype or GLOBAL	Type INTEGER is assumed
27	GLOBAL not followed by block identifier	An area %%%G is created
28	Block identifier not followed by :	A colon is assumed to be present
29	Block identifier followed by illegal declaration	Type INTEGER is assumed
30	LONG not followed by xtype or rtype declaration	Type INTEGER is assumed
31	SYN of item already declared	Skip to , or ;
32	Illegal displacement	Zero substituted
33	Error after ACC or SYN	Skip to , or ;
34	ACC αidentifier not followed by SYN	Skip to , or ;
35	Identifier list not terminated by SYN or ;	; assumed

Error Number	Meaning	Corrective action taken
36	Illegal expression in DEFINE declaration	Value of definee indeterminate
37	Illegal format in DEFINE declaration	Skip to , or ;
38	Illegally formed absolute expression in DEFINE declaration	Skip to , or ;
39	Error in initial value list (incorrect type, undefined ident. etc.)	Zero substituted
40	Illegal symbol within (----) sequence	Assume constant of 0
41	Illegal symbol following a modifier	Assume closing right bracket
42	Illegal symbol following SMO index	Assume closing right bracket
43	Illegal symbol in simple SMO index	The statement is ignored
44	Closing bracket not found	Assume closing right bracket
45	Link accumulator not specified in PROCEDURE	Link accumulator XO is assumed
46	Illegal accumulator given for PROCEDURE	Link accumulator XO is assumed
47	PROCEDURE not followed by identifier	Skip to ;
48	OBEY, not an xcell	The statement is ignored
49	Illegal operand .	Zero substituted
50	Undeclared identifier in L.H.S.	The statement is ignored
51	Identifier undeclared at current block	The statement is ignored
52	Unmodified cell assignment to UPPER	No action
53	Undeclared identifier in statement R.H.S.	A null instruction is generated

Error Number	Meaning	Corrective action taken
54	Identifier undeclared at current block	A null instruction is generated
55	Incompatible types	A null instruction is generated
56	Cell assignment: αcell := αcell	Rest of statement is ignored
57	Accumulator assignment of	The statement is ignored
58	Cell assignment of constant not equal zero	The statement is ignored
59	Illegal operator	A null instruction is generated
60	Identifier too long	The identifier is truncated to 32 characters
61	Real number out of range	Zero substituted
62	Illegal characters following & of real number	Number is ignored
63	NOT encountered on its own	NOT is ignored
64	Too many characters in character sequence	Truncated to four characters
65	Count (CNT) greater than 511	Count field is zeroised
66	Octal longer than 16 digits	Octal is truncated
67	8 or 9 appearing in an octal number	Put in as #10 or #11
68	#, or & standing alone	Character is ignored
69	Subfile name after INCLUDE too long	Statement is ignored
70	Subfile name in INCLUDE not found on MACROFILE	Statement is ignored
71	INCLUDE statement not on a line by itself	Other code on the line is ignored
72	MINUS not followed by a number	MINUS ignored
73	?number not followed by (	(is assumed

Error Number	Meaning	Corrective action taken
74	Conditional compilation switch >10	No conditional action taken
75	? not followed by a number	No conditional action taken
76	String longer than 244 characters	String is truncated
77	Truncation to >23 bits	No truncation done
78	Error in function statement	Statement ignored ; NULL output
79	Accumulator used as label	:=assumed
80	Undeclared identifier in a bracketed sequence	Identifier ignored
81	END found within a CASE statement	CASEND inserted before END
82	Illegal operator	Rest of statement is ignored
83	More than 100 branch aheads at this point	Branch ahead table cleared
84	WHILE clause not terminated by DO	The statement is ignored
85	IF clause not terminated by THEN	The statement is ignored
86	Stack error on fixing-up IF statement	The statement is ignored
87	Label identifier previously declared	Item redeclared as a label
88	Not :=0 in a cell assignment	Null generated
89	Illegal modifier in CASE clause	Modifier X1 is assumed
90	Multiple assignment in a single statement	Rest of statement is ignored
91	Illegal symbol in place of operator	A semi-colon is inserted following previous operand
92	Program or segment starts with illegal symbol	A BEGIN is assumed missing

Error Number	Meaning	Corrective action taken
93	Illegal first character of a statement	The statement is ignored
94	Statement starts with identifier and illegal symbol	A semi-colon is inserted between identifier and symbol
95	Cell designator not followed by ← or :=	The statement is ignored
96	Illegal symbol in place of operand	The statement is ignored
97	END, ELSE or; should have been found	A semi-colon is inserted
98	Wrong type of identifier in PROCEDURE or GOTO statement	The statement is ignored
99	ENTRY not in range 0 - 9	Entry not generated
A0	@ or £ not followed by identifier	The statement is ignored
A1	Illegal operator before address	The statement is ignored
A2	Illegal address assignment	The statement is ignored
A3	CHAR sequence incorrectly terminated	The statement is ignored
A4	Undeclared identifier after $	No action
A5	Undeclared identifier after £	No action
A6	Illegal declaration	Operand set to zero
A7	Illegal identifier after @	Operand set to zero
A8		
A9		
B0	BRINGOVERLAY not followed by identifier	The statement is ignored
B1	CUEOVERLAY not followed by identifier	The statement is ignored
B2	Identifier not correctly declared	The statement is ignored

Error Number	Meaning	Corrective action taken
83		
84		
85		
86		
87		
88		
89		
C0	Illegal condition statement	The condition is ignored skip to next THEN or DO
C1	Illegal symbol following relation	The condition is ignored skip to next THEN or DO
C2	Base symbol not followed by identifier	The condition is ignored skip to next THEN or DO
C3	Displacement symbol not followed by identifier	The condition is ignored skip to next THEN or DO
C4	Character comparison with $	'CH' is ignored
C5	$ identifier (index) outside range 0 - 4095	The condition is ignored skip to next THEN or DO
C6	AND, OR appear in same compound condition	The first to appear is assumed
C7	Undeclared identifier in condition	Zero is substituted
C8	Incompatible types in condition	The condition is ignored, skip to next THEN or DO
C9	Not integer accumulator on LHS of condition	The condition is ignored, skip to next THEN or DO
CA	SMO in a condition when SMOMACRO is set or assumed	The condition is ignored, skip to next THEN or DO
CB		
CC		
CD		
CE		
CF		

Error Number	Meaning	Corrective action taken
D7		
D8		
D9		
E0		
E1	FOR not followed by integer acc. assignment	No action
E2	Illegal increment in FOR clause	The statement is ignored up to DO
E3	UNTIL not found in FOR clause	No action
E4	Illegal limit in FOR clause	The statement is ignored up to DO
E5	DO does not terminate FOR clause	No action
E6	Illegal item following '-' in limit	The statement is ignored up to DO
E7	Stack is broken on fixing-up DO	No action
E8	No STEP between FOR and UNTIL	Step 1 assumed
E9	END not followed by ; END or ELSE	; inserted
F0		
F1	Illegal operand in function statement	The statement is ignored
F2	Illegal constant operand	The statement is ignored
F3	Function statement not terminated by)	No action
F4	Undeclared operand in function statement	The statement is ignored
F5		
F6		
F7		
F8		
F9		
G0	GOTO not followed by identifier	The statement is ignored
G1		
G2		
G3		

Error Number	Meaning	Corrective action taken
G4		
G5		
G6	Illegal character in DATA statement	The symbol is ignored
G7	Block identifier declared differently elsewhere	Skip to ;
G8	Redefinition of GLOBAL area in different mode	The area is forced into LOWER
G9	Too many GLOBAL declarations in segment	Compiler HALTS:-XX at this point
M0	Monadic operator in wrong context	The statement is ignored
M1	Unrecognised monadic operator	Rest of statement is ignored
M2	MEMBER not followed by (	The statement is ignored
M3	Member number >5	The statement is ignored
M4	Member number not followed by,	The statement is ignored
M5	Member priority >99	The statement is ignored
M6	No) at close of member statement	) assumed
M7	) not followed by ; in member statement	; assumed
M8		
M9		
P0	Not integer or definee after , in a procedure statement	0 substituted
P1	Procedures nested to depth >16	SENDTO file (if any) is closed
P2	Number after, in a procedure statement >63	0 substituted
P3	Accumulator not followed by , or) in a procedure statement	) assumed
P4		
P5		
P6		
P7		
P8		
P9		

Error Number	Meaning	Corrective action taken
R0	Identifier not found where expected	Skip to ;
R1	Identifier not followed by (in replacers statement	Skip to ;
R2	(not followed by integer in replacers statement	Skip to ;
R3	Integer >510 in replacers statement	510 substituted
R4	Integer not followed by) in replacers statement	Skip to ;
R5	; should terminate a replacers statement	; assumed

A1.2 Comments

Comment Number	Meaning
1	Section of source not compiled as conditional compilation switch is off
2	A second [has been found before a]
3	A] has been found without a matching [
4	a double-length 'shift-type' instruction has been generated
5	Nested LOWER declarations have been used
6	, not followed by SL in an INCLUDE statement, SL assumed
7	A SMO macro has been generated
8	CARRY or NOT CARRY not the first in a compound condition
9	Trivial condition, eg >=#400CNT
10	No D after a double length integer
11	Base generated, though no BASE statement present
12	Attempt to access item outside normal domain
20	Other than one machine instruction generated by a statement in a case sequence
21	Division will corrupt an accumulator not explicitly mentioned in the statement (as in PLAN)
22	Multiplication will use an accumulator not explicitly mentioned in the statement (as in PLAN)

APPENDIX 2

1900 CHARACTER SET

00	0	10	8	20	SPACE	30	(
01	1	11	9	21	!	31	)
02	2	12	:	22	"	32	*
03	3	13	;	23	#	33	+
04	4	14	<	24	£	34	,
05	5	15	=	25	%	35	-
06	6	16	>	26	&	36	.
07	7	17	?	27	'	37	/

40	@	50	H	60	P	70	X
41	A	51	I	61	Q	71	Y
42	B	52	J	62	R	72	Z
43	C	53	K	63	S	73	[
44	D	54	L	64	T	74	\
45	E	55	M	65	U	75	]
46	F	56	N	66	V	76	↑
47	G	57	O	67	W	77	←

APPENDIX 3

SYNTAX DEFINITIONS IN ALPHABETIC ORDER

SECTION NO			
17.1	αaccsyn	::=	αidentifier SYN αacc {α=x|r|xx|rr}
17.1	αaccsyndec	::=	ACC αaccsynlist; {α=x|r|xx|rr}
17.1	αaccsynlist	::=	αaccsyn |αaccsynlist, αaccsyn {α=x|r|xx|rr}
8.1	αcell	::=	simpleαcell | smoαcell {α=x|r|xx|rr}
16.1	αcelldec	::=	αtype αitemlist ; {α=x|r|xx|rr}
17.1	αcellsyndec	::=	αtype αsynlist ; {α=x|r|xx|rr}
13.8	αfixedcell	::=	αidentifier | αidentifier (integer) | αidentifier (-integer) {α=x|r|xx|rr}
16.1	αidentifier	::=	identifier {α=x| r| xx| rr}
13.8	αinitial	::=	αvalue {α=r|xx|rr}
16.1	αinitiallist	::=	αinitial|αinitial * integer | αinitiallist, αinitial | αinitiallist, αinitial * integer {α=x| r| xx| rr}
16.1	αitem	::=	αinitial|αidentifier = αinitial | αidentifier (integer) | αidentifier (integer) = (αinitiallist) {α=x|r|xx|rr}
16.1	αitemlist	::=	αitem | αitemlist,αitem {α=x|r|xx|rr}
14.1	αrelationex	::=	αacc relation αprimary {α=r| xx| rr}
17.1	αsyn	::=	αidentifier SYN fixedcell |αidentifier SYN (integer)| αidentifier SYN (integer)= αinitial {α=x|r|xx|rr}
17.1	αsynlist	::=	αsyn | αsynlist, αsyn {α=x|r|xx|rr}

```
17.1  accsyndec        ::=  xaccsyndec | raccsyndec | xxaccsyndec |
                            rraccsyndec

11.1  addop            ::=  +|-|++|--

 8.1  address          ::=  wordaddress | CHAR integer | CHAR integer
                            OF wordaddress |@ identifier

14.1  andexpression    ::=  condition AND condition |
                            andexpression AND condition

20.1  aop              ::=  +|-|*|/

14.1  assignment       ::=  xassignment | rassignment | xxassignment |
                            rrassignment

18.1  basedec          ::=  BASE;

13.8  basicfixed       ::=  @ fixedcell |$ fixedcell | £ identifier

 2.1  basicsymbol      ::=  char | reservedword |'|"

12.1  block            ::=  BEGIN blockbody label? END |
                            BEGIN declist blockbody label? END

12.1  blockbody        ::=  statement; | blockbody statement;

19.1  blockidentifier::=    identifier

14.1  casestatement    ::=  CASE modifier OF statementlist CASEND

 8.1  cell         :: ::=  xcell | rcell | *xcell | rrcell

14.1  cellassignment   ::=  xcellassignment | rcellassignment |
                            xxcellassignment | rrcellassignment

16.1  celldec          ::=  xcelldec | rcelldec | xxcelldec |
                            rrcelldec

18.1  celldeclist      ::=  celldec | celldeclist celldec

14.1  cellidentifier   ::=  upperidentififer | loweridentifier

17.1  cellsyndec       ::=  xcellsyndec | rcellsyndec | xxcellsyndec |
                            rrcellsyndec
```

4.1 ch ::= char |'' |"

2.1 char ::= letter | digit | + | - | * | / | < | = | > | # | , | .
| ; | : | @ | & | % | (|) | [|] | £ | $ | ? | !

14.1 charel ::= <CH | >CH | <=CH | >=CH

4.1 charsequence ::= 'ch' | 'ch ch' | 'ch ch ch' | 'ch ch ch ch'

14.1 complex statement ::= ifstatement | forstatement |
whilestatement | includestatement |
label complexstatement

20.1 conditcomp ::= ?integer (section of program) ?integer

14.1 condition ::= relationex | OVERFLOW | FOVERFLOW |
CARRY | NOT OVERFLOW |
NOT FOVERFLOW | NOT CARRY

8.1 coperand ::= integer | octalinteger | modifier |
simplexcell

4.1 couht ::= integer CNT

13.1 datastatement ::= DATA initial | DATA (initiallist)

12.1 dec ::= accsyndec | cellsyndec | definedec |
externaldec | nonplasyddec |
proceduredec | puredec | storagedec

12.1 declist ::= dec | declist dec

20.1 defbasic ::= xvalue | identifier

20.1 defexpr ::= defprim | defprim aop defbasic

20.1 definedec ::= DEFINE deflist ;

20.1 deflist ::= identifier = defexpr |
deflist, identifier = defexpr

20.1 defprim ::= defbasic | $ fixedcell |
@ fixedcell - @ fixedcell |
£ upperidentifier - £ upperidentifier |
£ loweridentifier - £ loweridentifier |
@ instridentifier - @ instridentifier

2.1 digit ::= 0|1|2|3|4|5|6|7|8|9

4.1 exponent ::= integer | MINUS integer

```
 8.1   extendedcell   ::=  cell | upperidentifier (integer) |
                           upperidentifier ( -integer)

19.1   externaldec    ::=  EXTERNAL  externallist ;

19.1   externallist   ::=  spec | externallist, spec

15.1   fidone         ::=  BRN| BVS| BVSR| BVC| BVCR| BCS| BCC| BVCI|
                           SUSTY| DISTY| DELTY| SUSWT| DISP| DEL| OBEY|
                           MODE| FIX| SFPZ| SMO| STOZ| ACT| RMS

15.1   fidsimple      ::=  NULL| SUSAR| LFPZ

15.2   fidtwo         ::=  SFP| SUSBY| REL| DIS| CONT| SUSDP| ALLOT|
                           PERI| SUSMA| AUTO| SUSIN| GIVE| RRQ| LDX|
                           ADX| NGX| SBX| LDXC| ADXC| NGXC| SBXC| STO|
                           ADS| NGS| SBS| STOC| ADSC| NGSC| SBSC| ANDX|
                           ORX| ERX| LDCH| LDEX| TXU| TXL| ANDS| ORS|
                           ERS| DCH| DEX| DSA| DLA| MPY| MPR| MPA| CDB|
                           DVD| DVR| DVS| CBD| BCE| BNZ| BPZ| BNG| BUX|
                           BDX| BCHX| BCT| CALL| EXIT| BFP| LDN| ADN|
                           NGN| SBN| LDNC| ADNC| NGNC| SBNC| SLC| SLL|
                           SLA| SRC| SRL| SRA| SRAV| NORM| MVCH| SLC2
                           SLL2| SLA2| SRC2| SRL2| SRA2| SRAV2| ANDN|
                           ORN| ERN| LDCT| MOVE| SUM| FLOAT| FAD| FSB|
                           FMPY| FDVD| LFP

13.8   fixedaddress   ::=  basicfixed | CHAR integer |
                           CHAR integer OF basicfixed |
                           @ identifier | @ lidentifier

13.8   fixedcell      ::=  xfixedcell | rfixedcell | xxfixedcell |
                           rrfixedcell

14.1   forstatement   ::=  FOR xassignment STEP increment UNTIL
                           xprimary DO statement

 4.1   fraction        ::=  integer . digit |  fraction digit

15.1   function       ::=  simplef | oneargf | twoargf
       statement

19.1   gl             ::=  GLABEL

19.1   gldeclist      ::=  blockidentifier : simpledeclist |
                           gldeclist blockidentifier : simpledeclist

19.1   globaldec      ::=  GLOBAL gldeclist GLOBEND ;
```

13.1	gotostatement	::=	GOTO lidentifier | GOTO pidentifier | GO TO lidentifier | GO TO pidentifier
3.1	identifier	::=	letter |%| identifier letter | identifier digit | identifier %
14.1	ifstatement	::=	IF logicalexpression THEN simplestatement ELSE statement | IF logicalexpression THEN statement
20.1	include statement	::=	INCLUDE (subfilename) | INCLUDE (subfilename, SL)
14.1	increment	::=	xprimary | -xprimary | CHAR
13.8	initial	::=	xinitial | rinitial | xxinitial | rrinitial
13.8	initiallist	::=	initial | initial * integer | initiallist, initial | initiallist, initial * integer
20.1	instridentifier	::=	lidentifier | pidentifier
4.1	integer	::=	digit | integer digit
13.1	label	::=	gl? lidentifier : | ENTRY digit :
2.1	letter	::=	A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
13.1	lidentifier	::=	identifier
8.1	logical	::=	AND |OR| ER
14.1	logical expression	::=	andexpression | orexpression | condition
5.4	lowerαidentifier	::=	identifier {α=x|r|xx|rr}
5.4	loweridentifier	::=	lowerxidentifier | lowerridentifier | lowerxxidentifier | lowerrridentifier
18.1	lowerdec	::=	LOWER lwdeclist LOWEND;
19.1	lowerglobaldec	::=	GLOBAL lowergldeclist GLOBEND;
19.1	lowgldwclist	::=	blockidentifier : celldeclist | lowgldeclist blockidentifier : celldeclist

18.1 lwdeclist ::= celldec | lower globaldec | lwdeclist
celldec | lwdeclist lowerglobaldec

3.1 modifier ::= X1|X2|X3

8.1 monadic ::= NEG | EX | CH | CARRY | NEG CARRY

19.1 nonplasyddec ::= NONPLASYD externallist ;

19.1 nonplasyd statement ::= pidentifier | lidentifier |
pidentifier (integer) |
lidentifier (integer)

15.1 nparam ::= simplecell | integer | octalinteger |
charsequence | pidentifier | lidentifier

13.1 obeystatement ::= OBEY xcell

4.1 octaldigit ::= 0|1|2|3|4|5|6|7

4.1 octalinteger ::= octaldigit | octalinteger octaldigit

15.1 oneargf ::= ! fidone (nparam)

14.1 orexpression ::= condition OR condition |
orexpression OR condition

14.1 partaddress ::= £ cellidentifier | $ cellidentifier

11.1 partop ::= EX|SA|LA|CH

13.1 pidentifier ::= identifier

13.1 proceduredec ::= gl? PROCEDURE pidentifier
(xacc ret?) ; statement ;

13.1 procedure statement ::= lidentifier | pidentifier

19.1 program ::= statement

19.1 puredec ::= PURE storagedeclist PUREND ;

```
3.1   racc              ::=  A1

9.1   rarithmetic       ::=  +|-|*|/

9.1   rassignment       ::=  A1 := rprimary | A1 := A1 |
                             A1 := NEG rprimary |
                             rassignment FROM rprimary |
                             rassignment UNDER rprimary |
                             rassignment rarithmetic rprimary

11.1  rcellassignment   ::=  rcell := A1 | rcell := 0.0

4.1   real              ::=  fraction | fraction & exponent |
                             integer & exponent

14.1  relation          ::=  <|>|<=|>=|NOT=|=

14.1  relationex        ::=  xrelationex | rrelationex | xxrelationex |
                             rrrelationex

2.1   reservedword      ::=  ACC|AND|A1|A12|
                             BASE|BEGIN|
                             CARRY|CASE|CASEND|CH|CHAR|CNT|
                             DATA|DEFINE|DO|
                             ELSE|END|ENTRY|ER|EX|EXTERNAL|
                             FIX|FLOAT|FOR|FOVERFLOW|FROM|FUNCTION|
                             GLABEL|GLOBAL|GLOBEND|GO|GOTO|
                             IF|INCLUDE|INTEGER|
                             LA|LOGICAL|LONG|LOWEND|LOWER|
                             MINUS|
                             NEG|NONPLASYD|NOT|NOT=|NULL|
                             OBEY|OF|OR|OVERFLOW|
                             PROCEDURE|PURE|PUREND|
                             REAL|RETURN|
                             SA|SLA|SLC|SLL|SRA|SRAV|SRC|SRL|STEP|SYN|
                             THEN|TO|TOPGLOBAL|
                             UNDER|UNTIL|WHILE|
                             X0|X1|X2|X3|X4|X5|X6|X7|
                             X01|X12|X23|X34|X45|X56|X67|X70|
                             :=|++|--|''|""|<CH|>CH|<=CH|>=CH|<=|>=

13.1  ret               ::=  , integer

13.1  returnstatement   ::=  RETURN | RETURN (integer)

9.1   rprimary          ::=  rvalue | rcell

3.1   rracc             ::=  A12

10.1  rrarithmetic      ::=  +|-|*|/

10.1  rrassignment      ::=  A12 := rrprimary |
                             A12 := NEG rrprimary | A12 := A12 |
                             rrassignment rrarithmetic rrprimary
```

```
11.1  rrcellassignment ::=  rrcell  :=  A12

10.1  rrprimary        ::=  rrvalue | rrcell

16.1  rrtype           ::=  LONG REAL

 4.1  rrvalue          ::=  real L | MINUS real L

16.1  rtype            ::=  REAL

 4.1  rvalue           ::=  real | MINUS  real

11.1  samexcellassign  ::=  xcell := xcell

11.1  samexxcellassign ::=  xxcell := xxcell

19.1  segment          ::=  program | proceduredec

 8.1  shift            ::=  SLL |SRL| SLA| SRA| SRAV |SLC |SRC

 6.1  simpleαcell      ::=  lowerαidentifier |
                            lowerαidentifier (integer) |
                            lowerαidentifier (-integer) |
                            (integer) | (modifier) |
                            (modifier + integer) |
                            αidentifier (modifier)
                            αidentifier (modifier + integer) |
                            αidentifier (modifier - integer)
                            {α=x|r|xx|rr}

15.1  simplecell       ::=  xcell | rcell | xxcell | rrcell

18.1  simpledeclist    ::=  celldec | basedec | simpledeclist celldec |
                            simpledeclist basedec

15.1  simplef          ::=  ! fidsimple

26.1  simplesmo        ::=  simplexcell | £ upperidentifier |
                            blockidentifier

14.1  simplestatement  ::=  assignment | cellassignment |
                            functionstatement | nonplasydstatement |
                            procedurestatement | casestatement |
                            block | gotostatement | returnstatement
                            datastatement | obeystatement | NULL |
                            label simplestatement
```

26.1 smoαcell ::= (simplesmo) | (simplesmo + integer) |
(simplesmo + modifier) |
(simplesmo + modifier + integer) |
αidentifier (simplesmo) |
αidentifier (simplesmo + integer) |
αidentifier (simplesmo + modifier) |
αidentifier (simplesmo + modifier + integer)
{α=x|r}

19.1 spec ::= pidentifier | lidentifier |
pidentifier (xacc) |
lidentifier (xacc)

13.8 st ::= char |""|␣|'

14.1 statement ::= simplestatement | complexstatement

14.1 statementlist ::= statement ; | statementlist statement ;

13.8 stlist ::= st | stlist st

19.1 storagedec ::= celldec | basedec | lowerdec | globaldec |
topglobaldec | lowerglobaldec

19.1 storagedeclist ::= storagedec | storagedeclist storagedec

11.1 storeop ::= NEG | CARRY | NEG CARRY

13.8 string ::= "stlist"

20.1 subfilename ::= identifier . identifier

19.1 topglobaldec ::= TOPGLOBAL blockidentifier :
simpledeclist GLOBEND ;

4.1 truncated ::= integer T integer | MINUS integer T
integer

15.1 twoargf ::= ! fidtwo (xParam, nparam) |
! fidtwo (,nparam)

5.4 upperαidentifier ::= identifier {α=x|r|xx|rr}

5.4 upperidentifier ::= upperxidentifier | upperridentifier |
upperxxidentifier | upperrridentifier

14.1 whilestatement ::= WHILE logicalexpression DO statement |
DO statement

```
8.1  wordaddress      ::=  $ extendedcell | @ extendedcell |
                           £ identifier

3.1  xacc             ::=  X0|X1|X2|X3|X4|X5|X6|X7

8.1  xarithmetic      ::=  +|++|-|--|*|/

8.1  xasgn            ::=  X0:=X0|X1:=X1|X2:=X2|X3:=X3|X4:=X4|
                           X5:=X5|X6:=X6|X7:=X7

8.1  xassignment      ::=  xacc := xprimary | xacc := address |
                           xasgn + address | xasgn - wordaddress |
                           xacc := monadic xprimary |
                           xacc := CNT coperand |
                           xassignment xarithmetic xprimary |
                           xassignment logical xprimary |
                           xassignment shift coperand

11.1 xcellassignment  ::=  xcell := xacc | xcell := 0 |
                           xcell := storeop xacc |
                           xcell := partop xacc | samexcellassign |
                           xcellassignment addop xcell |
                           xcellassignment logical xacc

13.8 xinitial         ::=  xvalue | string | fixedaddress |
                           functionstatement |
                           count + fixedaddress |
                           count + integer | count + truncated |
                           count + octalinteger

15.1 xparam           ::=  octaldigit | xacc | xxacc

8.1  xprimary         ::=  xvalue | xacc | xcell

14.1 xrelationex      ::=  xacc relation xprimary |
                           xacc relation partaddress |
                           xacc charel xprimary

16.1 xtype            ::=  INTEGER | LOGICAL

4.1  xvalue           ::=  integer | MINUS integer | octalinteger |
                           truncated | count | charsequence|string

3.1  xxacc            ::=  X01|X12|X23|X34|X45|X56|X67|X70

10.1 xxarithmetic     ::=  +|-|++|--
```

```
10.1  xxassignment      ::=   xxacc := xxprimary |
                              xxacc := NEG xxprimary |
                              xxassignment xxarithmetic xxprimary |
                              xxacc := xprimary |
                              xxassignment shift coperand

11.1  xxcellassignment  ::=   xxcell := xxacc | xxcell :=0 |
                              xxcell := NEG xxacc | samexxcellassign |
                              xxcellassignment addop xxacc

19.1  xxprimary         ::=   xxvalue | xxacc | xxcell

16.1  xxtype            ::=   LONG INTEGER

 4.1  xxvalue           ::=   integer D | MINUS integer D |
                              octalinteger D
```

APPENDIX 4

USE OF PROGRAM XMED

The following job will create a file called MYFILE, and will use PROGRAM XMED to copy three sections of source code into subfiles within MYFILE.

```
JOB MYJOB, :USER,T????
LOAD :SUBLIB.PROGRAM XMED
CREATE !
ASSIGN *LP,!
ONLINE *CR
CREATE MYFILE(*DA,BUCK1,KWOR16)
ASSIGN *DA2,MYFILE(WRITE)
ENTER 1
*OUT (FILE)
*ALGOL SUBFILE1

Text of first subfile

****
*ALGOL SUBFILE2

Text of second subfile

****
*ALGOL SUBFILE3

Text of third subfile

****
////
LISTFILE !,*LP
ERASE !
ENDJOB
????
```

Note that the subfile terminator is **** and the XMED run terminator is ////. The terminator for the JOB deck must therefore be neither of these, e.g. ???? as above. The parameter *ALGOL could be replaced by either *FORTRAN or *EMA.

The first of these subfiles could be included in a PLASYD program by the statement:-

```
INCLUDE (MYFILE.SUBFILE1);
```

or:-

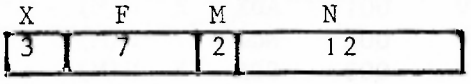
```
READFROM (MYFILE.SUBFILE1)
```

APPENDIX 5

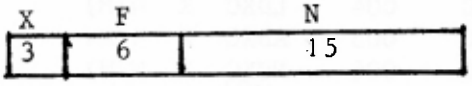
1900 ORDER CODE

The standard instruction formats on the 1900 range are:

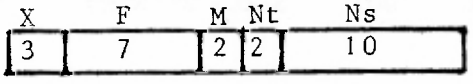
NORMAL ORDERS



BRANCH ORDERS

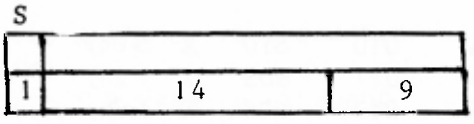


SHIFT ORDERS

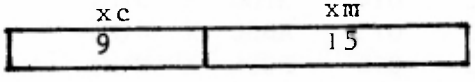


The standard word formats are:

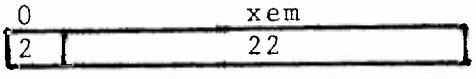
FLOATING POINT NUMBER



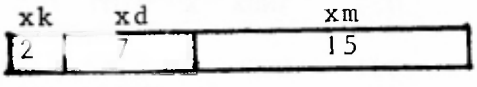
15AM COUNTER-MODIFIER



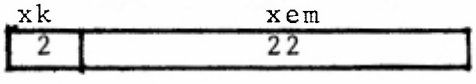
22AM COUNTER-MODIFIER



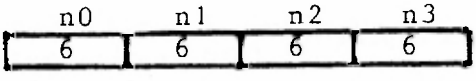
15AM CHARACTER COUNTER-MODIFIER



22AM CHARACTER COUNTER-MODIFIER



CHARACTER POSITIONS



Instructions marked with a * require the PLAN manual for a fuller description.
Instructions marked with a C may set the carry bit but cannot cause overflow.
The 043 order may set V or C. The carry register C is left clear by any order except 023, 117 and 123 unless that order sets C.
Instructions marked with a V may cause overflow.

	Function	Mnemonic			
Arithmetic into Accumulators					
V	000	LDX	X	N(M)	Load contents N(M) into X
V	001	ADX	X	N(M)	Add contents N(M) to X
V	002	NGX	X	N(M)	Load negative of contents N(M) into X
V	003	SBX	X	N(M)	Subtract contents N(M) from X
C	004	LDXC	X	N(M)	Load contents N(M) into X
C	005	ADXC	X	N(M)	Add contents N(M) to X
C	006	NGXC	X	N(M)	Load negative of contents N(M) into X
C	007	SBXC	X	N(M)	Subtract contents N(M) from X
Arithmetic into Store					
V	010	STO	X	N(M)	Store X in address N(M)
V	011	ADS	X	N(M)	Add X into address N(M)'s contents
V	012	NGS	X	N(M)	Store negative X into address N(M)
V	013	SBS	X	N(M)	Subtract X from address N(M)'s contents
C	014	STOC	X	N(M)	Store X in address N(M)
C	015	ADSC	X	N(M)	Add X into address N(M)'s contents
C	016	NGSC	X	N(M)	Store negative X into address N(M)
C	017	SBSC	X	N(M)	Subtract X from address N(M)'s contents
Logical Operations into Accumulator					
	020	ANDX	X	N(M)	X is result of ANDing X and contents N(M)
	021	ORX	X	N(M)	X is result of ORing X and contents N(M)
	022	ERX	X	N(M)	X is result of EXCLUSIVE ORing X and contents N(M)
Interpret					
	023	OBEY		N(M)	Obey instruction in address N(M)
Part Word into Accumulator					
	024	LDCH	X	N(M)	Load character at address N(M) into X
	025	LDEX	X	N(M)	Load exponent (xe) from address N(M) into X
Comparison					
	026	TXU	X	N(M)	Set C if X different from contents N(M)
	027	TXL	X	N(M)	Set C if X less than contents N(M)

Logical Operations into Store

	030	ANDS	X N(M)	Contents N(M) is result of ANDing X and contents N(M)
	031	ORS	X N(M)	Contents N(M) is result of ORing X and contents N(M)
	032	ERS	X N(M)	Contents N(M) is result of EXCLUSIVE ORing X and contents N(M)

Store Zero

	033	STOZ	N(M)	Store zero in address N(M)

Part Word Manipulation into Store

	034	DCH	X N(M)	Lower 6 bits X stored in character address N(M)
	035	DEX	X N(M)	Lower 9 bits X stored in exponent of address N(M)
	036	DSA	X N(M)	Lower 12 bits X stored in address N(M)
	037	DLA	X N(M)	Lower 15 bits X stored in address N(M)

Multiplication

V	040	MPY	X N(M)	Multiply X by contents address N(M)
V	041	MPR	X N(M)	Multiply X by contents address N(M) and round
*	042	MPA	X N(M)	Multiply and accumulate

Character Conversion

V	043	CDB	X N(M)	X is set to 10 times X plus contents of character address N(M)
*	047	CBD	X N(M)	Binary to decimal

Division

*	044	DVD	X N(M)	Divide
*	045	DVR	X N(M)	Divide and round
*	046	DVS	X N(M)	Divide single length

Branch on State of Accumulator

	050	BZE	X N	Branch to address N if X = 0
	052	BNZ	X N	Branch to address N if X ≠ 0
	054	BPZ	X N	Branch to address N if X ≥ 0
	056	BNG	X N	Branch to address N if X < 0

Indexing and Counting

	060	BUX	X	N	15AM:	Add 1 to xm part of X and subtract 1 from xc part. Branch to address N if xc ≠ 0
					22AM:	Add 1 to xem part of X and branch to address N
	062	BDX	X	N	15AM:	Add 2 to xm part of X and subtract 1 from xc part. Branch to address N if xc ≠ 0
					22AM:	Add 2 to xem part of X and branch to address N
	064	BCHX	X	N	15AM:	Increment character address (xk:xm) of X by 1 and subtract 1 from xd. Branch to address N if xd ≠ 0
					22AM:	Increment character address and jump to address N

Subroutine Linkage

	070	CALL	X	N	Call routine at address N leaving link in X
V	072	EXIT	X	N	Return to address N+X

Miscellaneous Branch

	074	BRN		N	X=0, branch to address N
	074	BVS		N	X=1, branch if V set to address N
	074	BVSR		N	X=2, branch if V set and clear V to address N
	074	BVC		N	X=3, branch if V clear to address N
	074	BVCR		N	X=4, branch if V is clear to address N, clear V
	074	BCS		N	X=5, branch if C is set to address N
	074	BCC		N	X=6, branch if C is clear to address N
V	074	BVCI		N	X=7, branch if V is clear to address N, invert V
V	076	BFP		N	X=0, branch to address N if A1=0.0
V					X=1, branch to address N if A1≠0.0
V					X=2, branch to address N if A1≥0.0
V					X=3, branch to address N if A1<0.0
					X=4, branch to address N if floating point overflow clear
					X=5, branch to address N if floating point overflow set

Arithmetic with Small Integers

	100	LDN	X	N(M)	Set X to N(M)
V	101	ADN	X	N(M)	Add N(M) to X
	102	NGN	X	N(M)	Set X to -N(M)
V	103	SBN	X	N(M)	Subtract N(M) from X
	104	LDNC	X	N(M)	Set X to N(M)
C	105	ADNC	X	N(M)	Add N(M) to X
C	106	NGNC	X	N(M)	Set X to -N(M)
C	107	SBNC	X	N(M)	Subtract N(M) from X

Shifting

	110	SLC	X	N(M)	Nt=0, shift X left Ns places circular
		SLL	X	N(M)	Nt=1, shift X left Ns places logical
V		SLA	X	N(M)	Nt=2, shift X left Ns places arithmetic
	111	SLC2	X	N(M)	Nt=0, shift X, X+1 left Ns places circular
		SLL2	X	N(M)	Nt=1, shift X, X+1 left Ns places logical
V		SLA2	X	N(M)	Nt=2, shift X, X+1 left Ns places arithmetic
	112	SRC	X	N(M)	Nt=0, shift X right Ns places circular
		SRL	X	N(M)	Nt=1, shift X right Ns places logical
V		SRA	X	N(M)	Nt=2, shift X right Ns places arithmetic
		SRAV	X	N(M)	Nt=3, shift X right Ns places arithmetic on overflow
	113	SRC2	X	N(M)	Nt=0, shift X, X+1 right Ns places circular
		SRL2	X	N(M)	Nt=1, shift X, X+1 right Ns places logical
V		SRA2	X	N(M)	Nt=2, shift X, X+1 right Ns places arithmetic
		SRAV2	X	N(M)	Nt=3, shift X, X+1 right Ns places arithmetic on overflow

Floating Point Normalise

114	NORM	X	N(M)	Normalise X
115	NORM	X	N(M)	Normalise X, X+1

Logical Operations with Small Integers

120	ANDN	X	N(M)	X is set to X AND N(M)
121	ORN	X	N(M)	X is set to X OR N(M)
122	ERN	X	N(M)	X is set to X EXCLUSIVE OR N(M)

Miscellaneous

	123	NULL			Null operation
	124	LDCT	X	N(M)	Sets xc of X to N(M), xm of X to zero
*	125	MODE		N(M)	Set zero suppression mode
	126	MOVE	X	N(M)	Transfer N(M) words from address X to address X+1
	127	SUM	X	N(M)	X is sum of N(M) words from address X+1

Floating Point

130	FLOAT	X	N(M)	Set A1 to contents N(M) floated
131	FIX	X	N(M)	Fix contents A1 and store in N(M)
132	FAD	X	N(M)	Add contents N(M) to A1 (X=0)
133	FSB	X	N(M)	Subtract contents N(M) from A1 (X=0)
133	FSB	X	N(M)	Contents A1 becomes contents N(M) minus A1 (X=4)
134	FMPY	X	N(M)	Multiply contents N(M) by A1 (X=0)
135	FDVD	X	N(M)	Divide contents A1 by N(M) (X=0)
135	FDVD	X	N(M)	A1 becomes contents N(M) divided by A1 (X=4)
136	LFP		N(M)	A1 set to contents N(M)
136	LFPZ			A1 set to zero, X=1
137	SFP		N(M)	Store A1 in address N(M)
137	SFPZ		N(M)	Store A1 in address N(M) and set A1=0, X=1

Peripheral Control

150	SUSBY	X		Suspend if specified peripheral is active
151	REL	X		Release a specified peripheral
152	DIS	X		Disengage a specified peripheral
154	CONT	X	N(M)	Read more program from a specified peripheral
155	SUSDP	X	N(M)	Suspend and dump program on specified peripheral
156	ALLOT	X		Allocate a specified peripheral
157	PERI	X	N(M)	Peripheral transfer depending on control area

Interrupt and Delete

160	SUSTY	N(M)	X=0, Suspend and type message on console typewriter
160	DISTY	N(M)	X=1, Display message
160	DELTY	N(M)	X=2, Delete program and type message
161	SUSWT		X=0, Suspend and wait
161	DISP		X=1, Display
161	DEL		X=2, Delete

*	162	SUSMA		
	163	AUTO	X N(M)	Activate member X at N(M)
	164	SUSAR		Suspend member awaiting reactivation
	164	SUSIN		Suspend member awaiting flag-setting interrupt
	165	GIVE	X	N(M)=0, Give date in binary in X N(M)=1, Give date in characters in X, X+1 N(M)=2, Give time in characters in X, X+1 N(M)=3, Give current core store location in X N(M)=4, Alter core store allocation to that specified by X N(M)=5, Give details of executive and central processor N(M)=8, Give current address mode and branch mode in X N(M)=9, Alter address mode and branch mode to that given in X N(M)=10, give mill time (in microseconds) used by current core image in X, X+1 N(M)=11, give the time in seconds since midnight as a mid-point number in X, X+1 N(M)=12, alter size of the active part of the core image to the value contained in X: this has the same effect as GIVE with N(M)=3 under GEORGE 4. N(M)=16, report on the status of an area of the current core image, as specified in X, X+1. 'Status' refers to read/write/ execute permission, shareability and an indication of whether the area has been used. This is only applicable to GEORGE 4 environments. N(M)=17 change the permission to that specified in X, X+1. This is only applicable to GEORGE 4 environments. N(M)=18, give the permission specified in X, X+1 (in addition to any current permission). This is only applicable to GEORGE 4 environments. N(M)=19, withhold the permission specified in X, X+1. This is only applicable to GEORGE 4 environments.

166	RRQ	N(M)	X=0, read request block into N(M) X=1, replace request block from address N(M)

Special (available on 1906A but not all 1900's)

066	BCT	N	Subtract 1 from xm part of X and branch N if nonzero
116	MVCH	X N(M)	Transfer N(M) characters
117	SMO	X N(M)	Supplementary modifier to next instruction.

The following mnemonics for handling R-TRUSTED programs exist in PLASYD (but not in PLAN):

160	RMS	X=7, request console message. This is not implemented for programs running under GEORGE 3/4
164	ACT	X=7, N(M)=0, activate the program under control (PUC)
167	SPP	Set up parameters for PUC

For further information on these, see User Notice 4 to the Central Processors Manual (ICL Technical Publication 4095).

APPENDIX 6

CODE GENERATED FOR TYPICAL PLASYD STATEMENTS

The statements used as examples assume the following declarations:

```
LOWER
INTEGER LIX,LIA(10);
REAL LRX,LRA(10);
LONG INTEGER LIIX;
LONG REAL LRRX;
LOWEND;
BASE;
INTEGER UIA(10),UIX;
```

In the examples following, the names of the lower identifiers will be used in the code to refer to the address of that identifier. The base directive will cause a lower cell with address BASEU to be defined and will contain the base address of the upper domain. The notation $UIX will be used to represent the address equal to the displacement of UIX from the base. Constants used in the programs are assumed to be stored at the following addresses:

ADDRESS	CONSTANT
L3	3
LM3	-3
L3P	3.0
L31CNT	#03700000
LABCD	'ABCD'
LABCE	'ABCE'

Branches used in the program are written as L1, L2, L3 etc. A temporary working space is assumed to be allocated at address TEMP.

PLASYD instruction	Code Generated
Integer Accumulator Assignments	
(1) X1 := 3;	LDN 1 3
(2) X1 := X2;	LDX 1 2
(3) X2 := MINUS 3;	LDX 2 LM3
(4) X2 := #12;	LDN 2 10
(5) X2 := 31CNT;	LDX 2 L31CNT
(6) X2 := 'ABCD';	LDX 2 LABCD
(7) X2 := LIX;	LDX 2 LIX
(8) X2 := LIA(-1);	LDX 2 LIA-1
(9) X2 := (30);	LDX 2 30
(10) X2 := (X3+4);	LDX 2 4(X3)
(11) X2 := LIA(X3);	LDX 2 LIA(X3)
(12) X2 := UIX(X3);	LDX 2 $UIX(X3)

```
(13)  X2 := LIA(X3+1);                    LDX 2  LIA+1(X3)

(14)  X2 := UIX(X3+1);                    LDX 2  $UIX+1(X3)

(15)  X2 := (LIX);                        SMO    LIX
                                          LDX 2  0

(16)  X2 := ((30));                       SMO    30
                                          LDX 2  0

(17)  X2 := (LIA(X3+2)+X1);               SMO    LIA+2(X3)
                                          LDX 2  0(X1)

(18)  X4 := LIA(LIX(X3+1)+2);             SMO    LIX+1(X3)
                                          LDX 4  LIA+2

(19)  X4 := NEG 3;                        NGN 4  3

(20)  X4 := EX 3;                         LDEX 4  L3

(21)  X4 := CH 3;                         LDCH 4  L3

(22)  X4 := CARRY 3;                      LDNC 4  3

(23)  X4 := NEG CARRY 3;                  NGNC 4  3

(24)  X4 := NEG X3;                       NGX 4   3

(25)  X4 := EX X3;                        LDEX 4  3

(26)  X4 := CH X3;                        LDCH 4  3

(27)  X4 := CARRY X3;                     LDXC 4  3

(28)  X4 := NEG CARRY X3;                 NGXC 4  3

(29)  X4 := CNT 3;                        LDCT 4  3

(30)  X4 := CNT X2;                       LDCT 4  0(X2)

(31)  X4 := £LIA;                         LDN  4  0

(32)  X4 := £UIA(X3);                     LDX  4  BASEU(X3)

(33)  X4 := @LIA;                         LDN  4  LIA

(34)  X4 := @UIA;                         LDX  4  BASEU
                                          ADN  4  0

(35)  X4 := @UIA(X3);                     LDX  4  BASEU
                                          ADN  4  0(X3)

(36)  X4 := $UIA;                         LDN  4  $UIA

(37)  X4 := $(X1+2);                      LDN  4  2(X1)

(38)  X4 := X3+X2;                        LDX  4  3
                                          ADX  4  2

(39)  X4 := 4++3-2;                       LDN  4  4
                                          ADNC 4  3
                                          SBN  4  2

(40)  X4 := LIX(3)--LIA(2)                LDX  4  LIX+3
                                          SBXC 4  LIA+2

(41)  X4 := LIX*LIA                       LDX  4  LIX
                                          MPY  4  LIA
                                          SLA2 4  23

(42)  X4 := LIX/LIA;                      LDX  4  LIX
                                          DVS  3  LIA
```

```
(43)  X4 := X4 AND 3 OR LIX;          ANDN 4  3
                                      ORX  4  LIX

(44)  X4 := 4 SLL 3 SRL 2;            LDN  4  4
                                      SLL  4  3
                                      SRL  4  2

(45)  X4 := X4 SLL X3 SLL(X1);        SLL  4  0(X3)
                                      SMO     0(X1)
                                      SLL  4  0
```

Real Accumulator Assignments

```
(46)  A1 := LRX;                      LFP   LRX

(47)  A1 := A1+LRX*LRA;               FAD   LRX
                                      FMPY  LRA

(48)  A1 := LRX(X1+3)-URX(X2);        LFP   LRX+3(X1)
                                      FSB   $URX(X2)

(49)  A1 := NEG LRX;                  LFPZ
                                      FSB   LRX

(50)  A1 := A1/LRX FROM LRA;          FDVD  LRX
                                      FSB 4 LRA

(51)  A1 := A1 UNDER LRX;             FDVD 4 LRX
```

Long Integer Accumulator Assignments

```
(52)  X12 := LIIX;                    LDX  2  LIIX+1
                                      LDX  1  LIIX

(53)  X12 := NEG LIIX;                NGXC 2  LIIX+1
                                      NGX  1  LIIX

(54)  X12 := X67+X67-LIIX;            LDX  2  7
                                      LDX  1  6
                                      ADXC 2  7
                                      ADX  1  6
                                      SBXC 2  LIIX+1
                                      SBX  1  LIIX

(55)  X67 := X67 SLA 2 SRL 3;         SLA2 6  2
                                      SRL2 6  3

(56)  X67 := (X1);                    LDX  7  1(X1)
                                      LDX  6  0(X1)

(57)  X67 := X67+(X1);                ADXC 7  1(X1)
                                      ADX  6  0(X1)
```

Long Real Accumulator Assignments

```
(58)  A12 := LRRX;                    LFP  2  LRRX

(59)  A12 := NEG LRRX+LRRX;           LFPZ
                                      FSB  2  LRRX
                                      FAD  2  LRRX

(60)  A12 := A12-LRRX*LRRX/LRRX;      FSB  2  LRRX
                                      FMPY 2  LRRX
                                      FDVD 2  LRRX
```

Integer Cell Assignments

```
(61)  LIX := X1;                          STO   1   LIX

(62)  (3) := X4;                          STO   4   3

(63)  (X1) := X2;                         STO   2   0(X1)

(64)  UIA(X1-5) := X2;                    STO   2   $UIA-5(X1)

(65)  LIX := 0;                           STOZ      LIX

(66)  LIX := NEG X3;                      NGS   3   LIX

(67)  LIX := CARRY X3;                    STOC  3   LIX

(68)  LIX := NEG CARRY X3;                NGSC  3   LIX

(69)  LIX := EX X3;                       DEX   3   LIX

(70)  LIX := SA X3;                       DSA   3   LIX

(71)  LIX := LA X3;                       DLA   3   LIX

(72)  LIX := CH X3;                       DCH   3   LIX

(73)  LIX := X1-X2+X3--X4;                STO   1   LIX
                                          SBS   2   LIX
                                          ADS   3   LIX
                                          SBSC  4   LIX

(74)  LIX := LIX AND X1 OR X2;            ANDS  1   LIX
                                          ORS   2   LIX

(75)  (3) := (3)+X2;                      ADS   2   3
```

Real Cell Assignments

```
(76)  LRX := 0.0;                         STOZ      LRX
                                          STOZ      LRX+1

(77)  LRX := A1;                          SFP       LRX
```

Long Cell Assignments

```
(78)  LIIX := 0;                          STOZ      LIIX
                                          STOZ      LIIX+1

(79)  LIIX := X12;                        STO   2   LIIX+1
                                          STO   1   LIIX

(80)  LIIX := NEG X12;                    NGSC  2   LIIX+1
                                          NGS   1   LIIX

(81)  LIIX := LIIX+X12--X56;              ADSC  2   LIIX+1
                                          ADS   1   LIIX
                                          SBSC  6   LIIX+1
                                          SBSC  5   LIIX

(82)  LRRX := A12;                        SFP   2   LRRX+2
                                          SFP       LRRX

(83)  (X1) := X67;                        STO   7   1(X1)
                                          STO   6   0(X1)
```

Procedure Calling

Assume that link used is X1

```
(84)  FRED;                          CALL  1  FRED
(85)  OBEY(X1+2);                    OBEY     2(X1)
(86)  RETURN;                        EXIT  1  0
(87)  RETURN(5);                     EXIT  1  5
(88)  GOTO F;                        BRN      F
```

Case Statement

```
(89)  CASE X1 OF                          BRN      L3
                                    L1               L2
      A1 := A1+LRX                  L2    FAD      LRX
      A1 := A1-LRX;                       FSB      LRX
      GOTO F;                             BRN      F
      X1 := NEG LIX;                      NGX   1  LIX
      LIX := 0;                           STOZ     LIX
      CASEND;                             BRN      L4
                                    L3    ADX   1  L1
                                          OBEY     0(X1)
                                    L4
```

If Statement

```
(90)  IF X1=0                             BNZ   1  L2
        AND OVERFLOW                      BVCR     L2
        AND A1=3.0                        SFP      TEMP
                                          FSB      L3P
                                          BFP   0  L1
                                          LFP      TEMP
                                          BRN      L2
                                    L1    LFP      TEMP
      THEN X2 :=0;                        LDN   2  0
                                    L2

(91)  IF X1=0                             BZE   1  L2
        OR OVERFLOW                       BVSR     L2
        OR A1=3.0                         SFP      TEMP
                                          FSB      L3P
                                          BFP   0  L1
                                          LFP      TEMP
                                          BRN      L3
                                    L1    LFP      TEMP
      THEN X2 := 0;                 L2    LDN   2  0
                                    L3

(92)  IF FOVERFLOW                        BFP   4  L1
      THEN X2 := 0;                       LDN   2  0
                                    L1
```

```
(93)  IF CARRY
      THEN X2 := 0;

(94)  IF NOT FOVERFLOW
      THEN X2 := 0;

(95)  IF A1>0.0

         AND A1<0.0
         AND A1=0.0
         AND A1<=0.0

         AND A1>=0.0
         AND A1 NOT=0.0
         THEN X1 :=0;

(96)  IF X1 <CH 'ABCD'

      THEN X2 :=0;

(97)  IF X1 >CH 'ABCD'

      THEN X2 :=0;

(98)  IF X1=0
      THEN X2 :=0

      ELSE X3 :=0;

(99)  IF X1 < 3
      THEN X1 := 0;

(100) IF X2 <CH 3
      THEN X1 := 0;

(101) IF X2 > 3
      THEN X1 := 0;

(102) IF X2 >CH 3
      THEN X1 := 0;
```

```
          BCC      L1
          LDN   2  0
      L1

          BFP   5  L1
          LDN   2  0
      L1

          BFP   0  L2
          BFP   3  L2
          BFP   2  L2
          BFP   1  L2
          BFP   0  L1
          BFP   2  L2
      L1  BFP   3  L2
          BFP   0  L2
          LDN   1  0
      L2

          TXL   1  LABCD
          BCC      L1
          LDN   2  0
      L1

          TXL   1  LABCE
          BCS      L1
          LDN   2  0
      L1

          BNZ   1  L1
          LDN   2  0
          BRN      L2
      L1  LDN   3  0
      L2

          BNG   2  L1
          TXL   2  3
          BCC      L2
      L1  LDN   1  0
      L2

          TXL   2  3
          BCC      L1
          LDN   1  0
      L1

          BNG   2  L1
          TXL   2  3
          BCS      L1
          LDN   1  0
      L1

          TXL   2  3
          BCS      L1
          LDN   1  0
      L1
```

```
(103)  IF X2 > X1                          STO  2  TEMP
       THEN X3 := 0;                       SBX  2  1
                                           BZE  2  L1
                                           BPZ  2  L2
                                       L1  LDX  2  TEMP
                                           BRN     L3
                                       L2  LDX  2  TEMP
                                           LDN  3  0
                                       L3

(104)  IF X2 >CH X1                        TXU  2  1
       THEN X3 := 0;                       BCC     L1
                                           TXL  2  1
                                           BCS     L1
                                           LDN  3  0
                                       L1

(105)  IF X2 < X1                          STO  2  TEMP
       THEN X3 := 0;                       SBX  2  1
                                           BNG  2  L1
                                           LDX  2  TEMP
                                           BRN     L2
                                       L1  LDX  2  TEMP
                                           LDN  3  0
                                       L2

(106)  IF X4 <CH X1                        TXL  4  1
          AND X5  <CH X1                   BCC     L1
       THEN X3 := 0;                       TXL  5  1
                                           BCC     L1
                                           LDN  3  0
                                       L1

(107)  IF X4  <CH X1                       TXL  4  1
          AND X5  <CH X1                   BCC     L1
       THEN GOTO FRED;                     TXL  5  1
                                           BCS     FRED
                                       L1

(108)  IF X4  <CH X1                       TXL  4  1
          OR X5  <CH X1                    BCS     L1
       THEN X3 := 0;                       TXL  5  1
                                           BCC     L2
                                       L1  LDN  3  0
                                       L2
```

For Statement

```
(109)  FOR X1 := LIX STEP 5                LDX  1  LIX
       UNTIL UIX(X3) DO                L1  STOZ    LIA(X1)
       LIA(X1) := 0;                       TXU  1  $UIX(X3)
                                           BCC     L2
                                           ADN  1  5
                                           BRN     L1
                                       L2

(110)  FOR X1 := LIX STEP -1               LDX  1  LIX
       UNTIL 3 DO                      L1  STOZ    LIA(X1)
       LIA(X1) := 0;                       SBN  1  3
                                           BZE  1  L2
                                           ADN  1  3
                                           SBN  1  1
                                           BRN     L1
```

```
(111)  FOR X1 := LIX STEP -1
       UNTIL 0 DO
       LIA(X1) := 0;
```

```
       LDX   1  LIX
L1     STOZ    LIA(X1)
       BZE   1  L2
       SBN   1  1
       BRN      L1
L2
```

```
(112)  FOR X1 := LIX STEP 1
       UNTIL LIA DO
       LIA(X1) := 0;
```

```
       LDX   1  LIX
L1     STOZ    LIA(X1)
       SBX   1  LIA
       BZE   1  L2
       ADX   1  LIA
       ADN   1  1
       BRN      L1
L2
```

```
(113)  FOR X2 := CHAR 1 OF £UIA
       STEP CHAR
       UNTIL LIX DO
       X6 := CH UIA(X2);
```

```
       LDCT  2  128
       ADX   2  BASEU
L1     LDCH  6  $UIA(X2)
       TXU   2  LIX
       BCC      L2
       BCHX  2  L1
L2
```

While Statement

```
(114)  WHILE A1<LRX DO
       A1 := A1+LRA;
```

```
L1     SFP      TEMP
       FSB      LRX
       BFP   3  L2
       LFP      TEMP
       BRN      L3
L2     LFP      TEMP
       FAD      LRA
       BRN      L1
L3
```

APPENDIX 7

A SAMPLE PLASYD PROGRAM

The following pages are an example of the output produced by the PLASYD compiler. The program in question is used to edit such listings so that source code and error messages are printed, but the store map (which the compiler gives after every segment) is ignored. The use of card reader and lineprinter PERIs may be seen and also the use of PLAN instructions such as SUSBY, SUSWT and MOVE.

If the source code of this program were in a file called TDYP, it could be compiled, and a binary produced, by issuing the command:

```
TASK PLAS,*CR TDYP,SAVE,NORUN
```

```
          COMPILED BY YAPQ/2  ON  07/09/73    AT  14/42/22

SENDTO(A.SUBROUTINES)
NAME(TDYP)
PROGRAM MODE(15AM,DBM)
LOCAL SEGMENTS
            LOCAL ALL;
            PROCEDURE MAIN(X0);
            BEGIN
              LOWER
                LOGICAL CONCHAR,BUFFER(30),
                CRCA(4) = (3CNT,0,120,@BUFFER),
                LPCA(4) = (2CNT,0,121,CHAR 3 OF @CONCHAR);
              LOWEND;

              SKIP: !PERI(0,CRCA); X4:= CRCA(1);
              IF X4 < 0 THEN !SUSBY(0,3);
              X4:= BUFFER(3); X5:= BUFFER(4);
              IF X4 = 'COMP' AND X5 = 'ILED' THEN GOTO RESTART;
              X4:= BUFFER: X5:= BUFFER(1);
              IF X4 = 'PLAN' AND X5 = '(CR)' THEN GOTO RESTART;
              IF X4 = 'ENDP' AND X5 = 'ROG ' THEN GOTO CLOSE;
              GOTO SKIP;

              RESTART: X4:= #51; CONCHAR:= X4;

              COPY: !PERI(0,LPCA); X4:= LPCA(1);
              IF X4 < 0 THEN !SUSBY(0,2);

              ENTRY 0; X4:= '    '; BUFFER(20):= X4; X4:= @BUFFER(20);
              X5:= X4 + 1; !MOVE(4,9); !PERI(0,CRCA); X4:= CRCA(1);
              IF X4 < 0 THEN !SUSBY(0,3);
              X4:= BUFFER; X5:= BUFFER(1);
              IF X4 = 'SEGM' AND X5 = 'ENT ' THEN GOTO SKIP;
              IF X4 = '*   ' AND X5 = '   0' THEN GOTO SKIP;
              IF X4 = 'MAST' AND X5 = 'ER S' THEN GOTO SKIP;
              IF X4 = 'SUBR' AND X5 = 'ILE ' THEN GOTO CLOSE;
              X4:= #41; CONCHAR:= X4; GOTO COPY;

              CLOSE: !PERI(0,LPCA); X4:= LPCA(1);
              IF X4 < 0 THEN !SUSBY(0,2);
              !SUSWT('OK');
            END;
```

```
SEGMENT     MAIN          LINK XO              64 WORDS

GLABEL PROCEDURES       MAIN          LINK XO        ENTRY          0

INTERNAL PROCEDURES     'NONE'

EXTERNAL    'NONE'

NONPLASYD   'NONE'

DEFINEES    'NONE'

LITERALS          15 WORDS

       0    #43575560          COMP
       1    #51544544          ILED
       2    #60544156          PLAN
       3    #30436231          (CR)
       4    #45564460          ENDP
       5    #62574720          ROG
       6    #20202020
       7    #63454755          SEGM
       8    #45566420          ENT
       9    #32202020          *
      10    #20202000             0
      11    #55416364          MAST
      12    #45622063          ER S
      13    #63654246          SUBF
      14    #51544520          ILE

GLABELS     'NONE'

LABELS

       0 SKIP
      21 RESTART
      23 COPY
      58 CLOSE

LOWER PRESET              39 WORDS

       0 (I)      CONCHAR
       1 (I)      BUFFER
      31 (I)      CRCA
      35 (I)      LPCA

PURE LOWER PRESET       'NONE'

UPPER PRESET    'NONE'

PURE UPPER PRESET       'NONE'

LOWER GLOBAL            'NONE'

PURE LOWER GLOBAL       'NONE'

UPPER GLOBAL            'NONE'

PURE UPPER GLOBAL       'NONE'

SYNONYMS OF ABSOLUTE ADDRESSES      'NONE'

ACCUMULATOR SYNONYMS    'NONE'
```

```
                COMPILED BY YAPQ/2  ON  07/09/73     AT  14/42/24

****
SUBFILE SUBROUTINES  OCCUPIES       5 BUCKETS IN AAAB00015501

****************************************

###### COMPILATION OK

****************************************




CONSOLIDATED BY XPCH  1B     DATE     07/09/73    TIME 14/42/57

 DENSE PROGRAM #TDYP (15AM/DBM/NON-SHARABLE)
==========*=======*==========*=========*=========*==========

LOWER DATA
LENGTH          99 WORDS

      LW              *55         45
      LV              *55         45
      LP              *55         45
      LR             *124         84
      LT             *124         84

PURE PART
LENGTH          64 WORDS

      RR              *143        99
      SEG             *143        99   %%%P (     0 WORDS)
      SEG             *143        99   MAIN (    64 WORDS)

IMPURE PART

NOT USED

SIZE          192     WORDS =     1    PAGES

****************************************

###### CONSOLIDATION OK

****************************************


****************************************

###### ENDTASK ON NORUN

****************************************
```

APPENDIX 8

LESS COMMONLY USED COMPILER DIRECTIVES

A8.1 ADDTO

This directive has a similar effect to SENDTO, the difference being that if the file used already contains subfiles, these will be erased by SENDTO, but left unaltered by ADDTO.

A8.2 DUMPON

This directive specifies the file to which the consolidator is to send the binary version of the program. It is necessary to have a DUMPON directive if an overlay program is being compiled, but otherwise the binary may be left in core to be run or SAVE'd.

A8.3 TRUSTED and PRIORITY

The trusted status of the object program may be set by means of the TRUSTED directive. Statuses Q,R,S,T are represented by the values 8,4,2,1 respectively, and the parameter of the directive is formed by adding together the required values. The priority of the program may be specified as a two digit decimal number in the PRIORITY directive, but is meaningless under GEORGE.

A8.4 PROGEVEN and IGNORE

These directives, with parameters LOWER and/or UPPER, affect the placement of relativiser areas within the compiled program. The PROGEVEN directive causes the appropriate areas to begin on even number words of store. This can be used to increase the speed of running of heavily CPU-bound programs. The IGNORE directive prevents any SEGEVEN directives from taking effect.

A8.5 SEGMENTS

This directive causes the compiler to halt EC at the end of successful compilation. The default action is to delete FI #XPCK.

A8.6 OVERLAY and OVERCOMMON

The format of an OVERLAY directive is:

```
OVERLAY (area,unit)name 1,name 2,name 3...
```

The area number must be less than 255, and the unit number less than 1023. The names are the names of segments in the overlay. The OVERCOMMON directive is similar, but the names in this case are of global areas. There may be any number of OVERLAY and OVERCOMMON directives in a program description, but each may be at most one line in length.

A8.7 SECMENT MODE

This directive is similar to PROGRAM MODE except that the address checking mode parameter is omitted, and the options MIXAM for mixed address mode and MIXBM for mixed branch mode are added. The default options are MIXAM and MIXBM.

A8.8 SEGEVEN

This directive is similar to PROGEVEN. If it is desired to remove the restriction that relativiser areas should begin on even numbered words of store, the directive SEGEVEN, without any bracketed qualifying sequence, may be used. If neither PROGEVEN nor SEGEVEN directives are used, relativiser areas are not restricted.

A8.9 SMOMACRO and COMPILESMO

If the SMOMACRO directive is used, all SMO instructions generated by the compiler will be converted into a sequence of instructions which will simulate the effect of the SMO. If the COMPILESMO directive is used, SMO instructions will be included in the object code where required. If neither directive is used, the compiler will check whether or not SMO instructions are available on the processor on which it is running, and act accordingly.

A8.10 READFROM

Use of this directive allows program source to be read from a direct

access file in subfile format, where it has been placed by the utility program XMED, as described in ppendix 4. If a filename and subfilename are given, the compiler will attempt to open a direct access file of that name, and include the contents of the specified subfile as if they were part of the source code. If a filename only is given, all subfiles in the specified file will be included. If a subfilename only is given, it will be assumed to be in the file which is currently in use. The compiler uses DA14 for reading files specified by this directive.

REFERENCES

(1) The ICL 1900 PLAN Reference Manual, (ICL Technical Publications 4004 and 4322) and 1900 PLAN Supplement, by D C Toll (Atlas Computer Laboratory, July 1973).

(2) PL/360, A Programming Language for the 360 Computers, by N Wirth, (JACM Vol 15 No 1, Jan 1968).

(3) The 1906A TASK System, by G W Robinson (Atlas Computer Laboratory, October 1973).


ATLAS