P-6800 PASCAL SYSTEM INSTALLATION PROCEDURE

(1) Use your Flex system to make a working master copy of the LUCIDATA supplied disk which you should then put away safely with all your other archived originals.

(2) Do a CAT of the master disk and check against Fig. 1.

(3) Copy the files RUN.CMD, PASCAL.BIN, P6800ERR.SYS and VALIDATE.TXT from your master disk to your current system disk (assumed 0). You will need about 300 spare sectors under mini Flex or 150 under Flex 2 or 9.

(4) Type RUN, PASCAL, 0.VALIDATE<CR> you will see the first four lines of Fig. 2 come up on your console. Type<CR> in response to the question ENTER NEW VALUE OR RETURN$

(5) The compiled listing of program VALIDATE will be printed on your console.

(6) When the +++ of Flex reappear type RUN,0.VALIDATE<CR>.

(7) Type <CR> in response to the question ENTER NEW VALUE OR RETURN$ and the message SYSTEM SEEMS INTACT should be printed three times on your console.

(8) If you are unable to achieve (7) after a careful check out of your system, testing your disk etc. and repeating the procedure, return the original disk to LUCIDATA and we will replace it. (Include your telephone number).

(9) Now read the rest of the manual carefully if you have not already done so.

```
DIRECTORY OF DRIVE NUMBER 1
DISK: P6800V2  #660     CREATED: 16-FEB-80

FILE#   NAME     TYPE   BEGIN    END    SIZE     DATE      PRT

    1   COMMON   .OVL   01-01   01-07      7   16-FEB-80
    2   RUN      .CMD   01-08   04-09     32   16-FEB-80
    3   RESTART  .CMD   04-0A   04-0A      1   16-FEB-80
    4   PR       .TXT   05-01   05-02      2   16-FEB-80
    5   RUNEQU   .TXT   05-03   05-0A      8   16-FEB-80
    6   OWNCODE  .TXT   06-01   06-05      5   16-FEB-80
    7   X        .OVL   06-06   06-06      1   16-FEB-80
    8   PR       .OVL   06-07   06-07      1   16-FEB-80
    9   C        .OVL   06-08   06-08      1   16-FEB-80
   10   RUNLGO   .CMD   06-09   0A-01     33   16-FEB-80
   11   PASCAL   .BIN   0A-02   13-08     97   16-FEB-80
   12   P6800ERR.SYS    13-09   14-06      8   16-FEB-80
   13   VALIDATE.TXT    14-07   14-07      1   16-FEB-80
   14   REALS    .DEM   14-08   15-03      6   16-FEB-80
   15   CHARINDX.DEM    15-04   15-06      3   16-FEB-80
   16   SCALINDX.DEM    15-07   15-08      2   16-FEB-80
   17   TYPES    .DEM   15-09   16-08     10   16-FEB-80
   18   CHARS    .DEM   16-09   17-03      5   16-FEB-80
   19   FILES    .DEM   17-04   17-0A      7   16-FEB-80
   20   SETS     .DEM   18-01   18-06      6   16-FEB-80
   21   QUIKSORT.DEM    18-07   19-07     11   16-FEB-80
   22   ALFAS    .DEM   19-08   1A-04      7   16-FEB-80
   23   WAITSECS.DEM    1A-05   1A-06      2   16-FEB-80
   24   EASTER   .DEM   1A-07   1B-01      5   16-FEB-80
   25   TRAVERSE.DEM    1B-02   1B-07      6   16-FEB-80
   26   PRIMES   .DEM   1B-08   1C-01      4   16-FEB-80
   27   EIGHTQ   .DEM   1C-02   1C-07      6   16-FEB-80
   28   USERCODE.DEM    1C-08   1D-05      8   16-FEB-80
   29   RANDOM   .DEM   1D-06   1E-0A     15   16-FEB-80
   30   SIN2COS2.DEM    1F-01   1F-05      5   16-FEB-80
   31   VALIDATE.BIN    1F-08   1F-08      1   16-FEB-80

    FILES=31, SECTORS=306, LARGEST=97, FREE=34
```

Fig 1 List of files.

```
P-6800  RUN-TIME  SYSTEM  V 2.2    : COPYRIGHT C 1980 LUCIDATA
USABLE CONTIGUOUS MEMORY  $8000
DEFAULT STACK RESERVATION $1000
ENTER NEW VALUE OR RETURN $

          SYSTEM SEEMS INTACT
          SYSTEM SEEMS INTACT
          SYSTEM SEEMS INTACT

END OF PROGRAM EXECUTION.
```

Fig 2 Validation run.

P-6800
PASCAL

User Guide and Reference Manual
for
The P-6800 PASCAL Language and Compiler
by

David R. Gibby

and

The P-6800 Run-Time System
by

Nigel W. Bennée

Published by: *LUCIDATA,*
OOSTEINDE 223,
VOORBURG 2271 EG,
NETHERLANDS.

NOTICE

Throughout this document, the companies *South West Technical Products* and *Technical Systems Consultants* are referred to as SWTP and TSC respectively.

INDEX

1. FOREWORD

PASCAL is one of the leading programming languages in computer science today. It is a small, practical and general purpose programming language, which is easy to learn and read.

It was developed in the late 1960s by Niklaus Wirth of the Zurich Technical University, with the aim of using it to teach systematic programming. In the last few years it has become increasingly popular in the computing industry as well as in educational institutions. There has been considerable interest among users of "personal" computer systems, but implementations of the language have tended to require hardware configurations which cost much more than most hobbyists can afford.

The P-6800 compiler and run-time system were designed specifically to allow compilation and execution of PASCAL programs on a SWTP 6800 system, using no more hardware than is needed to use the TSC FLEX 1.0 mini floppy disc operating system, even when a program is too large to fit into the available memory. This is made possible by a paging facility which is invoked automatically when the run-time system detects this condition. Programs will execute faster if there is sufficient memory to hold the run-time system, the program P-code, and the stack containing the data areas, but it is possible to compile and execute a program in as little as 12K bytes (+4K).

The P-6800 PASCAL compiler is itself written in the subset of the language which it supports. The compiler generates a file of pseudo (or P) codes (for a hypothetical stack machine) which are interpreted by the run-time system. This decodes them and executes the equivalent 6800 assembler instructions.

Since these P-codes are simple to decode, execution is tens of times faster than is possible with conventional interpreters (such as are used for BASIC programs). On the other hand, since one P-code instruction is equivalent to several M 6800 language instructions, a PASCAL program compiled into P-code is more compact than the object code produced by a conventional compiler would be.

The compiler supports a significant subset of full PASCAL - Files, Procedures, Functions, Recursion, multi-dimensional Arrays, the branching constructs IF..THEN..ELSE and CASE..OF, as well as the looping constructs REPEAT..UNTIL and WHILE..DO. Among the data types supported are BOOLEAN, CHAR, ALFA, INTEGER and BYTE (0..255) as well as the scalars which can be made members of SETs. Among the standard procedures and functions are PEEK, POKE and a USER function. These are provided specifically for the microcomputer hobbyist and the scientist or engineer wishing to use the system for specialist applications.

This guide is written for a wide range of PASCAL users. Those who are not familiar with the language should read Chapters 2 and 3 and the program examples provided as Appendix D, before starting Chapter 4 to learn how to use a standard system.

Experienced PASCALers may skip Chapter 2, but are recommended to read Chapter 3 briefly, to acquaint themselves with this implementation's characteristics.

Users wishing to gain a fuller understanding of how the run-time system operates and how resources are allocated and utilised should study Chapters 4 and 5.

Chapter 6 offers advice on "fine tuning" a P-6800 PASCAL program, and Chapter 7 provides information which will be required by users who wish to build systems for specialist applications.

2. INTRODUCTION TO THE PASCAL LANGUAGE

A PASCAL program comprises two parts, DATA described by declarations and definitions, and STATEMENTS describing actions which are to be performed on the data.

Data are represented within a program by values of VARIABLES. Each variable must be introduced by a variable DECLARATION which associates an identifier with a data TYPE, thereby defining the range of values which may be assumed by that variable. A data type may be one of the standard simple types (BOOLEAN, CHAR or INTEGER) or it may be a SCALAR type defined by enumeration within the program. P-6800 PASCAL also provides the predefined types BYTE (subrange of integer, 0 to 255) to permit the user to economise on storage of data when arrays of small non-negative integers are used, and ALFA (an array holding 6 characters) to permit easy referencing of names and similar strings.

Data may be structured by grouping together components of the same type into ARRAYS, SETS or FILES. In an ARRAY structure a component is selected by an array selector, or computable index, which in this implementation must be of type integer. In a SET structure, the components grouped together must be of the programmer-defined scalar type. A FILE structure is generated by sequentially appending components at its end; a file type definition does not determine the number of components the structure may contain, and the effective limit is determined by the capacity of the device on which the file is being written.

The fundamental STATEMENT is the ASSIGNMENT, in which an EXPRESSION is evaluated to calculate a new value to be assigned to a variable. An expression consists of variables, constants, sets, operators, and functions operating on current data values to produce new values.

Statements may be grouped together to form compound statements. A COMPOUND statement may be a linear sequence of assignments grouped together by the delimiters BEGIN and END, and/or it may specify conditional execution (by means of IF..THEN..ELSE or CASE..OF constructs) or repetition (by means of WHILE..DO, REPEAT..UNTIL or FOR..TO/DOWNTO..DO constructs).

A statement (or compound statement) may be given a name by means of which it can be referenced. It is then known as a PROCEDURE, unless its purpose is to compute a single value to be assigned to that name (identifier), in which case it is known as a FUNCTION. A procedure or function may include constants, variables, procedures and functions declared locally (only existing within the body of the procedure or function containing them) and they may have a fixed number of PARAMETERS. The number and type of these parameters is defined by the process of procedure or function declaration, where they are known as FORMAL parameters. At execution time ACTUAL parameters are used, consisting of an expression in place of each formal parameter. The expressions are evaluated and the resulting values are used as the initial values of the parameter identifiers.

This brief summary can only serve as an introduction to the PASCAL language. It does not mention those PASCAL constructs and facilities which have not been included in this P-6800 implementation e.g. records, pointers. Moreover, it is not possible to describe even a relatively simple language like PASCAL in a few paragraphs. However, after studying the next chapter (describing the language elements), Appendix A (the syntax diagrams which show, in pictorial form, the rules governing the use of the language elements), and the sample programs provided with the compiler, the newcomer to the PASCAL fraternity should very soon be able to start developing his own programs.

A good way to learn how to use PASCAL (or any other language) well, is to read articles and books on the language and how others use it. To assist you in this respect, Appendix E provides a list of texts you may find helpful. If you wish to keep abreast of the latest developments in the PASCAL field, you should find the PASCAL Users Group (PUG) newsletter of interest. Further information about PUG may be obtained from:

either

PASCAL Users Group,
University Computer
Center 227 EX,
208 SE Union Street,
University of Minnesota,
Minneapolis,
MN 55455, USA.

or

PASCAL Users Group,
c/o Computer Studies Group,
Mathematical Department,
The University,
Southampton, SO9 5NH,
Hampshire, England.

3. THE LANGUAGE ELEMENTS

3.1 Lexical tokens

Although the compiler can handle any character in the ASCII character set, lexical tokens are formed from the upper case letters, the digits and selected special characters. They comprise special symbols, reserved word symbols, identifiers, numbers, character strings and comments.

3.1.1 Special symbols

These comprise the following:

+	plus	-	minus
*	times or asterisk	/	slash
=	equals	:=	becomes
,	comma	;	semicolon
:	colon	..	up to
<	less than	>	greater than
<=	less than or equal to	>=	greater than or equal to
<>	not equal to	.	period or decimal point
(	left parenthesis	)	right parenthesis
[	left bracket	]	right bracket
(*	open comment	*)	close comment
"	quote		

3.1.2 Reserved word symbols

AND	ARRAY	BEGIN	CASE	CONST	DIV
DOWNTO	DO	ELSE	END	FILE	FOR
FUNCTION	IF	IN	NOT	OF	OR
PROCEDURE	PROGRAM	REPEAT	SET	THEN	TO
TYPE	UNTIL	VAR	WHILE		

3.1.3 Identifiers

Identifiers are used to denote constants, types, variables, procedures, functions and programs. They must

begin with an upper case letter, and may be followed by any sequence of digits or upper case letters. Identifiers may be of any length, but the P-6800 compiler distinguishes between identifiers by examining only the first 6 characters.

Examples: R J THIS DAY2 JOB7 SOFTWARE

3.1.4 Numbers

Numbers are used to denote constant integer values in decimal notation. Their permitted range is -32767 to +32767.

Examples: 12345 0 -6789

3.1.5 Character strings

Sequences of characters enclosed by quote marks (ASCII character decimal 34) are called character strings. A single character enclosed by quotes may be assigned to or compared with a variable of type CHAR. A sequence of 6 characters enclosed by quotes may be assigned to or compared with a variable of type ALFA. A sequence of any length (subject to line length limitations) may be used in a write statement to copy text on to the output device or a disc file.

```
Examples:  IF ANS = "N" THEN ANSWER := "N";
           IF NAME = "NEWMAN" THEN WRITE ("HOW NEW?");
```

3.1.6 Comments

A comment is any sequence of characters between the comment delimiters, which are (* and *). The presence of a comment in a program does not affect the semantics of that program. However, note that comments

may be nested in P-6800 PASCAL, thereby permitting the effective removal of a section of a program by inserting comment delimiters in the appropriate places in the program source text.

```
Examples: (*THIS IS A COMMENT*)
          PROCONE; (*PROCEDURE ONE*)
          (*THE FOLLOWING LINE IS DE-ACTIVATED BY
            THIS COMMENT
          PROCTWO; (*PROCEDURE TWO*)       *)
```

3.2 Blocks, Locality and Scope

A program consists of a heading followed by a block. A block may contain constant definitions, type definitions, variable definitions and procedure or function definitions. If these definitions are included in a block, they must occur in that order, and must precede the mandatory statement part. Definitions (declarations) are introduced by the reserved word symbols CONST, TYPE, VAR, PROCEDURE, FUNCTION. The statement part of a block has the same form as a compound statement - namely one or more statements delimited by the reserved word symbols BEGIN and END.

Since procedures and functions have the same form as a program - namely a heading followed by a block, they may be declared (nested) within other procedures or functions. Variables declared within a block are called "local" to that block, and exist only within that block. Variables declared in the outermost block (the main program) are said to be "global" since they exist during the activation of the entire program.

The scope of a variable (the range over which it may be accessed) is primarily the block within which it is declared. However, this scope may be restricted if a nested procedure or function includes the declaration of variables, among which is an identifier which is the same as one declared in the outer block. Within the inner block, all

occurrences of that variable identifier are taken to mean the one which was declared the most recently (at the innermost level). This effectively excludes the block containing the inner declaration of the variable from the scope of the variable declared in the outer block.

The activation of a block causes space to be reserved for variables which are local to that block, but the values of those variables are undefined until they receive values by the execution of the appropriate assignment statements.

3.3 Constant Definitions

A constant definition introduces an identifier to denote a constant. (No statement may therefore assign a new value to such an identifier during program execution). In this implementation the only constants permitted are integers.

```
e.g.  CONST  INCREMENT = 1;  DECREMENT = -1;
```

3.4 Type Definitions

A type determines the set of values which variables of that type may assume. In addition to the standard simple types INTEGER, CHAR and BOOLEAN (see below), PASCAL allows the user to define his own types. In this implementation the types BYTE and ALFA (see below) are pre-defined within the compiler, and the ability to define one's own types is limited to enumerated types and the structured type SET.

3.4.1. Simple types

INTEGER	the subset of whole numbers from -32767 to +32767
CHAR	the set of characters in the ASCII standard, ordered according to their 7 bit ASCII codes

BOOLEAN — the set of two values denoted by the identifiers TRUE and FALSE

ALFA — equivalent to the following type definition which would be permitted in full PASCAL:

```
ALFA = PACKED ARRAY[1..6]of CHAR
```

BYTE — equivalent to the following sub-range of integer type definition which would be permitted in full PASCAL:

```
BYTE = 0..255;
```

3.4.2 Enumerated types (scalars)

An enumerated type defines an ordered set of values by enumeration of the identifiers which denote these values. The ordering of these values is determined by the sequence in which the (constant) identifiers are listed.
(See Section 3.6.4.4 on Standard Ordinal Functions).

e.g.
```
TYPE
    PERSON = (ANN, BOB, FRED);
    COLOUR = (RED, YELLOW, GREEN);
```

3.4.3 Structured types

3.4.3.1 Set types

A set type defines the range of values which is the power set of its base type, which in this implementation must be an enumerated type.

e.g.
```
GROUP = SET OF PERSON;
TRAFFICLIGHT = SET OF COLOUR;
```

3.4.3.2 Array types

In this implementation, an array type may not be specifically defined, although variables

may be defined to be ARRAYs of simple types or enumerated types.
(See Section 3.5 and the program examples in Appendix D).

3.4.3.3 File types

Although a file type may not be specifically defined in this implementation, files of characters, bytes or integers are permitted.
(See Section 3.5 and the program examples in Appendix D).

3.5 Declarations of Variables

A variable declaration consists of a list of identifiers denoting the new variables, followed by their type. A variable exists during the existence (activation) of the block in which it is declared (see Section 3.2).

```
Examples:  CH1, CH2                        : CHAR;
           NAME                            : ALFA;
           OK, YES                         : BOOLEAN;
           I,J,K,NEXT                      : INTEGER;
           ENTRANT                         : PERSON;   }
           CLUB, ROOM                      : GROUP;    } See
HOTEL    : ARRAY[1..NROOMS]OF PERSON;                  } Sections
LIST     : ARRAY[1..100]OF INTEGER;                      3.4.2
MATRIX   : ARRAY[1..2, 1..10]OF INTEGER;                 3.4.3
LINE     : ARRAY[1..80]OF CHAR;
SCRATCH  : FILE OF INTEGER;
CODE     : FILE OF BYTE;
OLD,NEW  : FILE OF CHAR;
```

A component of an array variable is denoted by the variable followed by an index expression, which in this

implementation must yield an integer result. The index expression may itself contain a component of an array variable.

```
Examples:  LINE[J]:= CH2;
           CH1:=  LINE[MATRIX[I,J]];
```

3.6 Declarations of Procedures and Functions

3.6.1 Procedure declarations

A procedure declaration associates an identifier with a part of a program (a block) so that it can be activated by a procedure statement. The procedure heading (which immediately precedes the block containing any declarations and the statement part) specifies the identifier and any formal parameters.

```
Example:  PROCEDURE REWIND (TOREAD : BOOLEAN);
          BEGIN
            IF TOREAD THEN RESET(SCRATCH)
                      ELSE REWRITE(SCRATCH);
          END;
```

3.6.2 Function declarations

A function declaration associates an identifier with a part of a program (a block) which returns a value of simple type. The function heading specifies the identifier, any formal parameters, and the type of values returned by activation of the statement part of the function, which must include at least one assignment of a value to the function identifier, or the result will be undefined.

Examples:
```
FUNCTION NEXTCHAR : CHAR;
VAR      CH:CHAR;
BEGIN
  REPEAT READ(CH) UNTIL CH<> " ";
  NEXTCHAR:=CH;
END;

FUNCTION FACTORIAL (N:INTEGER): INTEGER;
BEGIN
  IF N=0  THEN FACTORIAL:= 1
          ELSE FACTORIAL:= N*FACTORIAL(N-1);
END;
```

3.6.3 Parameters

In this implementation, parameters of procedures and functions are of the value kind. This means that when the procedure or function is activated, an expression will be evaluated and the resulting value will be used as the initial value of the parameter identifier.

The parameters in a procedure or function declaration are known as formal parameters, which are treated as local variables within the block.

The number of actual parameters (the expressions substituted for the formal parameters when the block is activated) must be the same as the number of formal parameters and the type of each actual parameter must be the same as the formal parameter type.

The P-6800 PASCAL compiler detects and reports cases where the number of actual parameters is incorrect, but it does <u>not</u> check their type.

3.6.4 Standard procedures and functions

These procedures and functions are pre-declared in the P-6800 PASCAL compiler. No conflict will arise if

WRITE(FILENAME,E1,E2,..EN) Write on to the file specified (default is OUTPUT), the values of the expressions listed.

WRITELN (as above) Write as above and then execute as WRITELN.

3.6.4.2 Conversion and Format Control

If the value of an expression is to be written on to a file of the same type as the expression, or a value is to be read from a file for assignment to a variable of the same type as the file, a simple copy of the required number of bytes will take place, with no conversion.

Integer variables may be written on to and read from files of bytes, and vice versa, provided that the values of the integers are in the subrange 0..255. If this condition is violated, a run-time error will occur.

If an integer or byte value is to be read from or written on to a file of characters (e.g. standard input or output), the necessary conversion is done automatically. Spaces or a carriage return may be used to separate or terminate values on an input file. Values being output are formatted into a field width of 6 characters, with leading zeroes being replaced by spaces. The default width may be changed by a format control; if this is insufficient to print the value (including the sign, if negative) the field will be filled with asterisks. If the following example, K is to be printed in the default of 6 columns, L in 2 and M in 8 columns.

```
e.g.  WRITELN (K,L:2,M:8);
```

a declaration within a program redefines the identifier, since the scope rules (see Section 3.2) apply.

3.6.4.1 File handling procedures

RESET (FILENAME)	Resets the current file position to its beginning. This is a necessary initialising operation before reading any file other than the standard INPUT file.
REWRITE (FILENAME)	Discards all items previously appended to the file, so that a new file may be generated. This is a necessary initialising operation before writing onto any file other than the standard OUTPUT file.
READLN (FILENAME)	Skip (up to and) over the end of the current line to the first character of the next line. This may only be applied to files of CHAR, including INPUT. If the parentheses and the filename are omitted, INPUT is assumed.
WRITELN (FILENAME)	Terminate the current line and prepare to write the next character in the first position of the next line. This applies only to files of CHAR, for which OUTPUT is the default.
READ(FILENAME,V1,V2,..VN)	Read from the file specified (default is INPUT), values for the variables listed.
READLN (as above)	Read as above, and then execute READLN.

When characters (variables or literals in quotes) are printed, the default field width is 1; for ALFAs it is 6, and for text strings (e.g. "THIS IS A STRING") is the length of the string. A wider field width may be specified, in which case the characters will be right justified in the field.

e.g. WRITE (CH:3, "-":3, "VOWEL":10);

3.6.4.3 Other standard procedures

(Note that these procedures are standard only in the sense that they are predeclared within the compiler. They are not included in "standard PASCAL").

HALT	Terminate the program execution prematurely.
POKE(Address, Value)	This permits users to assign a value to a specified byte address rather than to a variable identifier. The first parameter is the address (in decimal), and the second is the byte or integer value in the range 0..255.

3.6.4.4 Standard Ordinal Functions

ORD(P)	The parameter P may be of type character, or scalar (enumerated type). The value returned will be an integer. If P is of type CHAR, the value will be the 7 bit ASCII code for that character. If P is a scalar, the value will be determined by the order in which the identifiers were enumerated (e.g. in 3.4.2 ORD(ANN) is 1).

CHR(N) — This yields the character from the ASCII set, whose ordinal (7 bit code) is equal to the value of the expression N. For any such character, the following relationship holds:

```
CH = CHR(ORD(CH));
```

SUCC(X) — The parameter X must be of scalar (enumerated) type. The function then returns the scalar whose ordinal is one greater than that of X. (e.g. in 3.4.2, SUCC(ANN) is BOB).

PRED(X) — As for SUCC(X), except that the value returned is one less (e.g. PRED(BOB) is ANN).

3.6.4.5 Standard Predicate Functions

ODD(N) — Gives the boolean value true as a result if the integer expression N is odd, otherwise the result is false.

EOF(F) — This gives the value true after the end of file F has been reached, otherwise it is false. If the file-name and the parentheses are omitted, the standard input device is assumed. Execution of a RESET or REWRITE statement clears the end-of-file condition. Under the FLEX operating system, EOF(F) will be true for a file of characters when a read statement is attempted after the last character on the file has been read.

For a binary file (i.e. not ASCII characters), EOF(F) is not set true until the end of the last sector of the file is encountered by a read statement. (See Section 4.2.2).

EOLN(F) Gives the value true when a carriage return character was encountered by the last read operation, and false otherwise. The procedure READLN(F) sets the EOLN condition to false. If the filename F and the parentheses are omitted, the standard input device is assumed.

3.6.4.6 Other Standard Functions

(These are standard only in the sense that they are predeclared within the compiler).

PEEK(Address) This function returns the value of the content of the location specified by the integer parameter expression "address". Since it is in the range 0..255, it may be assigned to a variable of type byte or integer.

USER(P1,...) This function must have at least one parameter, and returns an integer result. For further details see Section 7.3.

3.7 Expressions

Expressions consist of operators and operands (constants, variables and function designators). If an operand has an undefined value (because none has been assigned to it) the resulting expression will have an undefined value, and an error may occur during its evaluation.

From the syntax diagrams of Appendix A,it may be seen that an expression comprises 1 or 2 simple expressions. A simple expression is made up from 1 or more terms, and a term is made up from 1 or more factors. A factor may be any of the following:

- an unsigned constant
- a variable
- a function identifier (with its parameter list, if any)
- an expression enclosed by parentheses
- a factor preceded by the NOT operator
- a list of scalar identifiers, separated by commas, all enclosed by brackets
- a subrange (of byte or character) expression.

When an expression is evaluated, the order in which its components are evaluated is determined by the operator precedence rules given on the next page. Where operators have equal precedence, evaluation is from left to right.

OPERATOR PRECEDENCE

Highest	NOT	Used with Boolean factors
Second	the multiplying operators *DIV AND	Used between factors
Third	the adding operators + - OR	Used between terms
Lowest	the relational operators =,<>,<,>, <=,>=,IN	Used between simple expressions

In the following table listing the possible operator/operand combinations, "integer" also implies "byte".

PERMISSIBLE OPERATOR/OPERAND COMBINATIONS

OPERATOR	OPERAND TYPE	RESULT TYPE
NOT	Boolean	Boolean
*	Integer, Set (intersection)	Integer, Set
DIV	Integer	Integer
AND	Boolean	Boolean
+	Integer, Set (union)	Integer, Set
-	Integer, Set (difference)	Integer, Set
OR	Boolean	Boolean
=,<>	Integer, Char, Alfa, Scalar, Set	Boolean
IN	See note below	Boolean
Other relationals	Integer, Alfa	Boolean

Note: In this implementation, the operator IN may be used to test whether a scalar value is in a set, whether a character is in a specified subset of the ASCII characters (e.g. CH IN["A".."Z"]), or whether an integer known to be in the subrange 0..255 is in some smaller subrange e.g. J IN[10..20].

The following examples serve to illustrate the application of the aforementioned rules.

Factors	21	NOT PRIME(P)
	J	[ANN,BOB]
	PRIME(P)	[1..200]
	(J+K)	["0".."9"]

Terms

```
J*K
K DIV L
PRIME(P) AND (P<200)
```

Simple expressions

	K+L	K-L+1
	-K	PRIME(P) OR ODD(P)

Expressions

```
K=1
J<>2
THISCOLOUR IN [RED,BLUE,GREEN]
```

In this implementation, a Boolean expression is evaluated completely even if its value could have been determined by partial evaluation.

If a function designator (which denotes the activation of a function denoted by the function identifier) contains a list of parameters, they are evaluated from left to right.

The compiler checks that the number of actual parameters agrees with the function definition, but no checking is performed on the type of the parameters.

3.8 Statements

Statement denote algorithmic actions, and are said to be executable. They may be simple or structured.

3.8.1 Simple Statements

A simple statement may be an assignment statement or a procedure statement. No part of a simple statement constitutes another statement. An assignment statement causes the current value of a variable or function identifier to be replaced by the value of an expression. The variable or function identifier must be of the same type as the expression.

```
Examples:  NEW:= OLD+4;              (Integer types)
           ANSWER:= "N";             (Character)
           PRINTING:= ANSWER="Y";    (Boolean)
```

A procedure statement serves to execute the procedure denoted by the procedure identifier. If the procedure definition included formal parameters, then the correct number (and type) of parameters must be included in the procedure statement. The compiler checks the number of parameters, but does not check their type.

In this implementation, only value parameters are permitted. This means that when a procedure statement is executed, the current value of the parameter expression is evaluated and assigned to the formal parameter variable (which is regarded as local to the procedure block) as its initial value.

```
Examples:  CLEAR;
           QUICKSORT (LOW,HIGH);
           REPORT ("A", COUNT*COUNT);
```

3.8.2 Structured Statements

These are constructs which are of the following kinds:

(a) compound statements, comprising one or more statements to be executed sequentially.

(b) conditional statements, which select one of their component statements for execution.

(c) repetitive statements, which specify the number of times that certain statements are to be executed.

3.8.2.1 Compound Statements

The statements comprising a compound statement are executed in the same sequence as they are written. They are bracketed by the symbols BEGIN and END, and are separated by semicolons.

Example:
```
BEGIN
   J:= J+1;
   SIGMA:= SUM(J,K,L)
END
```

3.8.2.2 Conditional Statements

3.8.2.2.1 The IF Statement

This statement may have either of the following forms.

```
IF condition THEN statement;
IF condition THEN statement 1 ELSE statement 2;
```

in both of which the "condition" part is an expression giving a Boolean value.

Example: `IF MAX<NUM THEN MAX:= NUM;`

The syntactic ambiguity arising from

```
IF expression 1 THEN
IF expression 2 THEN statement 1 ELSE statement 2
```

is resolved by regarding the second line as being bracketed by the symbols BEGIN and END.

3.8.2.2.2. The CASE Statement

This statement consists of an expression (the case index or selector) and a list of statements, each being labelled by a constant of the same type as the case index expression. In this implementation, the types permitted are:

(a) character

(b) scalar (enumerated type)

(c) byte (or integer in the subrange 0..255)

All the case labels must be distinct (unique), and if the index expression is equal to none of them, a run-time error will result.

The labelled statements are separated by semicolons - the last one being terminated by the symbol END.

Examples:

```
CASE OPERATOR OF          CASE   CH  OF
PLUS  : N:=I+J;           "A" : WRITE("ALPHA");
MINUS : N:=I-J;           "B" : WRITE("BRAVO");
TIMES : N:=I*J            "C" : WRITE("CHARLIE")
END;                      END;
```

3.8.2.3 Repetitive Statements

3.8.2.3.1 The FOR Statement

This should be used when a statement is to be executed a number of times which can be determined before the repetition commences.

A control variable (which in this implementation must be of type integer) is used to count the number of repetitions. Its initial and final values are evaluated before the statement body is executed for the first time, but its value is undefined after the statement body has been executed for the last time. Within the statement body, the control variable may be used, but it must not be changed by any statement invoked by that body.

The FOR statement has 2 forms:

```
FOR CV:= low TO high DO statement body;
FOR CV:= high DOWNTO low DO statement body;
```

In the first form CV is incremented by 1 after the body has been executed; in the second form CV is decremented. The body will not be executed at all if low>high in the first form, or if high<low in the second form of the FOR statement.

Example: `FOR J:=MIN TO MAX DO LIST[J]:=0;`

3.8.2.3.2 The WHILE Statement

WHILE expression DO statement body.

The statement body is repeatedly executed while the Boolean expression has

the value "true". If its value is false at the beginning, it will not be executed at all.

```
Example:  WHILE CH=SPACE DO READ(CH);
```

3.8.2.3.3 The REPEAT Statement

```
REPEAT statement body UNTIL condition.
```

This statement is similar to the WHILE statement, except for the fact that the Boolean condition expression is evaluated at the end of the statement body. Consequently it will always be executed at least once.

```
Example:  REPEAT
            J:= J+1
          UNTIL LIST[J]= THATVALUE;
```

3.9 Programs

A PASCAL program has the form of a procedure - a heading followed by a block, which is terminated by a period.

The heading consists of the symbol PROGRAM, followed by an identifier which is the program name, optionally followed by a list of file identifiers, and separated from the main program block by a semicolon.

The program name identifier is ignored by the P-6800 compiler, but it must nevertheless be present in the heading. The file identifier list need only be included if files other than the standard INPUT and OUTPUT are to be used by the program. INPUT and OUTPUT are always present as FILEs of CHAR, since they are pre-declared within the compiler. If they are included in the file identifier list,

they will be ignored. The order in which the files are listed in the program heading is the order in which they should be specified in the RUN command (see Section 4.2). The compiler assigns logical file numbers (LFN) to them, which are used in the P-code instructions generated by the compiler as the means of identifying the files to the run-time system. Inclusion of the file identifiers in the program heading merely serves to tell the compiler that files bearing these names will be declared within the program which follows. It is in the subsequent declaration that the compiler is told what type of data will be written on to or read from those files.

Example:
```
PROGRAM SHORT (SCRATCH);
VAR
  SCRATCH : FILE OF CHAR;
BEGIN
  REWRITE(SCRATCH);
  WRITE(SCRATCH,"A TEXT STRING FOR THE
          SCRATCH FILE");
END.
```

4. THE RUN-TIME SYSTEM

The P-6800 PASCAL system has been designed with a broad spectrum of users in mind. The following sections cover the basic characteristics and use of the system.

4.1 What is the P-6800 Run-Time System?

The task of the run-time system is to provide a complete support environment for the execution of P-code object programs. Viewed from the top it looks like a stack oriented computer with a very specialised and high level instruction set. Viewed from below it is seen to be M 6800 machine code which makes frequent calls to FLEX and sometimes user written assembler routines.

Fig. 4.1 shows a perception of what your M 6800 system looks like when executing FLEX. FLEX only looks after the disk drives and the system console(s), and being a program has need of access to the CPU to do its job. Fig. 4.2 illustrates the initial effect of typing the simple command line

RUN,PROGRAM<CR>

on the console. FLEX loads the run-time system and gives it control of the CPU. The run-time system first determines how much contiguous memory is available; scans the command line for parameters and treats the first one as the file name of a P-code binary file. The run-time system attempts to open the P-code file and read the first block as this contains information about the size of the program and its functional requirements. Next the run-time system asks if the user wants to change the size of the stack reservation (see Section 5.1.4) and based on the response determines whether there is enough contiguous memory to fit everything in.

If there is sufficient memory the program is loaded directly at the end of the run-time system otherwise the paging subsystem is first installed and program pages only loaded on demand. In any event the final situation is portrayed in Fig. 4.3. The run-time system then starts interpreting the P-code.

During program execution run-time tests are constantly being performed by the system in an attempt to prevent a user program from generating bad results. The errors detected by the run-time system are all terminated with *** and are in plain English. They are listed in Appendix C. Following the error message, is printed the value of the pseudo program counter. This can be correlated directly with the code index value printed on the left-hand margin of the compiler listing so that finding where the error occurred is really easy, finding the why is your problem!

In the event that the program memory and run-time system have not been corrupted by the error AND the program was NOT running in page mode, the RESTART utility provided with your P-6800 PASCAL system can be invoked. Typing

RESTART<CR>

causes all the pseudo registers to be reset, a jump to be made to the user initialisation routine (see Section 7.2) if present and execution of the P-code program started from the beginning.

Note The normal FLEX conventions are adhered to in that the P-code binary file, if specified only by name, will be assumed to exist on the current working drive and have extension .BIN. The exception to this rule is the filename PASCAL which is recognised to be the compiler and is assumed to reside on the same drive as RUN unless explicitly declared otherwise, i.e. RUN,2.PASCAL.

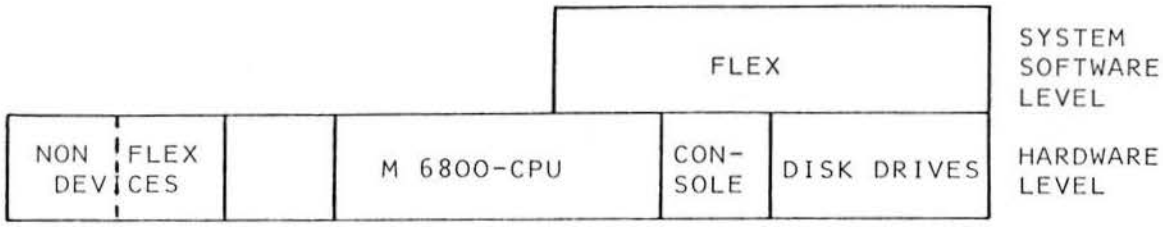


Figure 4.1 Idle State

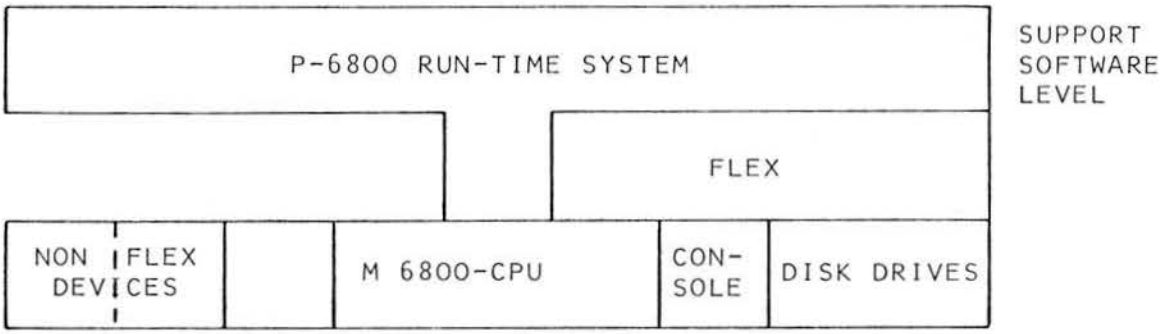


Figure 4.2 Run Initialisation

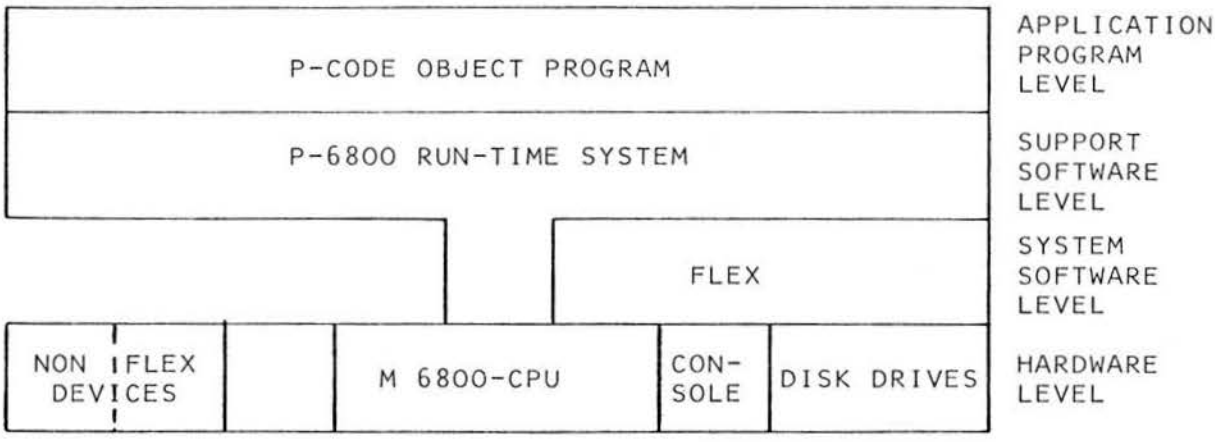


Figure 4.3 Program execution

4.2 Run-time System Support of Disk Files

4.2.1 Association of Disk Files with Program File Identifiers

In Section 3.9 we introduced the notion of the logical file number (LFN). This notion is nothing to do with PASCAL and is only used as a means of describing file assignments. The standard files INPUT and OUTPUT are predefined, ever present files of CHAR, always take the logical file numbers 1 and 2 which always refer to the system console keyboard and display device. If files are declared in the program statement then they are allocated LFNs 3,4,5 etc. in order of declaration. The association between real named disk files and LFNS is by the position of the file specification in the command line.

Consider the following command line

```
RUN,PROGRAM,F3,F4,F5...Fn
```

Fn is a FLEX disk file specification which will be associated with LFN#n. If more file specifications are provided than are actually declared in the program (see Section 5.1.2) memory is wasted because a file control block (FCB) is reserved at load time.

If fewer file specifications are provided than are actually declared in the program then the missing disk files default to the system console. As all LFNs are associated by default with the system console, disk files F3 and F6 in the following example will be associated with LFNs 3 and 6 while the console will be associated with LFNs 4 and 5.

```
RUN,PROGRAM,F3,,,F6
```

Where only a filename is given the default drive is the current working drive and the default extension is .TXT.

4.2.2 Limitations

The EOF (LOGICAL FILE) function relies on the status returned by the file management system (FMS) when the file is a disk file. In the case of a FILE of CHAR the end-of-file and end-of-information are the same, as FMS skips any zero bytes used to fill out the last sector as being non-printing characters. In this case EOF is true after the next read operation. In the case of FILE of BYTE or INTEGER however the data is treated transparently by FMS and the end of file indication is only given at the end of the last sector of the file. This is likely to occur some time after the end of recorded information. Our advice is structure your binary data so you know where the end is.

Version 1.1 and 1.2 of the run-time system limit the number of files to eight including standard INPUT and OUTPUT. Thus up to six file specifications are allowed after the P-code filename in the command line.

WARNING

It is worth noting the different effects produced by the two file procedures RESET and REWRITE.

> RESET (LOGICAL FILE) performs the FLEX operations CLOSE (if open), OPEN for read, on the associated file specification.
>
> REWRITE (LOGICAL FILE) performs the FLEX operations CLOSE (if open), DELETE (with no "Are you sure?"), OPEN a new file for write with the associated file specifications.

If you get your files in the wrong order you can rewrite your data out of existence.

5. UTILISATION OF COMPUTER SYSTEM RESOURCES

The purpose of this chapter is to provide guidelines to the user for calculating the resource requirements of the P-6800 system and his program.

5.1 Memory Requirements

5.1.1 Run-Time System

The run-time system is organised as a nucleus, which contains all the pseudo registers, tables, flags and service code for the basic logical, arithmetic and branch operations, and modules which load on top of it to support optionally integer, set and alfa operations. The nucleus occupies about the first 3K of memory including most of page zero (see file RUNEQU.TXT) and slightly less than 4K with all support modules. Run-time system memory is never paged but if the program has to be then memory is reserved for the necessary disk address tables which may be several hundred bytes in length.

5.1.2 File Buffers

An FCB is reserved for each file specification found in the command line that invoked the run-time system. Under version 1.1 this amounts to 192 bytes per file and 320 under version 1.2. The P-code file however is assigned to the system FCB to conserve memory.

Note Although the PASCAL compiler can be invoked simply by the command

RUN,PASCAL,PROGRAM

two additional disk files are used which are automatically generated by the run-time system for the compiler. Therefore three FCBs are always reserved and an additional one if a LISTING file is specified.

5.1.3 Estimating the P-code Memory Requirement

It is difficult to forecast exactly how many bytes of P-code will be produced by the compilation of a P-6800 PASCAL program, but it is possible to make rough estimates. From the many programs which have been compiled to date (including the compiler itself), the following rules-of-thumb have evolved:

(a) 3 P-code instructions (12 bytes) are generated per line of source code (excluding comments).

(b) 1 P-code instruction (4 bytes) is generated for every 12 characters of source program (including spaces used for indentation).

(c) The P-code binary file occupies half the number of sectors required to hold the source program as a TXT file.

As compilation proceeds (with listing) a code index is printed at the start of each line. The value of this index is the number of bytes of P-code already generated, when the compiler started to process the current line. It is therefore possible to deduce how many bytes of code a particular statement generated. For further details on this aspect, and how to "fine tune" the code generated,see chapter 6.

5.1.4 Estimating the Stack Size

The stack is the area of memory reserved for the data used by the P-code instructions and the run-time system. These data include single variables, arrays of variables, the activation records for blocks (procedures and functions), and the values of partially evaluated expressions.

For each variable of type character, Boolean, byte or scalar, 1 byte is reserved in the stack. For other types the requirements are:

integers	2 bytes
alfas	6 bytes
sets	8 bytes

The space required for an array is simply the number of bytes for a single variable of the appropriate type, times the number of elements in the array.

Each activation record occupies 6 bytes, so each time a procedure or function block is activated from within another, the stack grows by 6 bytes - plus the space required to hold the results of evaluating any parameters, and the space for any variables declared locally (within the block). The space is returned (i.e. the stack shrinks again) upon leaving a block.

The total space required for the stack is therefore easy to calculate - provided that no block is called recursively. If recursion does take place, it can be difficult to estimate how many nested activations will occur, and, consequently how large the stack should be to provide sufficient space.

The run-time system checks that there is sufficient space left in the area reserved for the stack each time a block is activated, and more space is required. It does not check before using the stack to store intermediate values during expression evaluation.

5.2 Disk Requirements

5.2.1 Paging Mode

If a program (the compiler is a program too) is too large for the available memory the run-time system automatically enters paging mode. This means in practice that program pages are read in a random fashion from the P-code file. However many sequences of code in programs are sequential in nature and consequently cause an effective sequential read over parts of the P-code file. Experience has shown that if a P-code file is very fragmented i.e. created on a much used disk, then the paging performance is reduced. This is particularly noticeable when the compiler has to be paged. The sound of a badly fragmented compiler file is very "chirpy". The recommendation is to reformat a system disk and instal the compiler (or other large program) as the last system file. This keeps the compiler together and also minimises the head movement necessary during compilation, when moving between the P-code file and the scratch file used by the compiler.

5.2.2 The PASCAL compiler's files

The program of the PASCAL compiler declares four files in addition to standard INPUT and OUTPUT. Their order on the program statement is SOURCE,LIST,BINARY,SCRATCH. If it were not for the run-time system recognising the name PASCAL the user would have to type quite a long command line. As you already know it is only necessary to type RUN,PASCAL,PROGRAM to compile a PROGRAM.TXT onto a file PROGRAM.BIN.

If the user requires different named files for the binary or if a LISTING file is wanted for later printing then the user should study the following. Assume that the system drive is drive 0 and the working drive is drive 1 then the tokens in parentheses are "provided" by the run-time system.

```
RUN,(0.)PASCAL(.BIN),(1.)PROGRAM(.TXT,,1.PROGRAM.BIN,
                                 0.PROGRAM.SCR)
```

Note The scratch file is assigned to the system drive. It will be about the same size as the final binary so space should be available.

Normally a user will only wish to generate a listing file in addition to the binary hence

```
RUN,PASCAL,PROGRAM,LISTING
```

will produce a binary file 1.PROGRAM.BIN which can be directly executed by

```
RUN,PROGRAM (assuming that the compiled
             program uses no files)
```

and a listing file 1.LISTING.OUT which can be simply LISTed under FLEX 1.0 or batched to the spooling system under FLEX 2.0.

Note The extension .OUT is always forced if a list file is specified. The compiler produces a listing by default on the console. If a minus(-) is concatenated with PASCAL, i.e. PASCAL-, then the listing feature is suppressed and compilation is faster.

5.3 Interrupts

The P-6800 system does not use interrupts in its operation neither does it prevent them. The user is free to make use of the interrupt features of the M 6800 so long as the contents of the machine stack are not damaged.

6. FINE TUNING OF P-6800 PASCAL PROGRAMS

6.1 Introduction

This chapter contains a few hints on how to select from the various facilities available in this implementation of PASCAL, those which will help to economise on storage and/or ensure fast execution.

An important consideration when writing a PASCAL program (or any other) is clarity. The easier a program is to understand, the more likely it is to be correct, and the easier it will be to identify parts of a program which would be judged inefficient (or event redundant) in any programming language.

Most of the P-6800 PASCAL programs you write will probably occupy between a few hundred and a few thousand bytes, the stack (containing the data areas), will require a similar amount, and the programs will probably run quite fast enough for normal purposes.

However, there may be occasions when it is desirable to ensure that the clear, well-written program is as compact and as fast as possible. The following information is provided to help you achieve these aims.

6.2 The P-code Instructions

The P-6800 PASCAL compiler generates P-code in fixed length "instructions" of 4 bytes. The run-time system spends a certain amount of "overhead" time decoding each instruction, before executing the appropriate assembler instructions. This time taken to decode the instruction

is almost independent of the instruction, while the time taken to execute the appropriate assembler instructions depends very heavily on the P-code instruction. For example, an unconditional jump is much faster than loading a subscripted variable on to the stack.

Most P-code instructions are "pure" instructions, but some include a constant value such as a character, a scalar ordinal, or an integer - to be used as an operand (which does not then need a separate instruction for it to be loaded on to the stack). Other P-code instructions are really two-in-one, which saves the 4 bytes which would be required for a separate instruction, as well as the time overhead required to decode that instruction.

The following information should enable you to select those programming constructs which will be compiled into the more compact, and in general, faster P-code instructions.

6.3 Constants

The instruction to load a constant value on to the stack is faster than the one to load the current value of a variable. Constants (either defined separately or included in a statement as a literal) should therefore be used in preference to a variable identifier containing a constant value. A constant following an operand is incorporated into the instruction. Consequently K+2 occupies only 2 instructions while 2+K occupies 3.

6.4 Subscripted Variables

Although a variable may have up to 7 subscripts, it is unusual for more than 1 or 2 to be used. Consequently the run-time system has been optimised to give particularly

efficient processing of singly subscripted variables. Those with 2 subscripts, while not processed as efficiently as those with only 1 subscript, are preferable to those with 3 or more. Faster execution will therefore result if a few vectors are used rather than a 2 dimensional array.

The compiler can handle up to a total of 30 sets of array dimensions (lower/upper bound pairs) in a program, and error 91 will be reported if this number is exceeded. However, the declaration

```
A,B,C,D : ARRAY[1..10]OF INTEGER
```

only requires one set to be stored, whereas if the arrays were declared separately, 4 sets would be stored.

6.5 Conditional Statements

The statement IF C1 AND C2 THEN S is compiled into 1 P-code instruction less than

```
IF C1 THEN IF C2 THEN S
```

which is usually (but not always) equivalent.

However, since Boolean expressions are evaluated completely even if the resulting value could have been determined after only partial evaluation, the second construct will usually be significantly faster, particularly where the probability of one factor being "false" is greater than that of the other, and it is evaluated first.

CASE statements are to be preferred to several consecutive IF..THEN..ELSE statements - in the interests of program clarity. However, in this implementation

they are compiled into more P-code instructions, so the P-code for the IF..THEN..ELSE construct is more compact and is slightly faster. The most likely cases should be listed first in order to ensure fast execution.

6.6 Repetitive Statements

Although the slight overhead associated with initialising a FOR loop construct causes a few more instructions to be generated than for the equivalent WHILE or REPEAT statements, it executes faster. Whether the difference in speed is significant will depend on the amount of time spent controlling the repetition, compared with the time executing the statement body. However, the null statement FOR J:=1 TO 30000 DO; executes twice as fast as the equivalent WHILE or REPEAT statement.

6.7 The Relational Operator IN

IF J IN[1..2] is compiled into 3 fewer P-code instructions than the equivalents

(a) IF (J=1) OR (J=2)

(b) IF (J>=1) AND (J<=2)

The P-code instructions generated from the IN subrange test are also relatively fast, so the use of this construct is recommended.

It can be particularly useful in "guarding" a CASE statement against a non-existent case,
e.g. IF CH IN["A".."F"]THEN CASE CH OF etc.

6.8 Format Control

If values of variables or strings of characters are to be widely spaced on a line, or simply printed on the right-hand side of the output medium, the format control

feature (see Section 3.6.4.1) is recommended. The field width specification is incorporated into the P-code write instruction, whereas the explicit specification of the number of spaces to be output generates additional P-code instructions, whether included in the text string (e.g. " ON THE RIGHT") or by means of a FOR loop.

6.9 The Use of BYTE Variables

The ability to declare variables to be of type BYTE (subrange of integer, 0..255) is provided to enable the user to economise on data storage (stack) requirements when it is known that integer arrays contain only small non-negative values.

All byte variables are converted into the normal 2 byte integer representation immediately after they are loaded on to the stack, so that the integer arithmetic P-code instructions can be used to operate on them. Similarly, an integer value on the top of the stack is converted into single byte representation (a run-time error will result if this is not possible) before it is stored as the new value of a byte variable.

The use of byte variables in a statement therefore causes extra P-code instructions to be generated, which occupy space and take time to execute. Consequently arrays should only be declared to be of type BYTE if there is insufficient space within the computer system for them to be declared as INTEGERs.

7. CUSTOMISING THE RUN-TIME SYSTEM

7.1 Overlays

The recommended method of customising the run-time system is by overlaying. This may be done by writing the desired code in mnemonic assembler with the help of the symbol definitions provided in the file RUNEQU.TXT. Producing a binary file via. an assembler and APPENDing this to the file RUN.CMD. Suppose the binary file produced is MINE.BIN then the command string

```
APPEND,RUN.CMD, MINE.BIN, MYRUN.CMD
```

would produce a customised version MYRUN, of the run-time system.

Two utility overlays are provided which enable useful variants of the run-time system to be produced. They are named X.OVL and C.OVL.

The command line

```
APPEND,RUN.CMD,X.OVL,RUNLGO.CMD
```

produces a run-time system called RUNLGO.CMD which does not prompt for stack size changes but goes straight into program execution.

The command line

```
APPEND,RUN.CMD,C.OVL,RUNSAVE.CMD
```

produces a run-time system called RUNSAVE.CMD which performs all the initialisation in the normal way but does not start executing the loaded P-code program. Instead it tells the user the area of memory to save in order to make a command file with the messages

```
SAVE FROM $0000 TO $XXXX
TRANSFER ADDRESS IS $NNNN
```

If the user now invokes the FLEX utility

```
SAVE,MINE.CMD,O,XXXX,NNNN
```

a directly executable command MINE is catalogued.

Note Any user written overlays should be appended to MINE not RUNSAVE if they use the command space of FLEX. MINE will call the user routine DEVINT (see Section 7.2) prior to execution of the P-code program.

7.2 Configuration Parameters

The following parameters are allowed to be changed with care to tailor the system to specific needs.

LIMIT contains a 16 bit value which is interpreted by the initialisation as the highest address + 1 of contiguous memory that is free for use, i.e. the end of stack for overflow testing purposes. If you want to reserve memory at the end of the available contiguous memory for your own routines then just overlay LIMIT with the address of the start of your reserved area. You can check this as the run-time system logs the usable memory size at initialisation. Default depends on version.

TXLF contains the ASCII value of the character which will be sent following the output of a <CR> to a terminal if declared as a FILE OF CHAR. Default <LF>.

RXLF contains the ASCII value of the character

which will be sent following the receipt of a <CR> from the terminal if declared as a FILE OF CHAR. Default <LF>.

DEVINT in the release system contains the 16 bit address of an RTS instruction. A JSR is made to this address by initialisation immediately before executing the loaded program. Therefore if your owncode or device drivers need initialisation, place the address of a suitable piece of code here and remember to exit with an RTS.

USRENT in the release system contains the 16 bit address of an RTS instruction. If you want to include a USER function place its address here (see Section 7.3 for details on USER).

DEVTAB is the device driver table. It contains eight pairs of 16 bit pointers, one pair for each logical device. The first is initialisation to FLEX GETCHR and the second to PUTCHR so all "device" I/O is by default through the console. If you have an interface on PORT#3 for example and decide to associate it with logical device#3 (you do not have to, its just aesthetic) then place the address of the input and output routines in the pair of pointers for device#3. By convention logical device#1 corresponds to PORT#1, i.e. the control PORT (see Section 7.4 for more detail).

NULLS — use is made of the FLEX field defined by TTYSET as NL, (number of nulls sent after <LF>) on output to terminals.

7.3 USER Function

Provided that the location USRENT has been correctly overlayed with the entry address of your own user function, then each time USER is executed in the compiled PASCAL program your code gets control.

The location MARKUS contains a 16 bit pointer which is the address of the base of the activation record for this call, Fig. 7.1. It should be pointed out that USER is pre-declared as an INTEGER function so care must be taken to clear the lower byte (MSB) if a byte sized quantity is returned. The six bytes following are not used for a USER call and can be freely used by the user, as can all locations higher than that pointed to by MARKUS. There is no limit to the number, type or order of the parameters that can be passed to the USER function so care must be exercised to make sure both the PASCAL program and the Assembler program agree. USER must be left via. an RTS instruction.

It is suggested that all user code be generated in one module as there is less chance of using different EQU values and of overlaying previous overlays erroneously. Fig. 7.2 shows conceptually what a fully customised system looks like.

7.4 Run-Time System Support of Non-FLEX Devices

Each logical file number (LFN) is associated with an entry in a table, the FILTAB which consists of two bytes per entry. These entries are initially set to #1 which

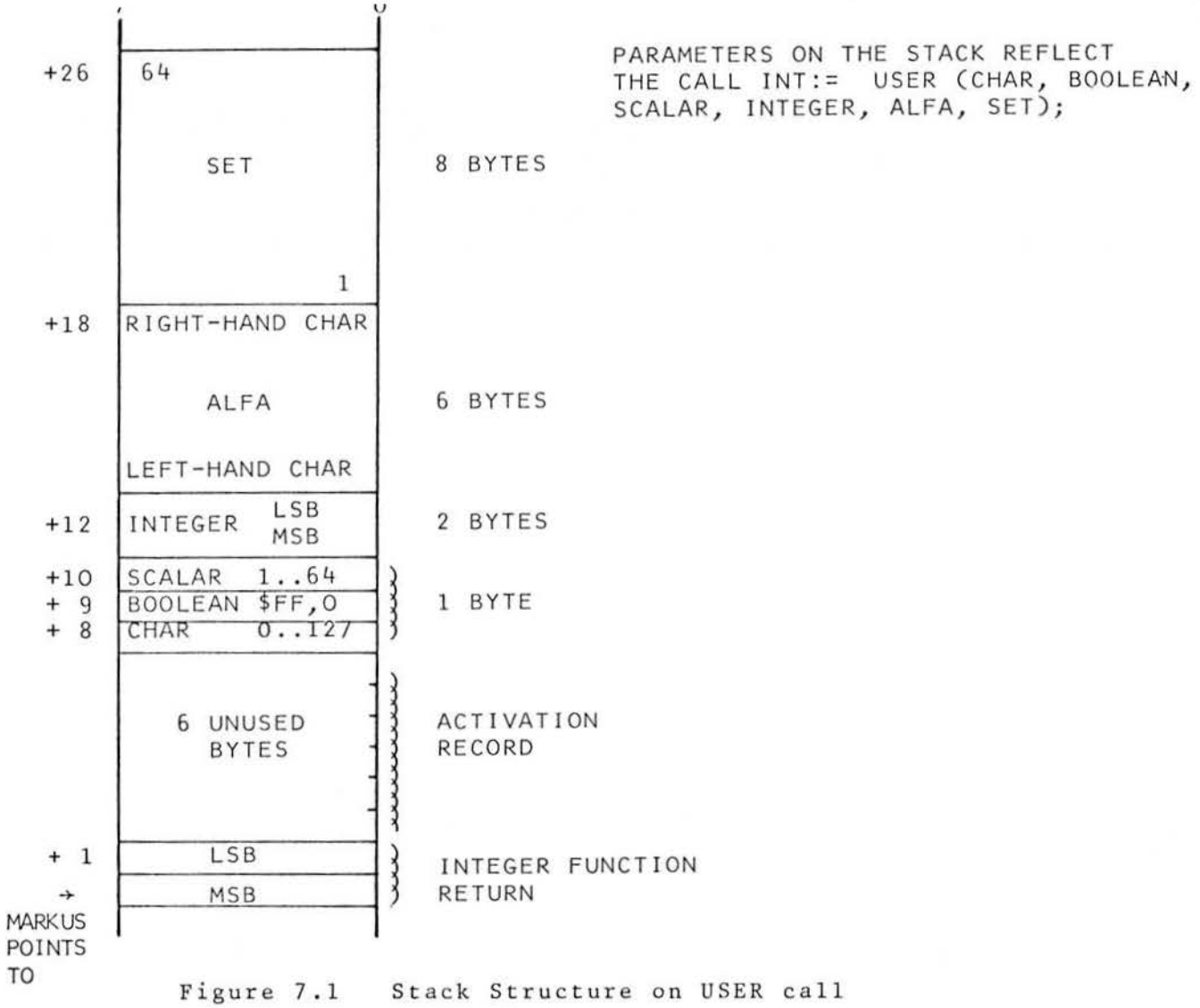


Figure 7.1 Stack Structure on USER call

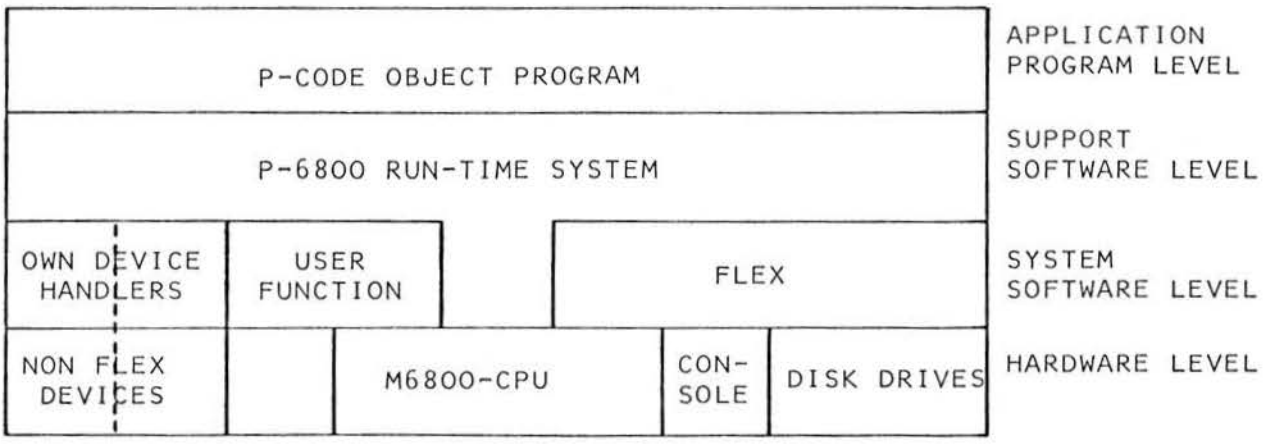


Figure 7.2 Fully Configured System

is the logical device number (LDN), chosen for the system console. During the initialisation phase of the run-time system, memory is allocated for an FCB for each file specification found in the command line and the address of the FCB placed in the associated entry in FILTAB. If no file specification is found for a particular position the default #1 remains. However if a command line such as

```
RUN,PROGRAM,F3,#2,#0,F6
```

is encountered, LDNs of 2 and 0 are associated with LFNs 4 and 5. These LDNs are written in the associated entries in FILTAB. The LDNs may be in the range 0..7 in the standard system and refer to the position of a four byte entry in a user accessible table DEVTAB. The first two bytes of each entry contain the address of a routine to input a single byte from the device associated with the LDN and the second two bytes the address of the output routine. Byte transfer is via. the A register and B and X are scratch. All DEVTAB entries are initialised to the FLEX GETCHR and PUTCHR entry points. This can be verified by experimenting with different logical files and LDNs.

Note If the associated logical file has been declared as FILE of CHAR, then the special characters <CR> and <CONTROL/Z> will be intercepted by the run-time system, converted to a blank and the appropriate EOLN or EOF flag for the file will be set. EOF implies EOLN.

If the file declaration is FILE of BYTE or INTEGER, no means is provided for communicating the EOF condition for use by the run-time system thus EOF (File associated with user device) is always false. However seven output routines and seven input routines can be interfaced so there is plenty of

scope for improvisation. LDN#1 is reserved for the system console usually on PORT#1 but no implicit association is made between any other LDN and PORT#.

The details of interfacing device handling routines is best illustrated by the example coded in the file OWNCODE.TXT.

WARNING

Do not declare a transfer address in your assembly otherwise RUN will not!

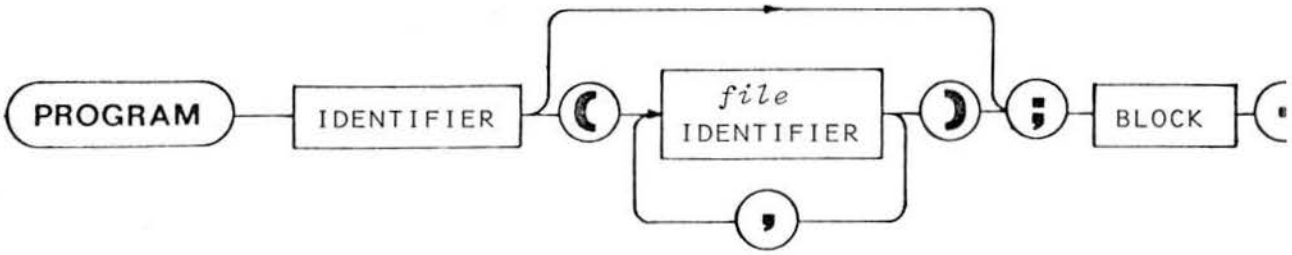
P-6800 PASCAL SYNTAX DIAGRAMS

The syntax diagrams on the following pages show, in pictorial form, the rules governing the syntax of the subset of PASCAL included in this implementation.

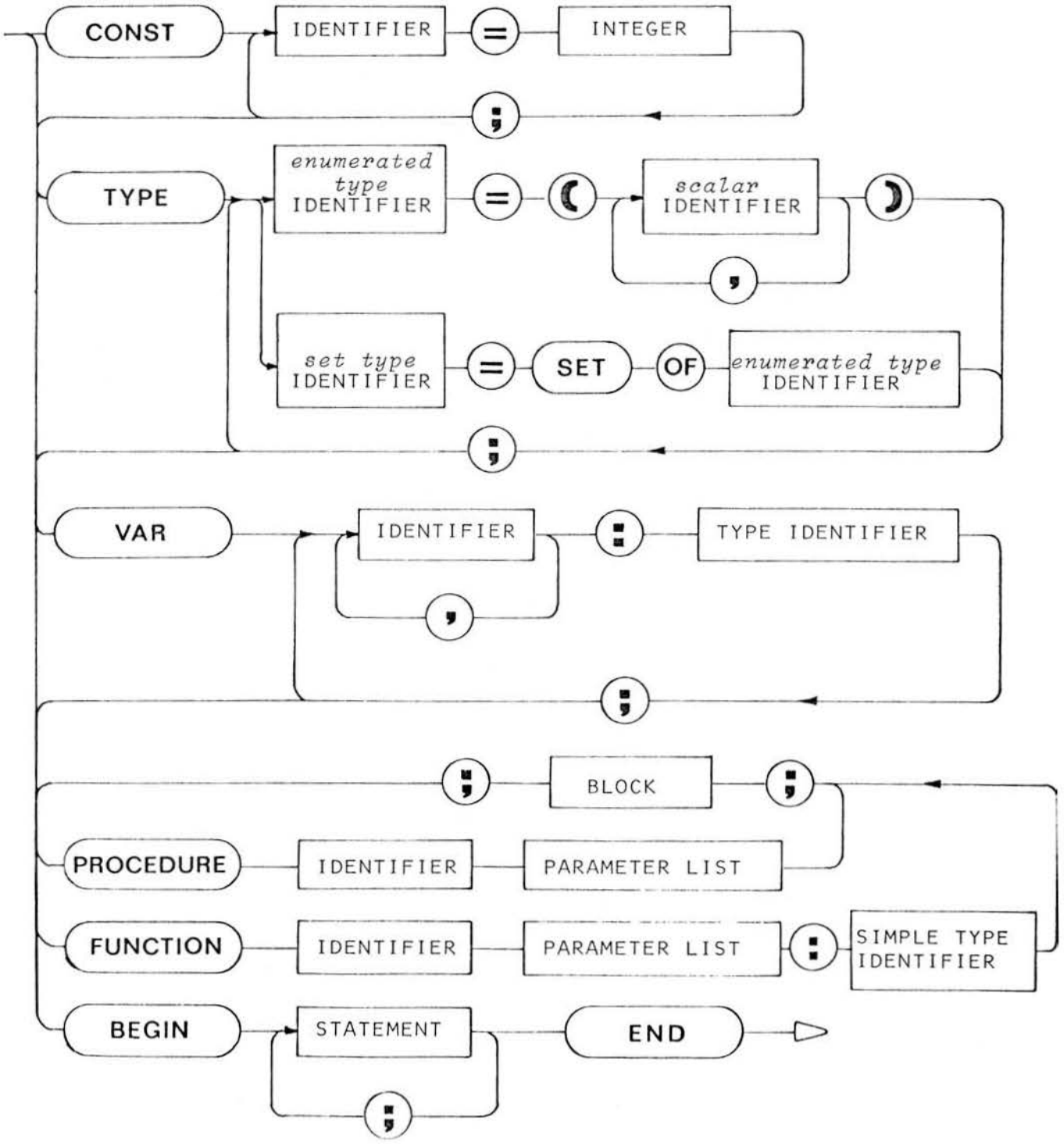
The diagrams should be read primarily from left to right, and from top to bottom. They may be followed in other directions where the curvature of the lines joining sections indicates that this is permitted. Terms shown in *italics* (e.g. *letter* are not defined on the diagrams, but their meaning should be self-evident.

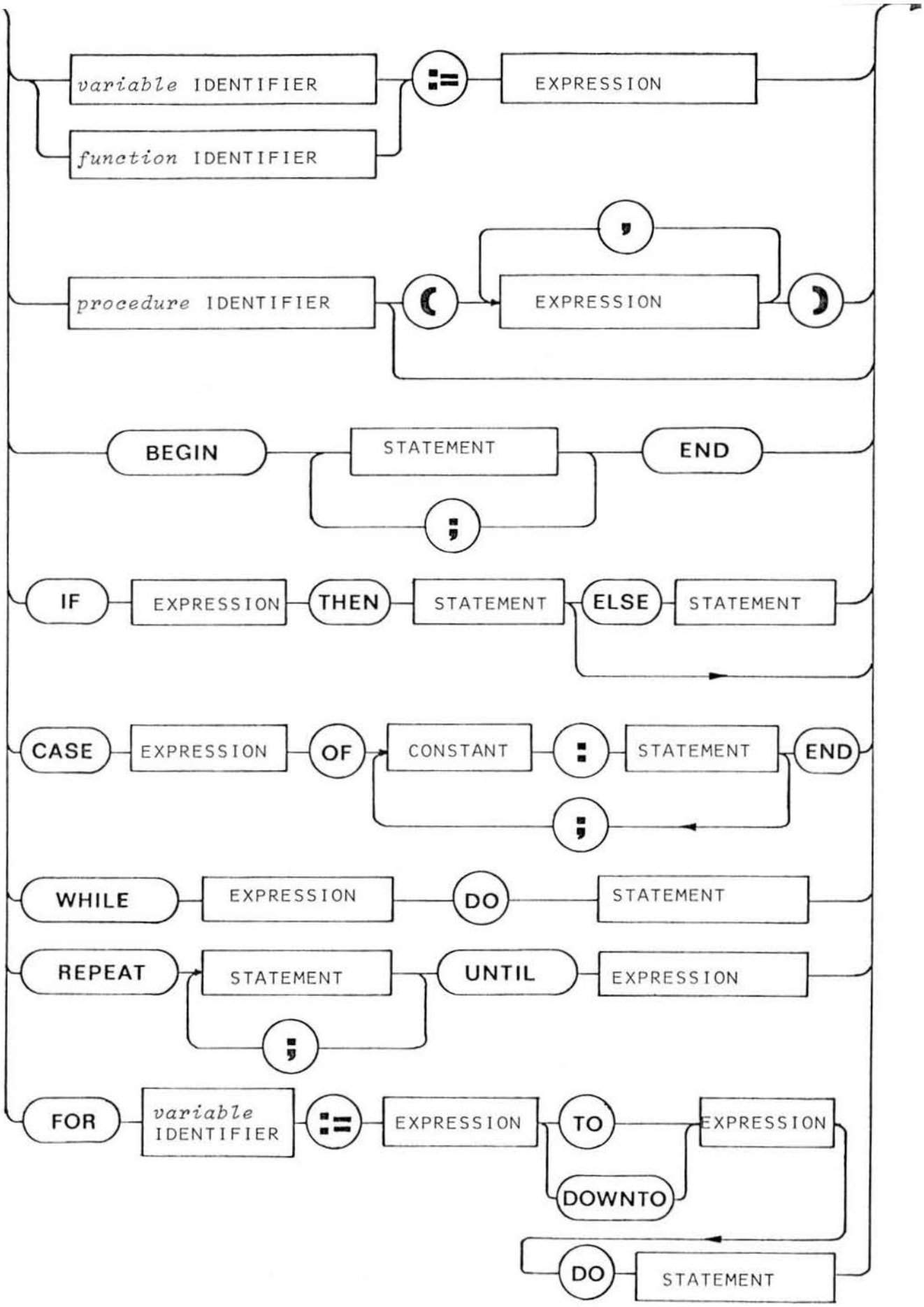
The following examples serve to illustrate the application of these rules.

1. A PROGRAM consists of the symbol PROGRAM, followed by an identifier (the program name), optionally followed by 1 or more file identifiers enclosed in parentheses, separated from the block by a semicolon. A program is then terminated by a period.
2. An UNSIGNED INTEGER consists of 1 or more digits.
3. An IDENTIFIER consists of a letter, optionally followed by any sequence of letters and/or digits.

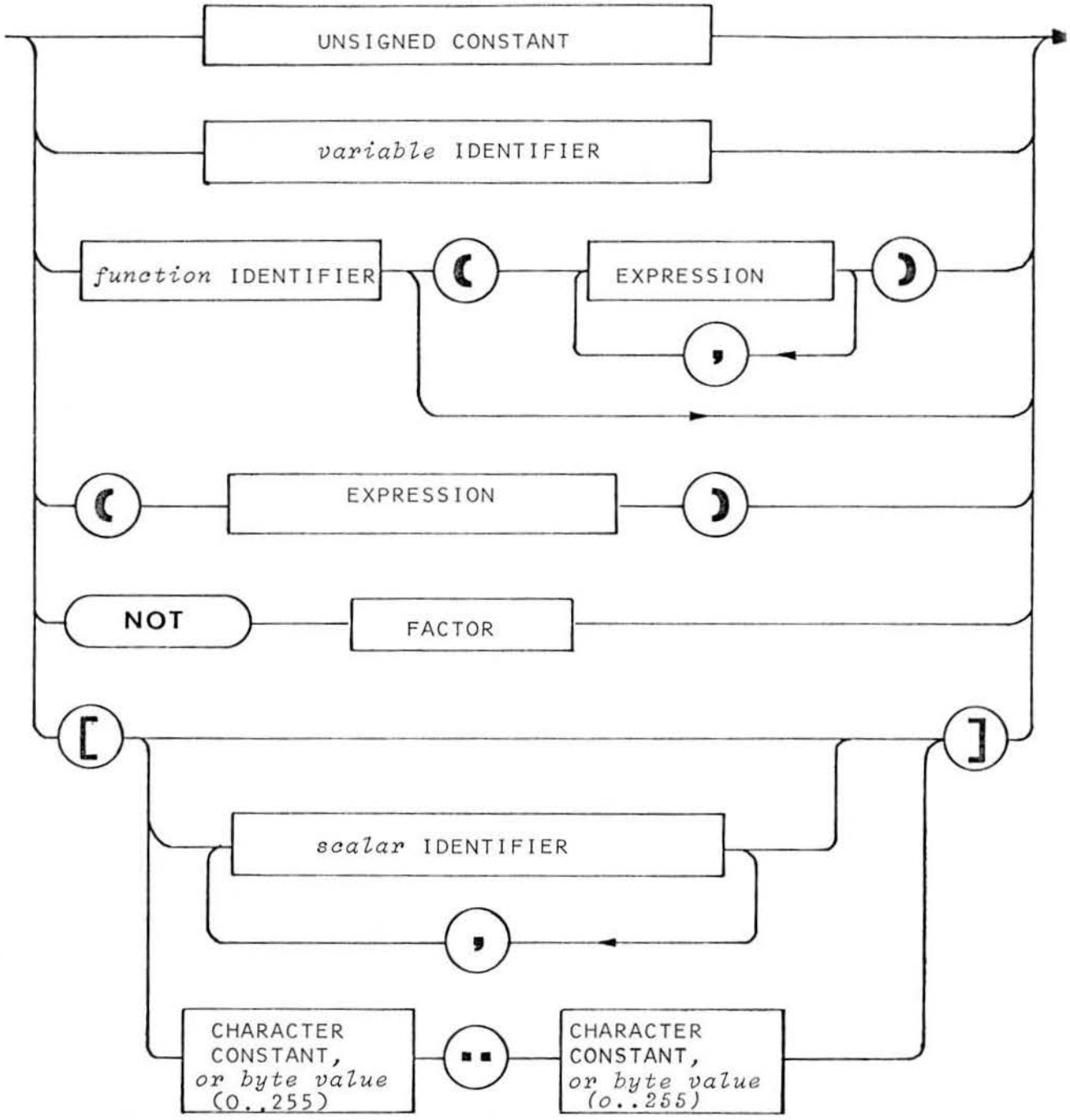

PROGRAM
IDENTIFIER
(
file
IDENTIFIER
,
)
;
BLOCK

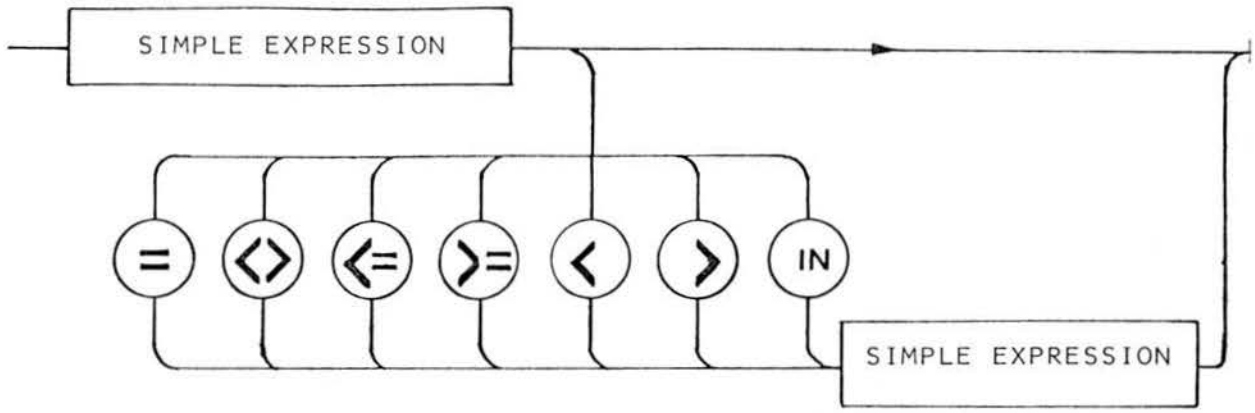
BLOCK


CONST
IDENTIFIER
=
INTEGER
;
TYPE
enumerated type IDENTIFIER
=
(
scalar IDENTIFIER
,
)
set type IDENTIFIER
=
SET
OF
enumerated type IDENTIFIER
;
VAR
IDENTIFIER
,
:
TYPE IDENTIFIER
;
;
BLOCK
;
PROCEDURE
IDENTIFIER
PARAMETER LIST
FUNCTION
IDENTIFIER
PARAMETER LIST
:
SIMPLE TYPE IDENTIFIER
BEGIN
STATEMENT
;
END

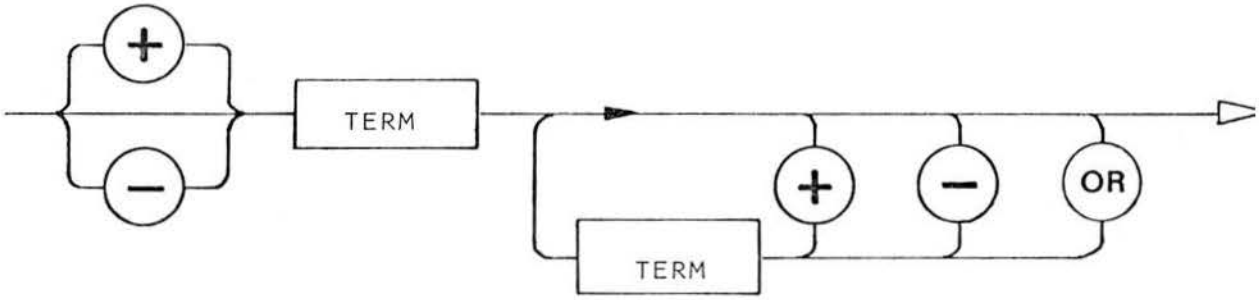

variable IDENTIFIER
:=
EXPRESSION
function IDENTIFIER
procedure IDENTIFIER
(
,
EXPRESSION
)
BEGIN
STATEMENT
;
END
IF
EXPRESSION
THEN
STATEMENT
ELSE
STATEMENT
CASE
EXPRESSION
OF
CONSTANT
:
STATEMENT
;
END
WHILE
EXPRESSION
DO
STATEMENT
REPEAT
STATEMENT
;
UNTIL
EXPRESSION
FOR
variable IDENTIFIER
:=
EXPRESSION
TO
DOWNTO
EXPRESSION
DO
STATEMENT

FACTOR

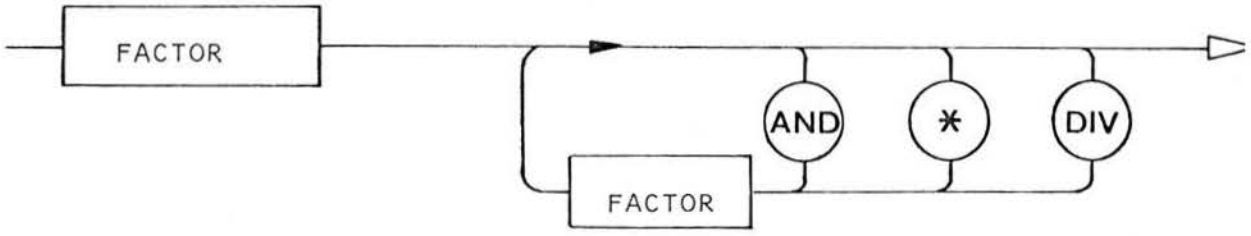

UNSIGNED CONSTANT
variable IDENTIFIER
function IDENTIFIER
(
EXPRESSION
)
,
(
EXPRESSION
)
NOT
FACTOR
[
]
scalar IDENTIFIER
,
CHARACTER CONSTANT, or byte value (0..255)
..
CHARACTER CONSTANT, or byte value (0..255)

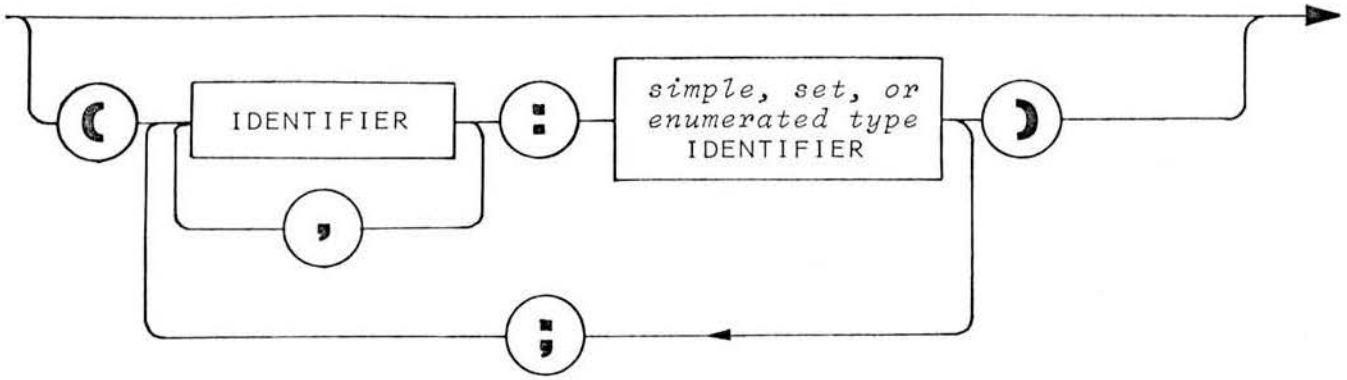


SIMPLE EXPRESSION

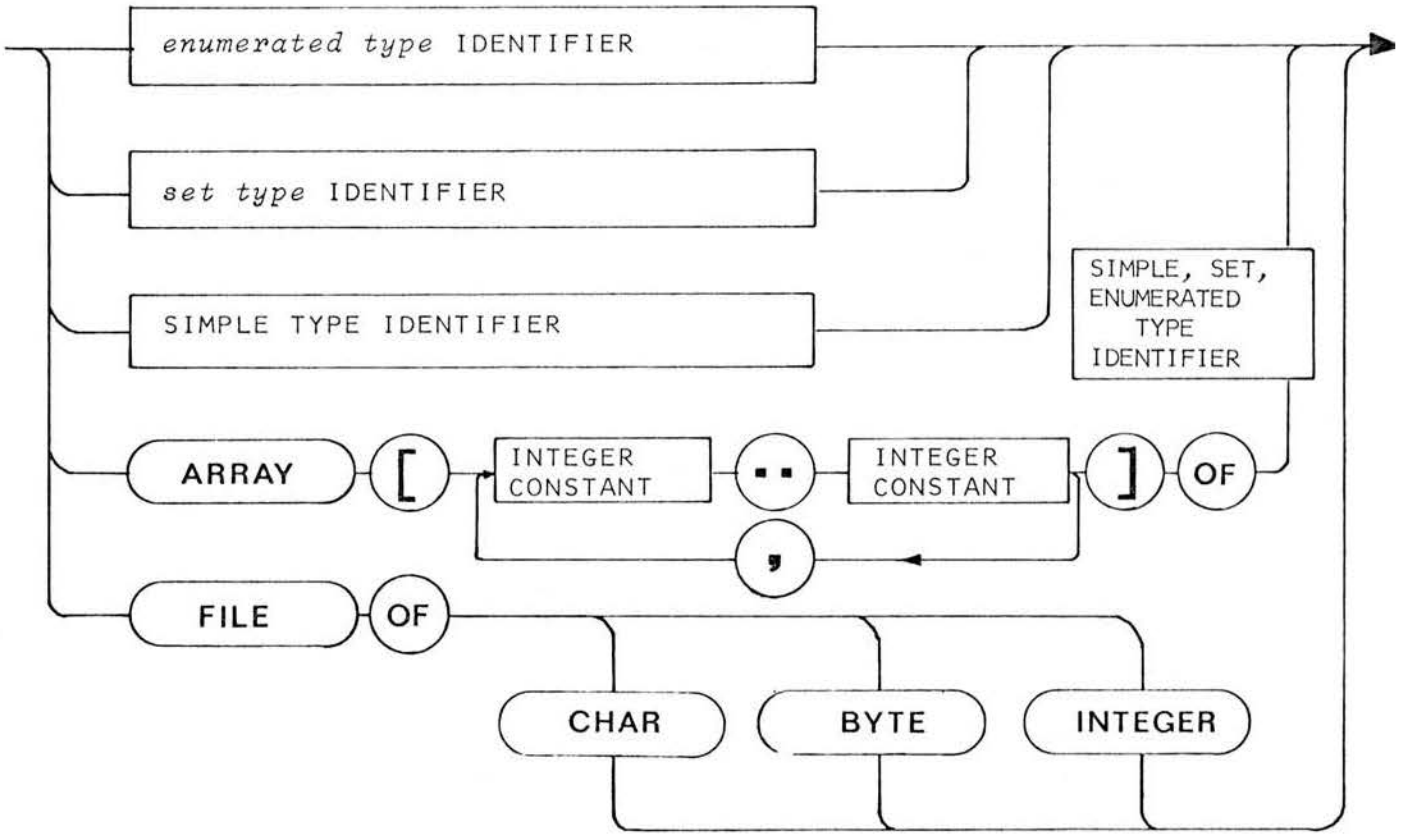


TERM

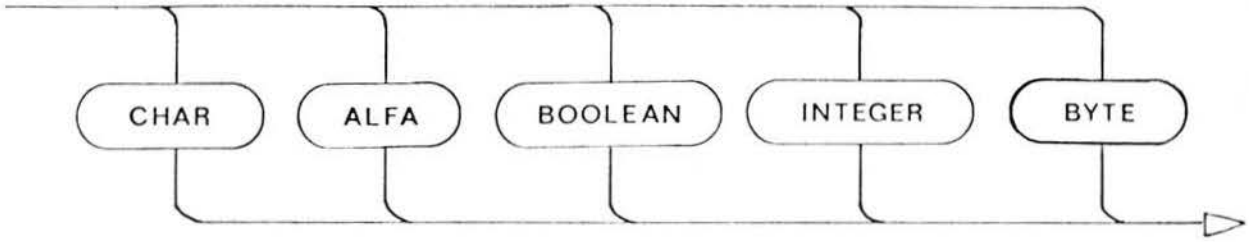




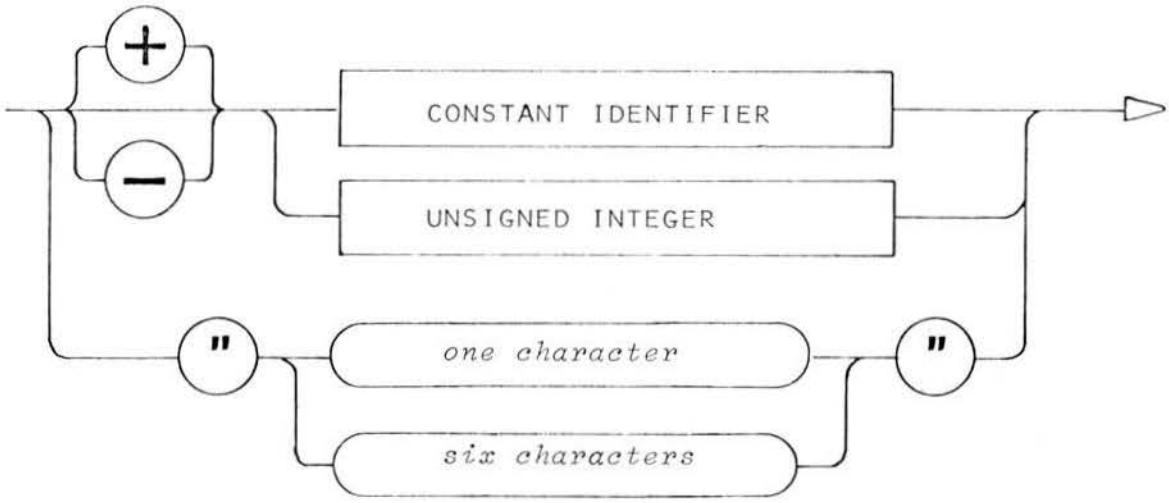
TYPE IDENTIFIER



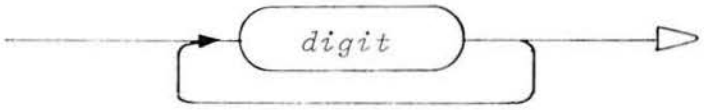
SIMPLE TYPE IDENTIFIER



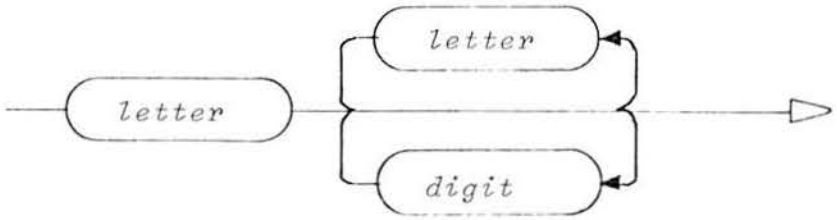
CONSTANT



UNSIGNED INTEGER



IDENTIFIER



LIST OF ERRORS DETECTED BY THE P-6800 PASCAL COMPILER

```
1     ERROR IN PROGRAM HEADER
2     ERROR IN CONSTANT DECLARATION
3     ERROR IN TYPE     DECLARATION
4     ERROR IN VARIABLE DECLARATION
5     ERROR IN PROCEDURE/FUNCTION DECLARATION

8     END OF FILE REACHED - PROGRAM INCOMPLETE

10    BEGIN EXPECTED
11    END   EXPECTED
12    THEN  EXPECTED
13    OF    EXPECTED
14    DO    EXPECTED
15    TO OR DOWNTO EXPECTED
16    UNTIL EXPECTED

20    EXPECTED EQUALS ( = )
21    EXPECTED BECOMES(:= )
22    EXPECTED COLON  ( : )
23    EXPECTED COMMA  ( , )
24    EXPECTED PERIOD ( . )
25    EXPECTED SEMICOLON ( ; )
26    EXPECTED LEFT PARENTHESIS ( ( )
27    EXPECTED RIGHT PARENTHESIS( ) )
28    EXPECTED LEFT BRACKET     ( [ )
29    EXPECTED RIGHT BRACKET    ( ] )

30    IDENTIFIER NOT DECLARED
31    IDENTIFIER ALREADY DECLARED
32    WRONG NO OF SUBSCRIPTS
33    WRONG NO OF PARAMETERS
34    STANDARD FUNCTION/PROCEDURE ERROR

40    CHARACTER/SYMBOL NOT RECOGNISED
41    CHARACTER STRING NEITHER ALFA NOR CHAR
42    ERROR IN FACTOR
43    ERROR IN TERM
44    ERROR IN SIMPLE EXPRESSION
45    ERROR IN EXPRESSION
46    ERROR IN STATEMENT
47    ERROR IN READ STATEMENT - SHOULD BE VARIABLE

50    TYPE CONFLICT IN TERM
51    TYPE CONFLICT IN SIMPLE EXPRESSION
52    TYPE CONFLICT INVOLVING SET EXPRESSION
53    TYPE CONFLICT IN EXPRESSION
54    EXPRESSION/OPERAND TYPE CONFLICT
55    ARRAY SUBSCRIPT MUST BE OF TYPE INTEGER
56    INTEGER NUMBER EXPECTED
```

60 READLN/WRITELN/EOLN ONLY ALLOWED ON FILE OF CHAR
61 FIELD WIDTH CONTROL MUST BE CONSTANT
62 IDENTIFIER IS NOT A FILE
63 TYPE CONFLICT BETWEEN FILE AND EXPRESSION OR VARIABLE

70 LOWER BOUND EXCEEDS UPPER BOUND
71 BOUND IDENTIFIER MUST BE CONSTANT
72 CASE EXPRESSION ERROR
73 CASE LABEL NOT A CONSTANT
74 FOR LOOP ERROR
75 SUBRANGE ERROR

THE FOLLOWING ARE DUE TO COMPILER RESTRICTIONS

90 TOO MANY IDENTIFIERS
91 NO OF LOWER/UPPER BOUND PAIRS EXCEEDS 30
92 TOO MANY CONDITIONAL EXPRESSIONS
93 INTEGER CONSTANT VALUE EXCEEDS 32767
94 LINE LENGTH EXCEEDS 80 CHARACTERS
95 NO OF ARRAY SUBSCRIPTS EXCEEDS 7
96 ONLY VALUE PARAMETERS PERMITTED
97 FOR LOOP VARIABLE MUST BE AN UNSUBSCRIPTED INTEGER
98 SCALAR (ENUMERATED TYPE) ORDINAL EXCEEDS 64
99 FEATURE NOT IMPLEMENTED

Run-Time System Errors Detected during Initialisation

```
DISK ERROR ON CODE FILE ***
CODE FILE SPECIFICATION ERROR ***
CANNOT OPEN CODE FILE ***          (probably on a different drive)
HIGHEST OP CODE NOT SUPPORTED ***  (code file not a P-code file)
```

Run-Time System Errors Detected during Program Execution

```
CASE VARIABLE ERROR ***
PAGE TABLES CORRUPTED ***          (memory overwritten)
INTEGER OVERFLOW ***
DIVISION BY ZERO ***
FILE ORDINAL EXCEEDS MAXIMUM ***   (memory overwritten/owncode
                                    out of control)
COULD NOT CLOSE FILE ***
COULD NOT OPEN FOR READ ***
COULD NOT OPEN FOR WRITE ***
READ ERROR ***
WRITE ERROR ***
STACK OVERFLOW ***                 (see Section 5.1.4)
ARRAY BOUNDS OR SUB-RANGE ERROR ***
INTEGER TO BYTE OVERFLOW ***
ASCII RANGE EXCEEDED ***
SCALAR RANGE ERROR ***
INPUT FORMAT ERROR ***
NON-RECOVERABLE ***                (if disk file)
RE-ENTER INTEGER :                 (if terminal)
```

All Error Messages Followed by the Message

```
ERROR OCCURRED AT PC= nnnn
```

SPECIMEN P-6800 PASCAL PROGRAMS

1. Prime number generation and tabulation.

2. Character processing.

3. Set operations.

4. File usage.

5. Binary tree creation and traversal.

These and other demonstration programs are included on the mini floppy disc on which the compiler and run-time system are provided.

```
  0 PROGRAM PRIMES ;
  0
  0    (* A PROGRAM TO COMPUTE AND TABULATE
  0       THE FIRST N PRIME NUMBERS .        *)
  0
  0 CONST
  0    NPRIMES = 400 ;
  0    SQRTNP  =  20 ; (* SQUARE ROOT OF NO OF PRIMES *)
  0
  0 VAR
  0    PRIME : BOOLEAN ;
  0    I,J,K,LIM,X,SQUARE : INTEGER ;
  0    P : ARRAY [ 1..NPRIMES ] OF INTEGER ;
  0    V : ARRAY [ 1..SQRTNP  ] OF INTEGER ;
  0
  0 FUNCTION MODFN ( NUMBER,MODULO : INTEGER ) : INTEGER ;
  4    BEGIN
  4       MODFN:=NUMBER - (NUMBER DIV MODULO) * MODULO ;
 36    END;
 40
 40 BEGIN
 44    P[1]:=2; X:=1; LIM:=1; SQUARE:=4;
 84    FOR I:= 2 TO NPRIMES DO
 96    BEGIN
104       REPEAT
104          X:=X+2;
116          IF SQUARE <= X THEN
128          BEGIN
128             V[LIM]:=SQUARE;
144             LIM:=LIM+1;
156             SQUARE:=P[LIM]*P[LIM]
180          END;
188          K:=2 ; PRIME:=TRUE;
204          WHILE PRIME AND ( K < LIM ) DO
224          BEGIN
224             IF V[K] < X THEN V[K]:=V[K]+P[K];
284             PRIME:=( X <> V[K] ) ;
308             K:=K+1
316          END
320       UNTIL PRIME;
332       P[I]:=X;
348       IF MODFN(I,10) = 0 THEN
372       BEGIN
372          WRITELN;
376          FOR J:= I-9 TO I DO WRITE(P[J]);
436       END;
436    END;
456 END.
```

```
  0 PROGRAM CHARS ;
  0                  (* PROGRAM TO DEMONSTRATE PROCESSING
  0                     OF SINGLE CHARACTERS BY MEANS
  0                     OF SUBRANGE AND CASE STATEMENTS.
  0                     SUBRANGE OF BYTE ALSO INCLUDED *)
  0 VAR
  0    CH   : CHAR ;
  0    BITE : INTEGER ;
  0
  0 PROCEDURE MESSAGE ;
  4 BEGIN
  4    WRITELN;
  8    WRITELN("TYPE ANY CHARACTER - SPACE TO EXIT");
 52 END;
 56
 56 BEGIN
 60
 60    MESSAGE;
 64
 64    REPEAT
 64       WRITE("TYPE A CHARACTER ");
 88       READ(CH); WRITE(" : ");
108       IF CH IN [ "A"..."Z" ] THEN
120                WRITELN(CH," IS A LETTER") ELSE
156       IF CH IN [ "0"..."9" ] THEN
168                WRITELN(CH," IS A DIGIT ") ELSE
204       WRITELN(CH," IS A SPECIAL CHARACTER");
248    UNTIL CH = " " ;
260
260    MESSAGE;
264
264    REPEAT
264       WRITE("TYPE A CHARACTER ");
288       READ(CH); WRITE(" ");
304       IF CH IN [ "A"..."E" ] THEN
316       CASE CH OF
328       "A" : WRITELN("ALPHA");
360       "B" : WRITELN("BRAVO");
392       "C" : WRITELN("CHARLIE");
428       "D" : WRITELN("DELTA");
460       "E" : WRITELN("ECHO")
480       END
500       ELSE WRITELN("ETCETERA");
524    UNTIL CH =" ";
536
536    WRITELN; WRITELN("ENTER 0 TO EXIT");
568    REPEAT
568       REPEAT
568          WRITE("ENTER AN INTEGER 0..255 ");
600          READ(BITE);
608       UNTIL (( BITE>=0 ) AND ( BITE < 256 ));
636       IF BITE <> 0 THEN
648       BEGIN
648          IF BITE IN [ 1.. 9] THEN WRITELN("A SINGLE DIGIT") ELSE
692          IF BITE IN [10..99] THEN WRITELN("TWO DIGITS")
724                                  ELSE WRITELN("THREE DIGITS");
756       END;
756    UNTIL BITE = 0 ;
768 END.
```

```
  0 TYPE
  0    PEOPLE = ( ANN,BILL,CHARLES,DAVID,EVE,FRED );
  0    GROUP  = SET OF PEOPLE ;
  0 VAR
  0    BLUEEYES,BROWNEYES,
  0    MALE,FEMALE,BOYS,MEN,GIRL,LADY : GROUP ;
  0
  0 PROCEDURE SHOW ( NAMETEXT : ALFA ; PERSONS : GROUP ) ;
  4    VAR
  4       NAME : PEOPLE ; DONE : BOOLEAN ;
  4    BEGIN
  8       WRITE(NAMETEXT," INCLUDES ");
 32       NAME:=ANN;
 40       DONE:=FALSE;
 48       REPEAT
 48          IF NAME IN PERSONS THEN
 60          CASE NAME OF
 72       ANN     : WRITE("ANN ");
100       BILL    : WRITE("BILL ");
128       CHARLES: WRITE("CHARLES ");
160       DAVID   : WRITE("DAVID ");
188       EVE     : WRITE("EVE ");
216       FRED    : WRITE("FRED  ")
236          END;
252          IF NAME <> FRED THEN NAME:=SUCC(NAME)
268                          ELSE DONE:=TRUE;
288       UNTIL DONE ;
296       WRITELN;
300    END;
304
304 BEGIN
308
308    (* EXAMPLES OF SET CREATION/ASSIGNMENTS,
308                          UNION AND DIFFERENCE *)
308
308    MALE  := [ BILL,CHARLES,DAVID,FRED ] ;
328    FEMALE:= [ ANN,EVE ] ;
340    BOYS  := [ BILL,CHARLES ] ;
352    MEN   := MALE - BOYS ;
368    GIRL  := [ ANN ] ;
376    LADY  := [ EVE ] ;
384    BLUEEYES := [ ANN,CHARLES,FRED ] ;
400    BROWNEYES:= MALE + FEMALE - BLUEEYES ;
424
424    WRITELN;
428
428    SHOW ( "MALE  ",MALE  );
448    SHOW ( "FEMALE",FEMALE);
468    SHOW ( "BOYS  ",BOYS  );
488    SHOW ( "MEN   ",MEN   );
508    SHOW ( "GIRL  ",GIRL  );
528    SHOW ( "LADY  ",LADY  );
548
548    WRITELN;  (* SET INTERSECTION EXAMPLES *)
552
552    SHOW("M.BLUE", MALE * BLUEEYES);
580    SHOW("M.BROW", MALE * BROWNEYES);
608    SHOW("F.BLUE", FEMALE * BLUEEYES);
636    SHOW("F.BROW", FEMALE * BROWNEYES);
664
664    IF GIRL = FEMALE * BLUEEYES (* SET EQUALITY TEST *)
676               THEN WRITE(" IT'S ANN");
700 END.
```

```
  0 PROGRAM FILES ( CHAFILE,BYTFILE,INTFILE ) ;
  0 VAR                        (* COMMAND IS RUN FILEDEMO A B C *)
  0    CHAFILE : FILE OF CHAR ;
  0    BYTFILE : FILE OF BYTE ;
  0    INTFILE : FILE OF INTEGER ;
  0    J       : INTEGER ; CH : CHAR ; B : BYTE ;
  0 BEGIN
  4
  4    REWRITE(CHAFILE); (* OPEN FOR WRITE *)
  8
  8    FOR J:= ORD("A") TO ORD("Z") DO WRITE(CHAFILE,CHR(J));
 68
 68    RESET(CHAFILE); (* OPEN FOR READ *)
 72
 72    REPEAT
 72       READ(CHAFILE,CH);
 80       IF NOT EOF(CHAFILE) THEN WRITE(CH);
 96    UNTIL EOF(CHAFILE);
100
100    WRITELN;
104    REWRITE(BYTFILE); REWRITE(INTFILE);
112
112    (* VARIABLES DECLARED TO BE OF TYPE BYTE
112       MAY BE WRITTEN ON TO INTEGER FILES,
112       AND INTEGERS MAY BE WRITTEN ON TO BYTE FILES,
112       PROVIDED THAT THE INTEGERS ARE IN THE RANGE 0..255
112       OR A RUN-TIME ERROR WILL RESULT.
112       BYTE TO INTEGER OR INTEGER TO BYTE CONVERSION
112       IS DONE AUTOMATICALLY BY THE COMPILER.
112       THE SAME APPLIES TO READ OPERATIONS. *)
112
112    FOR J:=0 TO 255 DO
124    BEGIN
132       WRITE(BYTFILE,J); WRITE(INTFILE,J);
152    END;
172
172    WRITELN;
176    RESET(BYTFILE); RESET(INTFILE);
184
184    REPEAT
184       WRITELN;
188       READ(BYTFILE,B); READ(INTFILE,J); WRITE(B,J);
224    UNTIL J=255 ; (* TEST FOR A CONDITION
236                     OTHER THAN END OF FILE
236                     ON BINARY ( BYTE OR INTEGER ) FILES
236                     SINCE FLEX ONLY SETS EOF TO TRUE WHEN
236                     THE FILE HAS BEEN READ TO THE END
236                     OF ITS LAST SECTOR *)
236    WRITELN;
240    RESET(BYTFILE); RESET(INTFILE);
248
248    REPEAT
248       WRITELN;
252       READ(BYTFILE,J); READ(INTFILE,B); WRITE(B,J);
296    UNTIL J=255 ;
308                        (* RE-OPEN THESE FILES FOR WRITE,
308                           IN ORDER TO RETURN THE SECTORS USED
308                           OTHERWISE THEY WILL BE CATALOG'D *)
308    REWRITE(CHAFILE);
312    REWRITE(BYTFILE);
316    REWRITE(INTFILE);
320 END.
```

```
  0 PROGRAM TRAVERSE ( DATA ) ;
  0                  (* A PROGRAM TO BUILD A BINARY TREE
  0                     AND THEN TRAVERSE IT
  0                     IN PRE-ORDER AND POSTORDER *)
  0 (* ADAPTED FROM A PROGRAM GIVEN IN
  0    "PASCAL USER MANUAL : REPORT" BY JENSEN + WIRTH *)
  0 CONST
  0       NUL=0; ROOT=0; LEFT=1; RIGHT=2;
  0
  0 VAR
  0       CH       : CHAR ;
  0       DATA     : FILE OF CHAR ;
  0       LASTNODE: INTEGER ;
  0       INFO     : ARRAY [ 1..255 ] OF CHAR ;
  0       POINTER : ARRAY [ 0..255,LEFT..RIGHT ]  OF INTEGER ;
  0
  0 PROCEDURE ENTER ( NODE,SIDE : INTEGER ) ;
  4    VAR
  4       NEWNODE : INTEGER ;
  4    BEGIN
  8       READ(DATA,CH); WRITE(CH);
 24       IF CH="." THEN POINTER[NODE,SIDE]:=NUL ELSE
 60       BEGIN
 60          NEWNODE:=LASTNODE+1;
 72          POINTER[NODE,SIDE]:=NEWNODE;
 92          INFO[NEWNODE]:=CH;
108          LASTNODE:=NEWNODE;
116          ENTER(NEWNODE,LEFT);
132          ENTER(NEWNODE,RIGHT)
144       END
148    END;
152
152 PROCEDURE PREORDER ( NODE : INTEGER ) ;
152    BEGIN
152       IF NODE <> NUL THEN
164       BEGIN
164          WRITE(INFO[NODE]);
180          PREORDER(POINTER[NODE,LEFT]);
204          PREORDER(POINTER[NODE,RIGHT])
224       END
228    END;
232
232 PROCEDURE POSTORDER ( NODE : INTEGER ) ;
232    BEGIN
232       IF NODE <> NUL THEN
244       BEGIN
244          POSTORDER(POINTER[NODE,LEFT]);
268          POSTORDER(POINTER[NODE,RIGHT]);
292          WRITE(INFO[NODE])
308       END
308    END;
312
312 BEGIN
316       LASTNODE:=ROOT;
324       REWRITE(DATA);
328       WRITE(DATA,"ABC..DE..FG...HI..JKL..M..N..");
364       RESET(DATA);
368       WRITELN; WRITE("INPUT      "); ENTER(ROOT,RIGHT);
404       WRITELN; WRITE("PRE-ORDER "); PREORDER(POINTER[ROOT,RIGHT]);
448       WRITELN; WRITE("POSTORDER "); POSTORDE(POINTER[ROOT,RIGHT]);
492       REWRITE(DATA);
496 END.
```