ProFortran

Prospero Software


ProFortran
Prospero Software

Prospero Software

LANGUAGES FOR MICROCOMPUTER PROFESSIONALS

LICENCE AGREEMENT

All material on the enclosed disc(s) and in any accompanying manual Copyright © 1982, 1983, 1984, 1985 Prospero Software ("the Licensor"). The Software recorded on the enclosed disc(s) is proprietary and its use is offered to you subject to the terms set out below.

PLEASE READ THESE TERMS BEFORE BREAKING THE SEALING STRIP

If you cannot accept the terms, then return the unopened packet to the dealer from whom you bought it: he will refund the price which you paid. If you accept the terms, then (a) break the seal on the disc: your doing so will constitute acceptance of the offer on the terms set out below: the contract will be made at the time of opening: the Licensor waives the requirement that acceptance must be communicated to the offeror.
(b) complete the registration card and send it to the Licensor: only if you do so will you receive notification of updates and qualify for technical support.

Terms Governing the Use of the Software

1. These terms govern the following material ("the Proprietary Material") (a) The software encoded onto the disc(s) ("the Software") and (b) Updates to the Software issued by the Licensor and taken by the User from time to time during the currency of these terms ("the Updates"). If the User completes the registration card and sends it to the Licensor, the Licensor will from time to time send notification of Updates to the User at the User's registered address and the user will have the opportunity of taking those Updates on payment of the relevant handling charges.
2. On breaking the seal and in consideration of that part of the payment made by the User to the dealer which is made to the dealer as agent for the Licensor, the Licensor grants to the User a non-exclusive, non-transferable licence ("the Licence") to use the Software and the Updates on the single microcomputer which is specified on the registration card.
3. The term of this Licence shall commence on the date of opening of this packet and shall be indefinite but the Licensor shall have the right to terminate the Licence on not less than fourteen days' notice in the event of (a) breach by the User of these terms or of the terms of any other agreement between the Licensor and the User (b) the User (being a corporation) entering into liquidation whether compulsory or voluntary except for the purpose of amalgamation or reconstruction or (being an individual) committing any act of bankruptcy or making any assignment for the benefit of creditors. Immediately upon termination the User shall return all Proprietary Material to the Licensor.
4. (a) All intellectual property and copyright in the Proprietary Material are and will remain the property of the Licensor. (b) The User may make such copies of the Software as are reasonably required for the proper use of the Software and for security or archival purposes provided that each such copy carries a full and sufficient eye-readable copyright notice in acknowledgement of the Licensor's intellectual property rights. Such copies will

14-8-89
LATEST VERSION 1.17
(MINOR FIXES) PRINT
UPDATE FOR £10 IF
CARDS + FLPROM RETURNED

Prospero Software

LANGUAGES FOR MICROCOMPUTER PROFESSIONALS

REGISTRATION CARD

BLOCK CAPITALS PLEASE

PRODUCT NAME Pro Fortran-77 Version mmq 1.1 for QL

SERIAL NUMBER 1326

PURCHASER'S NAME

POSITION

COMPANY

ADDRESS

TELEPHONE

SIGNATURE

COMPUTER SINCLAIR QL

OPERATING SYSTEM QDOS

NAME OF DEALER

ADDRESS OF DEALER

DATE WHEN PACKET OPENED 1st October 1986

IMPORTANT NOTE

To qualify for technical support and new releases at concessionary prices, you *must* register your purchase of this software by returning this card to the address below.

Prospero Software Limited
190 Castelnau
London SW13 9DH
England

THANK YOU

P R O F O R T R A N - 7 7

U S E R M A N U A L

Version mmq 1.1
for Sinclair QL
with QDOS

March 1986

Copyright (C) 1985 Prospero Software

190 Castelnau
London SW13 9DH England

P R O F O R T R A N - 7 7

U S E R M A N U A L

Version mmq 1.1
for Sinclair QL
with QDOS

March 1986

Copyright (C) 1985 Prospero Software

190 Castelnau
London SW13 9DH England

PART I - PRO FORTRAN-77 OVERVIEW

1 PROGRAM UNITS

1.1 Compilation input

The input for a compilation is a file containing one or more Fortran program units (main program, subroutine, function, or block data).

If desired, commonly used sequences of statements may be kept on separate files, and incorporated in the source at compile time by use of the statement

```
INCLUDE 'filename'
```

where filename is the name of the file containing the statements to be included at that place in the source. (These statements may not, themselves, contain any INCLUDE statements.)

The object file contains a relocatable module combining the source program units. (There should not therefore be more than one main program in a compilation input.) Listing and map options are provided - see Part III.

Compilation is terminated by end-of-file on the input.

1.1.1 Main program unit

A main program unit may start with a PROGRAM statement in the form

```
PROGRAM name
```

If there is no PROGRAM statement the name .MAIN is supplied. The name of the first unit in the compilation input becomes the object module name.

1.1.2 SUBROUTINE

A subroutine subprogram must start with a SUBROUTINE statement in the form

```
SUBROUTINE name [(dummy-argument-list)]
```

where "name" becomes the entry name in the object file, and also the module name if this is the first unit in the compilation input.

1.1.3 FUNCTION

A function subprogram must start with a FUNCTION statement in the form

[type] FUNCTION name (dummy-argument-list)

where "name" becomes the entry name in the object file, and may also become the module name as above.

1.1.4 BLOCK DATA

A block data subprogram must start with a BLOCK DATA statement, which may optionally specify a name for the unit. If no name is given, the default name .BDATA is supplied.

1.2 Matching of actual and dummy arguments

In the call of a subroutine or function, each actual argument must match the corresponding dummy argument in length as well as type (for instance INTEGER*2 does not match INTEGER). When the actual argument is an expression, rather than a variable or array element, it is evaluated as an INTEGER.

2 DATA TYPES

2.1 Standard types

The standard data types are implemented as follows.

INTEGER	A 4-byte integer in the range -2147483647 to +2147483647.
REAL	A 4-byte floating-point value in a format corresponding to the proposed IEEE standard. The 32 bits are allocated as follows (from most to least significant): 1-bit sign 8-bit binary exponent biassed by 127 23-bit mantissa with an implied 1 in the most significant (24th) position. Approximate decimal equivalents: 7-digit precision range E-38 to E+38
DOUBLE PRECISION	An 8-byte floating-point value in the IEEE format: 1-bit sign 11-bit binary exponent biassed by 1023 52-bit mantissa with an implied 1 in the most significant (53rd) position. Approximate decimal equivalents: 16-digit precision range E-308 to E+308
COMPLEX	A pair of values of type REAL, real part in low memory, imaginary in high.
LOGICAL	A 4-byte quantity taking the values 0, 1 for .FALSE., .TRUE. respectively.
CHARACTER	One byte per element, in ascending address order.

(Thus a "numeric storage unit" is 4 bytes, and a "character storage unit" is 1 byte.)

2.2 Additional types

The following shorter types are provided to allow economy of data space and for compatibility with other implementations. In machines based on 8086 or related processors, the shorter types will also tend to result in more compact object code also. The 68000 architecture on the other hand gives little advantage in code size from the use of shorter types, and indeed it may even be adversely affected.

INTEGER*2	A 2-byte value in the range -32768 to 32767.
INTEGER*1	A 1-byte value in the range -128 to 127.
LOGICAL*2	A 2-byte equivalent to LOGICAL.
LOGICAL*1	A 1-byte equivalent to LOGICAL.

INTEGER*4 and LOGICAL*4 are recognised, and treated exactly as INTEGER and LOGICAL. REAL*8 is recognised and treated as DOUBLE PRECISION. COMPLEX*8 is recognised and treated as COMPLEX.

The shorter types may be used in place of the standard equivalents in any circumstances where the reduced range is still sufficient. Any necessary extension is supplied automatically. However, when supplying variables or array elements as actual arguments in a subroutine or function call, the user must ensure that length as well as type matches the dummy argument. (If the actual argument is a constant or an expression, it will always be automatically passed as a 4-byte integer.)

In the 68000, all 2- and 4-byte variables must be placed at even addresses. Declaring INTEGER*1 or LOGICAL*1 may require that a "slack" byte is introduced before the next variable. Normally this is done quite automatically by the compiler, but it is possible to write EQUIVALENCE statements which cannot be correctly processed, and in such cases the compiler will signal an error.

2.3 Additional constants

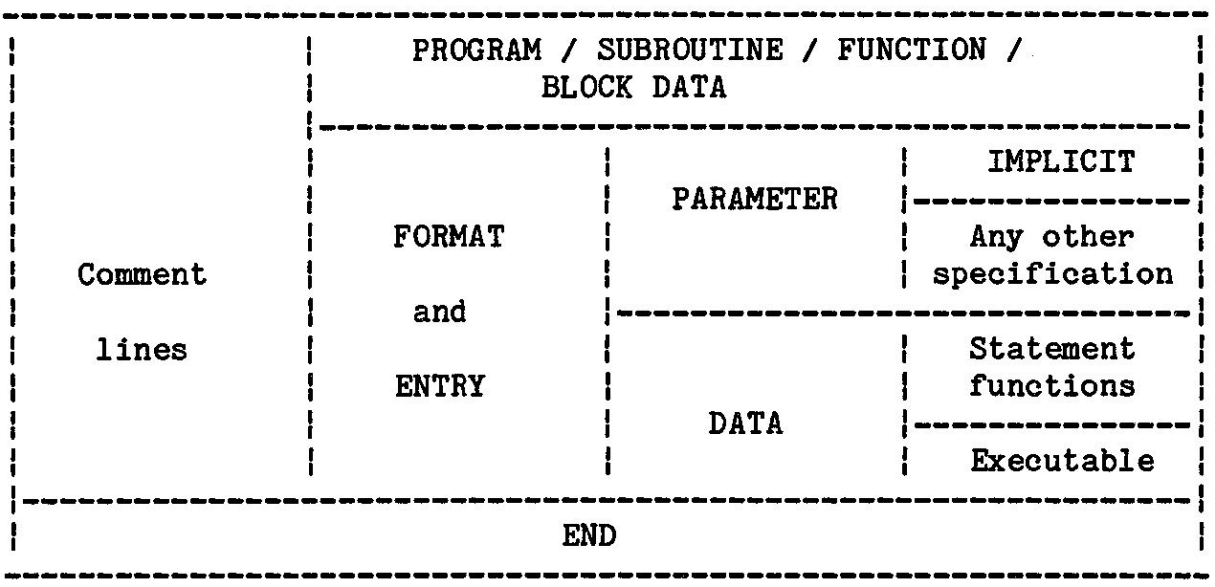
Hexadecimal constants may be written anywhere an integer constant is allowed, for example:

```
$3AF
```

3 STATEMENTS

3.1 Standard statements

The statements of Fortran-77 are all provided. Furthermore, the Fortran rules about statement ordering within program units (as illustrated in the diagram below) are fully implemented.



3.1.1 Non-executable statements

Type statements (INTEGER etc.)

DIMENSION statement (up to 7 dimensions)

COMMON statement (blank and named common)

EQUIVALENCE statement

IMPLICIT statement

PARAMETER statement

EXTERNAL statement

INTRINSIC statement

SAVE statement

DATA statement

FORMAT statement

ENTRY statement

4 INPUT AND OUTPUT

4.1 Files and records

There are four kinds of file, distinguished by the records they contain. A record may be

formatted or unformatted
variable-length or fixed-length.

These distinctions are made in the OPEN statement. Formatted and unformatted records cannot be mixed in the same file, nor can variable-length and fixed-length.

Unformatted records are normally confined for practical reasons to filestore files, as are fixed-length formatted records. Variable length formatted records may be read from or written to filestore files or devices such as console or printer. However the Fortran run-time routines are written to avoid device-dependence, and will attempt to do whatever they are asked.

Output of variable-length formatted records always includes interpretation of the first character of a record as carriage control, to preserve device-independence.

Formatted records are limited in size to a maximum of 200 bytes. Fixed-length unformatted records are limited in size to 32767 bytes. Variable-length unformatted records may be arbitrarily long.

4.2 Unit numbers

A unit number is associated with a file, and is the means by which a program refers to that file. Unit numbers can be in the range 0 to 255.

The standard input and output (unit *) are preconnected when execution starts. Any other unit must be connected to a file by means of an OPEN statement before it can be used. At most 15 files (in addition to standard input and output) may be open at one time.

4.3 Sequential access

Sequential processing is supported only for files of variable-length records, i.e. no RECL= parameter must be given in the OPEN statement.

4.4 Direct (random) access

A file of fixed-length records (RECL specified in the OPEN statement) must be specified as ACCESS='DIRECT'. Individual records are addressed using the REC= option in READ or WRITE, and the file may be updated using a mixture of READ and WRITE operations to the same unit.

When a direct access file is being created or extended, the records may be written in any order. Any "holes" thereby formed are automatically filled in with null records by the system. The user is responsible, however, for not reading such records before they have been properly written.

4.5 END, ERR and IOSTAT options

Generally, if an error occurs during input or output, or end-of-file is detected during input, a message is issued and the program is terminated. The END, ERR and IOSTAT options allow the user to modify this behaviour.

The ERR= and IOSTAT= options can be used in all input/output statements, except PRINT and the simple form of READ. The END= option can only be used in a READ statement for sequential input.

In the set of values which can be returned by IOSTAT, zero indicates successful completion, -1 indicates end of file detected during READ, and a positive (non-zero) value indicates an error condition. (For error status values, see Appendix C.) If neither END nor ERR is also specified, program execution continues at the next statement, and it is the user's responsibility to detect and handle abnormal conditions.

In the event of end of file or error during a READ operation, the values of any variables in the input list are indeterminate.

4.6 BACKSPACE, ENDFILE and REWIND

The syntax for these statements allows the unit number to be quoted either with or without parentheses, for example

```
REWIND 6
REWIND (J)
```

On a sequential input file, the BACKSPACE operation causes the file to be repositioned so that the next record obtained will be the one which has just been read. On a sequential output file, BACKSPACE causes the file to be repositioned so that the next record written replaces the last record written. (In both cases, if the file is already at its initial position, BACKSPACE is equivalent to CONTINUE. On output, a backspace may leave undefined information in the file.)

4.7 OPEN statement

An OPEN statement must be executed to establish a connection between a unit number and a file before the unit is used for input or output. The main options in OPEN are described below, and more detail will be found in Part II.

1. The FILE= option allows the unit to be connected to a filestore file, a device, or in some cases a window. If no name is specified, a scratch filestore file is connected; all such files are deleted at the end of the execution.

2. The RECL= option specifies a record length in bytes and defines fixed-length records of this length. (A "numeric storage unit" is 4 bytes, a "character storage unit" is 1 byte.) The file should be a (named or unnamed) filestore file. If it already exists the record length must be consistent with the RECL value.

3. The ACCESS= clause must be present if access is DIRECT.

4. The FORM= clause must be present if the file is formatted and access is direct, or unformatted and access is sequential.

5. The ERR= or IOSTAT= options may be used to allow errors in the file open operation to be handled within the program. Note however that a failure to find a named file will not be detected at the time of the OPEN statement, but when the first READ or WRITE is executed.

5 IMPLEMENTATION NOTES

The purpose of this section is to bring together a number of points relating to the implementations of Pro Fortran-77 on different hardware and/or operating system environments.

5.1 68000 implementations

The instruction set of the Motorola MC68000 and related processors (68008, 68010, 68020) uses in many situations a signed 16-bit word to contain address displacements. The Pro Fortran-77 implementation for 68000-based machines generally avoids any limitation of program components arising from this format, but there are inevitable consequences for efficiency, and the developer of large programs should be aware of this underlying fact. Thus for example there is no restriction on the sizes of Fortran COMMON blocks, but if a block exceeds 32K bytes some of the data becomes more difficult to access, and if a program is to be "tuned" for the machine it may be best to rearrange or split such a block.

One specific limitation is made: it is that the generated code for one program unit (main program, subroutine or function) cannot exceed 32K bytes. As many program units as desired may be linked to form a complete program.

No hard restrictions are placed on the sizes of arrays, on COMMON blocks, or on local data storage.

5.2 Resident Library

This implementation introduces the concept of a "resident library" which contains commonly-used routines (concerned for example with input/output). By removing these routines from the conventional library, link times and object program sizes are reduced. In a multi-tasking environment, the resident library ("PRL") is shared between concurrent tasks. There is also a facility for one object program to execute another, and again the resident library is shared between parent and child. The PRL must be installed before executing the compiler or any Fortran object program; installation is described in Part III. Once installed it remains available, and each execution of the compiler or an object program establishes a connection with it as part of its initialisation process.

(It should perhaps be made clear that the PRL is a set of slave routines, and is not in any sense an interpreter. Each object program is compiled, and linked selectively with the non-resident library. The resident routines are simply a selection from the complete library of those which are likely to be needed by a majority of programs.)

5.3 QL and QDOS

On the QL, two versions of the resident library (PRL) are supplied. A ROM version is needed to run the compiler. This will also run Fortran object programs. In addition, a "software" PRL is supplied, to enable Fortran object programs to be run on other machines.

Fortran object programs may be run under the QL Toolkit ("EX"), in which case the standard input and output, which become unit * in the Fortran program, can be specified. If the Toolkit is not used ("EXEC") there is an opportunity to specify them before the main program is entered. In either case, the default is to keyboard and a standard window.

In each object program (and in the compiler) there is a field which can be used to define a default device for the program, and a utility is provided to modify this field in the linked binary file. The default device in an object program determines where anonymous files (that is, having no name given in the OPEN statement) are placed. In the compiler, it defines the device which will be searched for a file with further configuration details. More information will be found in Part III.

A number of routines are provided as Fortran analogues of the window/graphics output routines available in SuperBASIC. These are described in Part II, sections 8.2.13 thru 8.2.17.

PART II - PRO FORTRAN-77 LANGUAGE DEFINITION

1 LEXICAL ASPECTS

Considered from the aspect of its representation on the printed page, rather than with regard to its syntax or meaning, a Fortran program unit can be viewed as being constructed of characters grouped into lines and statements.

(The notation used, throughout this manual, for defining the Fortran syntax is described at the start of Appendix A.)

1.1 Characters

Except within comments and character constants (see below), a program unit is written using 3 kinds of characters:

```
character = letter | digit | special-character

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
special-character = blank | tab | = | + | - | * | / |
                    ( | ) | , | . | ' | :
blank = ASCII-code-32
tab = ASCII-code-9
```

Letters may be in upper- or lower-case, and, except within character constants (see 1.4.3.6) and Hollerith descriptors (see 4.3.6), no distinction is made between the upper and lower case of a letter.

After the first six positions on a line - see 1.2 below - blanks and tabs have no meaning (again, except within character constants), and may be freely used to improve the layout of the program text. In particular, names (see 1.4.2) may contain embedded blanks, and word-symbols (see 1.4.1) need not be separated by blanks from neighbouring names.

1.2 Lines

Lines consist of up to 72 characters. The character-positions are called "columns", starting at 1 for the left-most character position. Any characters beyond column 72 are ignored.

Tab characters encountered in the source file are expanded into one or more blanks, until the next "tab stop" is reached, these being at columns 1, 9, 17 and so on (every 8 columns).

A program unit is composed of 3 kinds of lines.

1.2.1 Comment line

If column 1 of a line contains the character 'C' (or 'c') or '*', the line is a comment line; or, if columns 1 onwards of a line contain only blanks, the line is a comment line. Such a line must be immediately followed by an initial line or by another comment line, and does not affect the meaning of the program in any way.

1.2.2 Initial line

If the line is not a comment line, and if column 6 is a blank or the character 0 (zero), then the line is an initial line. It is the first line of a statement.

1.2.3 Continuation line

If the line is not a comment line, and if column 6 is any character other than blank or 0, then the line is a continuation line. Such a line may only follow an initial line or another continuation line. A statement must not have more than 19 continuation lines.

1.3 Statements

A statement consists of an initial line optionally followed by up to 19 continuation lines.

Columns 1 thru 5 of the initial line may, optionally, contain a statement-label (see 1.4.4). In determining the meaning of a statement, only columns 7 thru 72 of the initial and of any continuation lines are significant. The contents of these columns consists of a sequence of lexical "tokens".

An END statement must not have any continuation lines.

1.4 Tokens

These are of 4 kinds:

```
token = special-symbol ¦ name ¦ constant ¦ statement-label
```

1.4.1 Special symbols

The special-symbols are tokens with special fixed meanings.

```
special-symbol = = ¦ , ¦ ( ¦ ) ¦ : ¦
                 + ¦ - ¦ * ¦ / ¦ ** ¦
                 .LT. ¦ .LE. ¦ .EQ. ¦ .NE. ¦ .GE. ¦ .GT. ¦
                 .OR. ¦ .AND. ¦ .NOT. ¦ .EQV. ¦ .NEQV. ¦
                 word-symbol

 word-symbol = ACCESS ¦ ASSIGN ¦ BACKSPACE ¦ BLANK ¦ BLOCKDATA ¦
               CALL ¦ CHARACTER ¦ CLOSE ¦ COMMON ¦ COMPLEX ¦
               CONTINUE ¦ DATA ¦ DIMENSION ¦ DIRECT ¦ DO ¦
               DOUBLEPRECISION ¦ ELSE ¦ END ¦ ENDFILE ¦ ENTRY ¦
               EQUIVALENCE ¦ ERR ¦ EXIST ¦ EXTERNAL ¦ FILE ¦
               FMT ¦ FORM ¦ FORMAT ¦ FORMATTED ¦ FUNCTION ¦
               GOTO ¦ IF ¦ IMPLICIT ¦ INCLUDE ¦ INQUIRE ¦
               INTEGER ¦ INTRINSIC ¦ IOSTAT ¦ LOGICAL ¦ NAME ¦
               NAMED ¦ NEXTREC ¦ NUMBER ¦ OPEN ¦ OPENED ¦
               PARAMETER ¦ PAUSE ¦ PRINT ¦ PROGRAM ¦ READ ¦
               REAL ¦ REC ¦ RECL ¦ RETURN ¦ REWIND ¦ SAVE ¦
               SEQUENTIAL ¦ STATUS ¦ STOP ¦ SUBROUTINE ¦ TO ¦
               UNFORMATTED ¦ UNIT ¦ WRITE
```

Note that the word-symbols are not "reserved words"; that is, it is possible to use STOP, IF, etc. as names (see 1.4.2). The syntax of Fortran is such that it is always possible to deduce from the context whether a sequence of characters belongs to a word-symbol or a name.

1.4.2 Names

Names are used to denote constants, data items (variables or arrays), procedures (functions or subroutines) and common blocks. They consist of from 1 to 6 characters, starting with a letter:

```
name = letter { letter ¦ digit }
```

Because the case of a letter, and embedded blanks, are not significant, the following represent one and the same name:

```
LARRY
Larry
l a  rr Y
```

1.4.3 Constants

Constants can be classified into three kinds:

```
constant = arithmetic-constant ¦ logical-constant ¦
           character-constant
```

The first kind, in turn, consists of 4 types:

```
arithmetic-constant = integer-constant ¦ real-constant ¦
                      double-precision-constant ¦ complex-constant
```

1.4.3.1 Integer constants

```
integer-constant = [sign] unsigned-integer
sign = + ¦ -
unsigned-integer = decimal-integer ¦ hexadecimal-integer
decimal-integer = digit-string
hexadecimal-integer = $ hexdigit {hexdigit}
digit-string = digit {digit}
hexdigit = digit ¦ A ¦ B ¦ C ¦ D ¦ E ¦ F
```

The value of the unsigned-integer must not exceed 2147483647.

Examples:

```
0       -128    1000001    $FF
```

1.4.3.2 Real constants

```
real-constant = [sign] unsigned-real
unsigned-real = <basic-real [E exponent] ¦ digit-string E exponent>
basic-real = <digit-string . [digit-string] ¦ . digit-string>
exponent = [sign] digit-string
```

E means "times 10 to the power of", and may be in upper or lower case.

Examples:

```
10.0      1e-10      .314159265E1
```

1.4.3.3 Double precision constants

```
double-precision-constant = [sign] unsigned-double
unsigned-double = <basic-real ¦ digit-string> D exponent
```

D means "times 10 to the power of", and may be in upper or lower case. Double-precision constants are held to greater precision than real constants (see 8.3.2). Examples:

```
1D0      .12345678901234567 89d-99
```

1.4.3.4 Complex constants

A complex constant is written as a pair of real or integer constants, the first representing the real part, the second the imaginary part:

```
complex-constant = [sign] ( real-part , imaginary-part )
real-part = real-constant ¦ integer-constant
imaginary-part = real-constant ¦ integer-constant
```

Examples:

```
(0.5 , 1.5)        -(2.1,-9.5e-6)        (1, 0)
```

1.4.3.5 Logical constants

The values taken by quantities of type LOGICAL are true and false, and the corresponding constants are written as .TRUE. and .FALSE.:

```
logical-constant = .TRUE. ¦ .FALSE.
```

1.4.3.6 character constants

```
character-constant = ' string-element {string-element} '
string-element = string-character ¦ apostrophe-image
apostrophe-image = ''
```

The string can contain between 1 and 255 (inclusive) 8-bit characters.

Examples:

```
'INPUT'
''' is an apostrophe'
```

1.4.4 Statement labels

A statement label is a sequence of 1 to 5 decimal digits, representing a value in the range 1 to 99999:

statement-label = digit-string

As usual, blanks may be embedded. The label is uniquely determined by the value of the digit-string, so that, in particular, leading zeros are not significant.

1.5 Source file inclusion

It is sometimes useful, particularly when developing large programs, to be able to keep declarations etc. which are used in more than one program unit as separate files, and cause them to be included in the source at compile time. In Pro Fortran-77, the INCLUDE statement is provided for this purpose. The syntax of the statement is:

```
INCLUDE character-constant
```

where the character constant is the name of a source file. The extension _FOR is automatically supplied by the compiler.

Example:

```
INCLUDE 'MDV1_commdefs'
```

Note that only one level of source file inclusion is supported; that is, an INCLUDEd file may not itself contain INCLUDE statements.

2 PROGRAMS AND SUBPROGRAMS

The input to the compiler consists of one or more program-units, each of which may be either a main program or one of 3 kinds of subprogram: subroutine, function or block data. Formally:

```
compilation-input = program-unit {program-unit}
program-unit = main-program | subroutine-subprogram |
               function-subprogram | block-data-subprogram
```

In each case, an END statement terminates the program unit, and the compilation process itself terminates when an END statement is followed by end-of-file.

An executable Fortran program is composed, in source terms, of a main program together with zero or more subprograms. The latter may form part of the same source file as the main program, or be in separate source files which are compiled individually and then linked together with the main program to form the executable program. Execution commences at the first executable statement of the main program. Control passes (temporarily) to a subroutine or function subprogram only when that subprogram is the subject of a CALL or function reference, respectively. Execution terminates when a STOP statement, or the END statement of the main program, is encountered.

2.1 Main program

If a program unit does not start with a SUBROUTINE, FUNCTION or BLOCK DATA statement, it is taken to be a main program. A main program can optionally be given a name, by starting it with a PROGRAM statement.

```
main-program = [program-statement] program-body
program-statement = PROGRAM name
program-body = specifications
               definitions
               executable-part
               end-statement
specifications = { specification-statement |
                   format-statement | entry-statement }
definitions = { statement-function-definition |
                data-statement | format-statement |
                entry-statement }
executable-part = { executable-statement | data-statement |
                    format-statement | entry-statement }
end-statement = END
```

The "specifications", "definitions" and "executable-part" are each optional but, if present, must appear in that order. They are treated in sections 3, 4 and 5, respectively.

Within the "specifications" part of a program unit, IMPLICIT statements must precede all other specification statements except PARAMETER statements.

(For a pictorial representation of the constraints on statement ordering within each program unit, refer to Part I, section 3.1.)

An example of a trivial but complete main program is:

```
PRINT *, ' Hello'
END
```

2.2 SUBROUTINE subprogram

A subroutine subprogram consists of a SUBROUTINE statement, which specifies both the name by which the subroutine is CALLable from other program units and any arguments which it needs, followed by a "program-body", which is structured precisely as for a main program.

```
subroutine-subprogram = subroutine-statement program-body
subroutine-statement = SUBROUTINE subroutine-name
                                  [ ( [dummy-argument-list] ) ]
subroutine-name = name
dummy-argument-list = dummy-argument {, dummy-argument}
dummy-argument = variable-name ¦ array-name ¦ procedure-name ¦ *
variable-name = name
array-name = name
procedure-name = subroutine-name ¦ function-name
```

As an example of a complete subroutine, the following subprogram interchanges the contents of two real variables (the actual parameters could also be array elements):

```
SUBROUTINE  SWAP (X,Y)
W = X
X = Y
Y = W
RETURN
END
```

2.3 FUNCTION subprogram

A function subprogram consists of a FUNCTION statement followed by a "program-body". As for a subroutine, the first statement specifies the name of the subprogram and any arguments which it needs. It is possible to specify a type for the function, either as part of the FUNCTION statement or by a type-statement (see 3.2) or IMPLICIT statement (see 3.3) within the function's program-body.

```
function-subprogram = function-statement program-body
function-statement = [type-specifier] FUNCTION function-name
                                      ( [dummy-argument-list] )
type-specifier = < arithmetic-type ¦ logical-type ¦
                   CHARACTER [ * len ] >
function-name = name
```

In order that the function shall return a value to the calling program-unit, the function-name must appear on the left-hand-side of at least one assignment-statement within the program-body.

As an example of a complete function subprogram, the following returns as its (real) result the cube of its (real) argument:

```
FUNCTION  CUBE (X)
CUBE = X ** 3
END
```

[illegible]

2.4 ENTRY statement

A subroutine or a function subprogram may optionally contain one or more ENTRY statements.

```
entry-statement = ENTRY name [ ( [dummy-argument-list] ) ]
```

The name in the ENTRY statement becomes accessible outside the program unit in the same way as the subroutine- or function-name, enabling it to be called from other program units, with an optional actual argument list. Execution begins with the first executable statement following the ENTRY statement, which itself may appear anywhere within the program unit except inside a DO loop or a Block IF.

The number and types of the dummy arguments do not have to be the same as the number and types of the dummy arguments in the SUBROUTINE or FUNCTION statement.

If the ENTRY is in a function subprogram, the name in the ENTRY statement refers to a variable which is associated with, in the sense of sharing storage with, the function-name variable. The two do not have to be of the same type, except when they are of character-type when they must be of identical type. This variable must have been assigned a value at the time of exit from the function.

In the following example, the main program calls the subroutine two different ways, once using the SUBROUTINE name and once using the ENTRY name:

```
      PROGRAM main
      A = 1.0
      B = 2.0
      C = 3.0
      CALL PERM3 (A,B,C)
      CALL PERM2 (A,B)
      END

      SUBROUTINE PERM3 (X, Y, Z)
      T = Z
      Z = X
      GOTO 10
      ENTRY PERM2 (X, Y)
      T = X
10    X = Y
      Y = T
      RETURN
      END
```

Although not realistic, this example serves to point up, in particular, the fact that a dummy argument in an ENTRY statement and a dummy argument in a SUBROUTINE/FUNCTION statement which have the same name are one and the same variable, so far as references within that program unit are concerned.

2.5 BLOCK DATA subprogram

The purpose of a block data subprogram is to give initial values (via DATA statements) to items in named common blocks. This is the only way common items may be initialised. (Items in blank common may not be initialised at all.) The statements which can appear in a block data subprogram are a subset of those which can appear in the other 3 kinds of program unit; in particular, there is no "executable-part", and no INTRINSIC or EXTERNAL or FORMAT statements may be present.

```
block-data-subprogram = block-data-statement
                        block-data-body
block-data-statement = BLOCK DATA [ name ]
block-data-body = block-data-specifications
                  block-data-definitions
                  end-statement
block-data-specifications = {specification-statement}
block-data-definitions = {data-statement}
```

Example:

```
BLOCK DATA
COMMON /CB/ A, B, C, D
DOUBLE PRECISION D
DATA A, B, C, D / 3*0.0, 1.0D0 /
END
```

3 SPECIFICATION STATEMENTS

Specification statements are non-executable: they are concerned with specifying to the compiler the types and sizes of data items, and how storage should be allocated for them.

```
specification-statement = type-statement ¦ implicit-statement ¦
                          dimension-statement ¦ common-statement ¦
                          equivalence-statement ¦ save-statement ¦
                          external-statement ¦ intrinsic-statement ¦
                          parameter-statement
```

3.1 Data types

Named items (constants, variables, arrays and functions) can be of 6 basic types: integer, real, double-precision, complex, logical or character. Constants corresponding to each of these data types can be written, as described in section 1.

3.2 Type statements

The most explicit way to give a type to a named item is by use of a type statement:

```
type-statement = < non-character-type-statement ¦
                   CHARACTER [ * len [,] ] character-item-list >
non-character-type-statement = <arithmetic-type ¦ logical-type>
                               typed-item {, typed-item}
len = decimal-integer ¦ ( integer-constant-expression ) ¦ ( * )
integer-constant-expression = constant-expression
character-item-list = character-item {, character-item }
arithmetic-type = integer-type ¦ real-type ¦
                  double-precision-type ¦ complex-type
logical-type = LOGICAL [ * <1¦2¦4> ]
typed-item = variable-name ¦ array-name ¦ array-declarator
character-item = typed-item [ * len ]
integer-type = INTEGER [ * <1¦2¦4> ]
real-type = REAL [ *4 ]
double-precision-type = < DOUBLEPRECISION ¦ REAL*8 >
complex-type = COMPLEX [ *8 ]
```

If the typed-item is a variable- or function-name, the item itself takes on values of the stated type; if the typed-item is an array-name or an array-declarator (see 3.4), then the corresponding array elements take values of that type.

If an arithmetic- or logical-type includes a "*" qualifier, the digit after the "*" specifies the storage allocation, in bytes, for that type. INTEGER is synonymous with INTEGER*4, REAL with REAL*4, and LOGICAL with LOGICAL*4.

The following table summarises the properties of the various types:

Type	Size	Values
INTEGER	4	-2147483647 to +2147483647
INTEGER*4	4	ditto
INTEGER*2	2	-32768 to +32767
INTEGER*1	1	-128 to +127
REAL	4	approx. 7 digit accuracy, range E(+-)38
REAL*4	4	ditto
DOUBLEPRECISION	8	approx. 16 digit accuracy, range D(+-)308
REAL*8	8	ditto
COMPLEX	8	2 reals (real-part, imaginary-part)
COMPLEX*8	8	ditto
LOGICAL	4	.FALSE. or .TRUE.
LOGICAL*4	4	ditto
LOGICAL*2	2	ditto
LOGICAL*1	1	ditto
CHARACTER*n	n	n (>= 1) 8-bit characters

It will be observed that, with a "storage unit" equal to 4 bytes, the rules of Fortran-77 concerning storage allocation are satisfied, namely: that a DOUBLE PRECISION or COMPLEX item occupies two consecutive storage units, and an INTEGER, REAL or LOGICAL item one storage unit.

The following are examples of legal type statements:

```
INTEGER*2  I2, K
REAL  MONEY, MEALS(7)
COMPLEX  CARR(10,20,10), CSUM, CSQR
CHARACTER*10  CA(20), CB, CC*12, CF * (*)
```

3.3 IMPLICIT statement

An IMPLICIT statement is provided as a way of altering the default type associations based on the first letter of the name (i.e. I thru N corresponding to INTEGER, the remaining letters to REAL). The syntax is:

```
implicit-statement = IMPLICIT implicit-declaration
                                {, implicit-declaration}
implicit-declaration = type-specifier
                          ( implicit-item {, implicit-item} )
implicit-item = letter [- letter]
```

For the definition of "type-specifier", refer to section 2.3.

As an example, the following statement classifies as double-precision all constants, variables, arrays and functions whose names begin with T thru Z and which are not explicitly typed in a type statement:

```
IMPLICIT  DOUBLE PRECISION (T-Z)
```

It is important to realise, however, that IMPLICIT statements have no effect on the types of the Intrinsic functions (section 7.2).

3.4 DIMENSION statement

A DIMENSION statement is used to specify the size and dimensionality (number of subscripts) of arrays.

```
dimension-statement = DIMENSION array-declarator
                                  {, array-declarator }
array-declarator =
        array-name ( subscript-bounds {, subscript-bounds} )
array-name = name
subscript-bounds = [ lower-bound :] upper-bound
lower-bound = integer-expression
upper-bound = integer-expression ¦ *
```

The dimensionality of the array may range from 1 to 7.

If the lower bound is omitted, the value 1 is assumed.

The bounds may take any positive or negative values. If all the bound expressions are constants, the array declarator is a "constant array declarator", otherwise, it is an "adjustable array declarator". The upper-bound of the last dimension may, optionally, be an asterisk, in which case the array declarator is an "assumed-size array declarator".

Adjustable and assumed-size arrays must be dummy arguments.

If a bound expression is not constant, it must have a value at the time the procedure is entered; it may not contain function calls and any variables or arrays referenced must either be dummy arguments or be in COMMON.

The elements of an array are held in storage in a definite order, which is given by the "subscript value" function. The effect of this is that, when moving through sequential storage locations, the first subscript of the array "cycles" fastest. As the simplest example, the elements of an array declared as

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

are held in storage in the sequence A(1,1), A(2,1), A(1,2), A(2,2).

The following example contains a declaration of an adjustable-dimension array (the first dummy argument), with its size being given by the second and third arguments (number of rows and number of columns, respectively):

```
SUBROUTINE  INVERT (A, M, N)
DIMENSION  A (M,N)
...
END
```

3.5 COMMON statement

A COMMON statement defines the contents of named common block(s) and/or blank common:

```
common-statement = COMMON [common-block]  common-item-list
                           { [,] common-block  common-item-list}
common-block = / [block-name] /
common-item-list = common-item {, common-item}
block-name = name
common-item = variable-name ¦ array-name ¦ array-declarator
```

If the "block-name" is omitted, the items in the corresponding common-item-list are allocated storage in blank common. A particular block-name (or blank common) may occur more than once in one COMMON statement, and/or in more than one COMMON statement; in such cases, the meaning is the same as if all the common-items had been specified in one common-item-list, in the order in which they occur in the source program.

Character and non-character items may not be mixed in the same common block.

A common-item may not be a dummy-argument.

A block-name may not be the same as that of any subroutine-name or function-name occurring in that program unit (but may be the same as the name of a variable, array or statement-function).

Storage is allocated sequentially for the items within each common block, in the order in which they occur in the source program.

As an example, the following statements define two common blocks: a named common block containing 3 real variables, and blank common containing an integer array:

```
COMMON /CB1/ A, B
COMMON  IARR(1000), /CB1/ X
```

3.6 EQUIVALENCE statement

The EQUIVALENCE statement is used to cause two or more data items to be allocated storage starting at the same address.

```
equivalence-statement = EQUIVALENCE equivalence-group
                                    {, equivalence-group}
equivalence-group = ( equiv-item , equiv-item {, equiv-item} )
equiv-item = variable-element | substring | array-name
variable-element = variable-name | array-element
```

If the "equiv-item" is an array element, then the number of subscripts must be equal to the dimensionality of the array (as specified in a type-, dimension- or common-statement), and each subscript must be an integer constant expression.

If the effect of EQUIVALENCE statement(s) is to cause two items to share storage, then at most one of those items may be in a common block.

EQUIVALENCE statements may have the effect of extending a common block, but only beyond the end of the storage allocated for the last item (not before the first item).

As an example of an EQUIVALENCE statement, the contents of a complex data item can be accessed as a byte array by the following specification statements:

```
COMPLEX  C
INTEGER*1  BA(8)
EQUIVALENCE  (C, BA(1))
```

3.7 SAVE statement

The SAVE statement is provided by Fortran-77 to enable the programmer to ensure that certain variables and arrays retain their values after execution of a RETURN or END statement in a subprogram. The syntax is:

```
save-statement = SAVE [ save-item { , save-item } ]
save-item = /block-name/ | variable-name | array-name
```

A SAVE statement with no save-items is treated as referring to all allowable items in that program unit.

If a common-block name is specified, it refers to all data items in that common block.

In Pro Fortran-77, all data items, in COMMON or otherwise, retain their values on exit from a procedure (except for arguments, of course - which are not permitted in a SAVE statement anyway). So SAVE statements are recognised but have no effect.

3.8 EXTERNAL statement

The purpose of EXTERNAL statements is to declare names to be those of external procedures (subroutines or functions) rather than variables or arrays. This is only necessary (i.e. there can only be confusion) if the name is to be passed as an actual argument.

external-statement = EXTERNAL procedure-name {, procedure-name}
procedure-name = subroutine-name | function-name

The name may not be that of a statement function, since this may not be passed as an actual argument. If the name is that of an Intrinsic function (see 7.2), then that becomes the name of an external procedure, and is not available as an intrinsic function within the program unit containing the EXTERNAL statement.

Example:

```
EXTERNAL  SUBA, SUBB, F
```

3.9 INTRINSIC statement

An INTRINSIC statement declares names to be the names of intrinsic functions. In particular, if the specific name of an intrinsic function (see section 7.2) is to be passed as an actual argument, then that name must appear in an INTRINSIC statement in the calling program unit.

intrinsic-statement = INTRINSIC function-name {, function-name }

Example:

```
INTRINSIC LEN, ATAN
```

3.10 PARAMETER statement

The PARAMETER statement enables constants to be given names.

```
parameter-statement = PARAMETER ( param-item {, param-item} )
param-item = constant-name = constant-expression
constant-name = name
constant-expression = expression
```

If the constant-name is of arithmetic type, the expression must be an arithmetic expression, and the same rules of type conversion apply as in an arithmetic assignment statement (see 5.1.1). If the constant-name is logical type, so must the expression be. If the constant-name is character type, so must the expression be, and the expression is truncated or blank-filled to fit the length of the constant-name's type.

Examples:

```
CHARACTER*20 CC
LOGICAL LC
 ..
PARAMETER (LENGTH = 178, CC = 'PERIMETER')
PARAMETER (LC = LENGTH .LT. 200)
```

4 DEFINITION STATEMENTS

There are three further types of non-executable statements that have not been described in sections 2 and 3, and were classified as "definitions" in section 2. These are:

statement function definition
data statement
format statement

Arbitrarily many (or none) of each of these 3 kinds of statement can appear, inter-mixed with one another, after the last specification statement and before the first executable statement.

As described in section 2, only the second kind of definition statement may appear in a block data subprogram.

As is also clear from the formal syntax in section 2, format statements may appear anywhere at all within a (non-block-data) program unit.

4.1 Statement function definition

A statement function is a sort of "local" function subprogram: when referenced, with an actual argument list, it returns a value, but it is only accessible within the program unit in which it is defined.

```
statement-function-definition =
            function-name ( [stf-argument-list] ) = expression
stf-argument-list = variable-name {, variable-name}
```

The names in the argument-list must be distinct from one another, but may be the same as variable names occurring elsewhere in the program unit. In particular, a type-statement can be used to give a dummy argument a type other than the default type associated with the first letter of its name.

The expression (see section 6) must be of a type which is assignable to the type of the function name; for example, if the latter is logical-type, the expression must be a logical expression. If the expression contains references to other statement functions, then the latter must already have been defined earlier in the program unit.

As an example, the following defines a statement function which converts a lower-case letter to upper-case:

```
CHARACTER*1 UPPER, CH
...
UPPER(CH) = CHAR( ICHAR(CH) - 32)
```

4.2 DATA statement

DATA statements are used to define initial values of variables, array elements or whole arrays.

```
data-statement = DATA data-initialisation
                        { [,] data-initialisation }
data-initialisation = variable-list / constant-list /
variable-list = initialised-item {, initialised-item}
constant-list = initial-setting {, initial-setting}
initialised-item = variable-element ¦ substring ¦
                   array-name ¦ data-implied-do
initial-setting = [ <unsigned-integer ¦ constant-name> * ]
                  < constant ¦ constant-name >
data-implied-do = ( implied-do-item {, implied-do-item}
                    , do-control )
implied-do-item = array-element ¦ data-implied-do
```

The syntax of "do-control" is given with that of DO statements, in section 5.2.13.

The "variable-list" may not contain any dummy arguments, nor any items in blank common. An item in a labelled common block may only be initialised in a DATA statement in a BLOCK DATA subprogram.

Specifying an array-name in a variable-list is equivalent to enumerating all its elements in the order determined by the subscript value function (see 3.4). For example, in the fragment:

```
INTEGER  IA(2,2)
DATA  IA / 1,2,3,4 /
```

the data statement is completely equivalent to:

```
DATA  IA(1,1), IA(2,1), IA(1,2), IA(2,2) / 1,2,3,4 /
```

The number of items in a variable-list (with an array name counting as N items, where N is the number of elements in the array) must be equal to the number of constants in the associated constant-list (with a repeat-count of M causing the constant to be counted M times). For example, the following data statement is correct in this respect:

```
REAL  RA(3), RS
DATA  RA,RS / 2*0.0, 2*1.0 /
```

Character constants may appear in the constant-list. If the size M bytes of the corresponding character variable is smaller than the number of characters N in the character constant, then the variable is initialised to the left-most M characters of the constant; if an the other hand M is larger than N, the right-most (M-N) bytes of the variable are initialised to blanks. For example, the following statements cause C1 and C4 to be initialised to 'A' and 'AB ', respectively:

```
CHARACTER*1  C1
CHARACTER  C4
DATA  C1,C4 / 2*'AB' /
```

4.3 FORMAT statement

FORMAT statements are used to control the editing and conversion of data during "formatted" input and output (see 5.3). A FORMAT statement must be labelled, the label being used to reference the format from an input/output statement.

```
format-statement = FORMAT format-specification
format-specification = ( [format-list] )
format-list = format-item { , format-item }
format-item = [repeat-count] repeatable-descriptor ¦
              nonrepeatable-descriptor ¦
              [repeat-count] ( format-list )
repeat-count = decimal-integer
repeatable-descriptor = D-descriptor ¦ E-descriptor ¦
                        F-descriptor ¦ G-descriptor ¦
                        I-descriptor ¦ L-descriptor ¦
                        A-descriptor
nonrepeatable-descriptor =
                apostrophe-descriptor ¦ H-descriptor ¦
                T-descriptor ¦ X-descriptor ¦
                slash-descriptor ¦ colon-descriptor ¦
                S-descriptor ¦ P-descriptor ¦
                B-descriptor
```

Note the mutual recursion in the definitions of "format-list" and "format-item". The maximum depth of (..) bracketing permitted is 7. A group of descriptors enclosed within parentheses is called a "basic group". Basic groups and descriptors can, in turn, be enclosed within parentheses to form a "group", and this nesting then be repeated (up to a maximum depth of 7).

The presence of a repeat count before a descriptor or group causes that descriptor or group to be used for format control repetitively, the stated number of times.

The syntax of the various kinds of descriptor, and the format control interaction of each with the associated input-output list element, are described in the following sub-sections. A more general discussion of "format control" will be found in section 5.3.

4.3.1 Real descriptors

There are 4 of these:

```
D-descriptor = D w . d
E-descriptor = E w . d [ E e ]
F-descriptor = F w . d
G-descriptor = G w . d [ E e ]
w = decimal-integer
d = decimal-integer
e = decimal-integer
```

The D descriptor is intended for use with double-precision list elements, but can also be used with real (or complex) elements. The E, F and G descriptors are intended for use with real (or complex) list elements, but can also be used with double-precision elements. If the list element is complex-type, its conversion is specified by a pair of consecutive real-descriptors, the first of which controls the real part and the second the imaginary part.

Any of these descriptors may optionally be preceded by a P edit descriptor (see 4.3.12) to establish a scale factor. Once established, such a scale factor remains operative for the remainder of the format-specification, or until replaced by another scale factor. The "current" scale factor will be denoted by k in the following subsections. (If no scale factor is explicitly specified, k is zero.)

4.3.1.1 D-descriptor

The real descriptor Dw.d specifies an external representation of width w character positions, with a fractional part consisting of d decimal digits.

For input, the w character-positions may contain an integer-constant, a real-constant or a double-precision-constant, where these terms are as defined in 1.4.3 above; in the case of a real- or double-precision constant with a signed exponent, the D or E character may, optionally, be omitted. Leading blanks are ignored; embedded blanks are ignored or treated as zeros, depending on BN/BZ editing (see 4.3.13). If a decimal point is supplied, its position overrides the position implied by d; if no decimal point is supplied, the position of the implied decimal point is d digits before the end of the mantissa. If no exponent is present in the external representation of the constant, the value assigned to the list-element is the value of the external constant divided by 1.0Dk, where k is the current scale factor. If the external constant includes an explicit exponent, the scale factor has no effect. Reading stops when w characters have been read or (as a Pro Fortran-77 extension to the Standard) when a comma (,) or end-of-record is encountered. Use of a comma or end-of-line to terminate a number is particularly useful when inputting from an interactive device.

For output, assuming a current scale factor of k, the w character-positions consist of the following:

blank(s), if necessary, to make the total width up to w

-, if the value is negative

0, if k is negative or zero, or
first k significant digits of mantissa

. (decimal point)

(-k) zeros and (d+k) digits, if k is negative or zero, or
remaining (d-k+1) digits of decimal representation

D, if magnitude of exponent less than 100

+ or - (sign of exponent)

2-digit exponent, if magnitude of exponent less than 100, or
3-digit exponent

If w is less than 6, the value can not be represented in this format and the output field is filled with asterisk (*) characters.

As an example of output, if the internal value of the list element is 0.123456789D13 and k = 0 and the descriptor is D16.9, then the output field contains ' 0.123456789D+13'. If k = 3, or if the descriptor was 3PD16.9, then ' 123.4567890D+10' would be written.

4.3.1.2 E-descriptor

The real descriptor Ew.d specifies an external representation of width w character positions, with a fractional part consisting of d decimal digits.

The rules governing input under this format descriptor are as for the D-descriptor.

The rules governing output are also as for the D-descriptor, with the difference that E rather than D is used to introduce the exponent.

The optional "Ee" in an E- or G-descriptor specifies that, on output, the exponent part is to have e digits; it has no effect on input.

As an example of output, if the internal value of the list element is -3.14159265 and k = 0 and the descriptor is E14.7, then the output field is '-0.3141593E+01'. If k = -2, or if the descriptor was -2PE14.7, then '-0.0031416E+03' would be written.

4.3.1.3 F-descriptor

The real descriptor Fw.d specifies an external representation of width w character positions, with a fractional part consisting of d decimal digits.

The rules governing input under this format descriptor are as for the D-descriptor.

For output, assuming a current scale factor of k, the w character-positions consist of the following:

blank(s), if necessary, to make the total width up to w

-, if v is negative

integral part of v

. (decimal point)

d digits representing the fractional part of v rounded to d digits

where v = the internal value of the list element divided by 1.0Ek.

As an example of output, if the internal value of the list element is 0.987654E+2 and the descriptor is F6.2 (and k = 0), then the output field is ' 98.77'.

4.3.1.4 G-descriptor

The real descriptor Gw.d specifies an external representation of width w character positions, with d significant digits.

The rules governing input under control of this format descriptor are as for the D-descriptor (see 4.3.1.1).

For output, an equivalent E- or F-descriptor is employed, depending on the internal value, X say, of the list element. Define

```
Y = ALOG10(10*X)
N = IFIX(Y)
```

If N is in the range 0 thru d, then the output field is formed just as it would be for a scale factor of zero and under control of the format descriptors

Fu.e, 4X

where u = (w-4)
and e = (d-N)

(See 4.3.8 for an account of the X descriptor.)

If N is outside the range 0 thru d, then the output field is formed as for the descriptor Ew.d (see 4.3.1.2), with due account being taken, in this instance, of the scale factor too.

As an example, suppose X = 0.1234E+5 and the format descriptor is 2PG10.6. Then N = 5. This is in the range 0 thru d (=6), so the equivalent descriptor used is

F6.1, 4X

and, by the rules given in 4.3.1.3, the output field therefore consists of ' 123.5 '.

4.3.2 Integer descriptor

I-descriptor = I w [. m]
m = decimal-integer

The integer descriptor is for use with integer-type list elements.

For input, the external representation must contain a decimal integer-constant, as defined in 1.4.3.1 above. Leading blanks are ignored; embedded blanks are ignored or treated as zeros, depending on BN/BZ editing (see 4.3.13). A field consisting of all blanks is treated as zero. Reading stops when w characters have been read, or when a comma or end-of-record is encountered, whichever is the earlier. An Iw.m descriptor is treated identically to an Iw descriptor.

For output, the w character-positions of the output field contain a right-justified integer constant; that is to say:

blank(s), if necessary, to make the total width up to w

-, if the value of the list element is negative

digit(s) representing the value of the list element

If w is too small for the value to be represented in this way, the output field is filled with asterisk (*) characters.

The output for an Iw.m descriptor is the same as for Iw, except that the decimal integer representing the value of the list item consists of at least m digits, and if necessary has leading zeros. The value of m must not exceed w.

As an example of output, if the internal value of the list element is -32768 and the descriptor is I8, then the output field will consist of ' -32768'.

4.3.3 Logical descriptor

L-descriptor = L w

The logical descriptor is for use with logical-type list elements.

For input, the external representation must consist of optional blanks, optionally followed by a decimal point, followed by the character T or F followed by optional additional characters, for true or false, respectively. In particular, these rules imply that .TRUE. and .FALSE. are acceptable.

For output, the w character-positions of the output field consist of (w-1) blanks followed by the character T or F, according as the internal value of the list element is true or false, respectively.

As an example of input, if the record consists of the characters TRUE, then the value true will be assigned to the logical-type list element.

4.3.4 Character descriptor

A-descriptor = A [w]

The A-descriptor is for use with character-type list elements. Let len be the length of the list item. If w is omitted, the value of len is assumed.

For input, if w >= len, the rightmost len characters will be taken; if w < len, w characters will be taken from the input field and (len - w) trailing spaces will be appended.

Reading stops when w characters have been read or (as a Pro Fortran-77 extension to the Standard) when a comma (,) or end-of-record is encountered.

For output, if w > len, the output field will consist of (w-len) leading blanks followed by len characters from the list item; if w <= len, the left-most w characters of the item will be output.

As an output example, a character list item containing the value 'ABCD' will be written, under control of the format descriptor A3, as 'ABC'.

4.3.5 Apostrophe editing

apostrophe-descriptor = character-constant

This descriptor causes characters to be transferred from the descriptor itself to the external record. No list item is involved. This edit descriptor must not be used on input.

As an example, the descriptor 'Hello' causes those 5 characters to be written to the external record.

4.3.6 H editing

H-descriptor = n H {string-character}
n = decimal-integer

The n characters (including blanks) following the H of the descriptor are written to the output record. No list item is involved. This edit descriptor must not be used on input.

As an example, the descriptor 5HHello causes Hello to be written to the external record.

4.3.7 T editing

The function of T editing is to alter the position at which characters are transferred to or from a record.

T-descriptor = < T | TL | TR > c
c = decimal-integer

The Tc edit descriptor makes the next character transfer to or from a record occur at the c'th character position.

The TLc descriptor sets the transfer position c characters backwards from the current position.

The TRc descriptor sets the transfer position c characters forward from the current position.

4.3.8 X editing

The function of X editing is to alter the position at which characters are transferred to or from a record.

X-descriptor = n X

The effect of nX is the same as the effect of TRn (see 4.3.7).

4.3.9 Slash editing

The function of the slash edit descriptor is to indicate the end of data transfer on the current record.

slash-descriptor = /

A format describes the layout of data, generally in an external file. Fortran uses the word "record" to describe groupings of data within a file, and in any "formatted" file a record is effectively equivalent to a line of text. One format may describe the layout of one or more records (lines), and for this purpose the slash symbol (/) denotes the separation of records.

For example, the format

(2I10 / 1X, F12.4, 2E14.4)

describes a data layout with two integer values on one line and three real values on the next line.

4.3.10 Colon editing

colon-descriptor = :

The colon edit descriptor terminates format control (see 5.3) if there are no more items in the input/output list; otherwise, it has no effect.

4.3.11 S editing

S-descriptor = S | SP | SS

These descriptors control optional "+" characters in numeric output fields. They have no effect on input.

At the start of each input-output statement, the printing of a + in a real (D, E, F, or G edit descriptor) or integer (I descriptor) output field is under control of the Fortran system. By using an SP descriptor, the program can force printing of a +; by using an SS descriptor, the program can force suppression of a +; by using an S descriptor, printing reverts to being under control of the Fortran system.

4.3.12 P editing

A P edit descriptor establishes a scale factor, which affects the interpretation of real edit descriptors (see 4.3.1).

```
P-descriptor = k P
k = [sign] decimal-integer
```

At the start of processing an input-output statement, the scale factor is zero.

The effect of a P descriptor lasts until another such descriptor is encountered. For example, in the format statement

```
FORMAT (-2PE10.6, G10.6)
```

both the E- and the G-descriptors will be processed with a scale factor of -2.

4.3.13 BN and BZ editing

These descriptors enable a program to control the interpretation given to blanks encountered in numeric input.

```
B-descriptor = BN | BZ
```

At the start of a formatted input statement, blanks are either ignored or interpreted as zeros, depending on the BLANK= specifier in the OPEN statement (see 5.3.7).

A BN descriptor causes all such blank characters in succeeding numeric input fields to be ignored.

A BZ descriptor causes all such blank characters in succeeding numeric input fields to be treated as zeros.

These descriptors affect only real (D-, E-, F- or G-descriptors) or integer (I-descriptor) editing. They have no effect on output.

5 EXECUTABLE STATEMENTS

Apart from block data subprograms, every program unit has an "executable part", as defined in 2.1. This consists primarily of "executable statements", but these can be mixed with DATA, ENTRY and FORMAT statements.

Executable statements can be classified into three groups:

```
executable-statement = assignment-statement ¦ control-statement ¦
                       input-output-statement
```

5.1 Assignment statements

These are of four kinds, depending on whether the thing being assigned is an arithmetic quantity, a logical quantity, a character quantity, or a statement label:

```
assignment-statement = arithmetic-assignment ¦
                       logical-assignment ¦
                       character-assignment
                       label-assignment
```

5.1.1 Arithmetic assignment

An arithmetic assignment statement causes the value of an arithmetic expression (see 6.1) to be assigned to a variable or array element of arithmetic (i.e. integer, real, double-precision, or complex) type.

```
arithmetic-assignment = variable-element = arithmetic-expression
variable-element = variable-name ¦ array-element
array-element = array-name ( subscript {, subscript} )
```

All combinations of "left-hand-side" and "right-hand-side" types are allowed, values being converted to the target type, by truncation if necessary, before the assignment takes place. For example, if a real- or double-precision-type expression is assigned to an integer-type variable, the value is converted before assignment to the nearest integer whose magnitude does not exceed the magnitude of the expression. For example, after the assignment statement

```
K = -3.7
```

K (assuming it to be of type integer) will have the value -3.

5.1.2 Logical assignment

A logical assignment statement causes a logical expression (see 6.2) to be assigned to a logical-type variable or array-element:

```
logical-assignment = variable-element = logical-expression
```

Example:

```
LOGICAL  L, L1, L2
...
L = L1 .AND. L2
```

5.1.3 Character assignment

A character assignment statement causes a character expression (see 6.3) to be assigned to a character-type variable, array-element or substring:

```
character-assignment = character-field = character-expression
character-field = variable-element | substring
substring = variable-element ( [substring-expression] :
                               [substring-expression] )
substring-expression = integer-expression
```

If the right-hand side expression is shorter than the left-hand side destination, blanks are added on the end; if longer, the left-most characters are stored.

(For more details on "substring", see section 6.3.)

Example:

```
CHARACTER*(L) C1, C2, C3
...
C1(5:I) = C2(1:4) // C3
```

5.1.4 Label assignment

This kind of assignment statement is used, preparatory to an "assigned GOTO" statement (see 5.2.1.2) or input-output statement (see 5.3), to assign (the address of) a statement-label to an integer-type variable:

```
label-assignment = ASSIGN statement-label TO integer-variable
integer-variable = variable-name
```

The type of the variable must be INTEGER (i.e. INTEGER*4).

Example:

```
ASSIGN 90 TO ILABEL
```

5.2 Control statements

Control statements have to do with with the flow of control within a program. There are 14 kinds:

```
control-statement = goto-statement ¦ arithmetic-if-statement ¦
                    logical-if-statement ¦ block-if-statement ¦
                    else-if-statement ¦ else-statement ¦
                    end-if-statement ¦ end-statement ¦
                    call-statement ¦ return-statement ¦
                    pause-statement ¦ stop-statement ¦
                    continue-statement ¦ do-statement
```

5.2.1 GOTO statements

There are 3 of these:

```
goto-statement =
       unconditional-goto ¦ assigned-goto ¦ computed-goto
```

5.2.1.1 Unconditional GOTO

An unconditional GOTO causes control to be transferred to a specified statement label.

```
unconditional-goto = GOTO statement-label
```

Example:

```
GOTO 100
```

5.2.1.2 Assigned GOTO

An assigned GOTO causes control to be transferred to the statement label whose address is currently held in a specified integer-type variable (having been put there by a preceding ASSIGN statement).

```
assigned-goto = GOTO integer-variable [ [,] label-list ]
label-list = ( statement-label {, statement-label} )
```

The type of the variable must be INTEGER. The label list, which is optional, does not affect the execution of the statement in any way.

Example:

```
GOTO ILABEL , (10, 20, 90)
```

5.2.1.3 Computed GOTO

A computed GOTO causes control to be transferred to one from a list of statement labels, which one depending on the value of an integer-type expression.

```
computed-goto = GOTO label-list [ , ] integer-expression
```

(This is the Fortran analogue of the Pascal CASE statement.)

Let m be the number of labels in the list, and i the value of the integer expression. If i is less than 1 or greater than m, then the statement acts as a CONTINUE (i.e. does nothing), otherwise control is transferred to the i'th label in the list.

Example:

```
GOTO  (10, 20, 90), K
```

5.2.2 Arithmetic IF statement

An arithmetic IF causes an arithmetic expression (see 6.1) to be evaluated and then control to be transferred to the first, second or third of three labels, according as the value of the expression is less than, equal to or greater than zero, respectively.

```
arithmetic-if-statement =
     IF ( arithmetic-expression ) statement-label ,
                                  statement-label , statement-label
```

The expression may be integer-, real- or double-precision-type.

Example:

```
IF (Y - SIN(X)) 40, 50, 50
```

5.2.3 Logical IF statement

A logical IF causes a logical expression (see 6.2) to be evaluated and then a second statement to be conditionally executed; namely, the second statement is executed if, and only if, the logical expression has the value true.

```
logical-if-statement =
                IF ( logical-expression ) executable-statement
```

The "executable-statement" may not be another logical IF statement, nor may it be a DO, Block IF, ELSE IF, ELSE, END IF or END statement.

Example:

```
IF ((I .EQ. 0) .OR. (I .LE. J))  GOTO 190
```

5.2.4 Block IF statement

The Block IF statement is used, with the ELSE IF, ELSE and END IF statements, to control the flow of execution, dependent on the values of logical expressions.

block-if-statement = IF (logical-expression) THEN

To describe the control flow, the concept of "IF-level" is introduced. The IF-level of any statement S is defined to be

n1 - n2

where n1 is the number of Block IF statements from the start of the program unit up to and including S, and n2 is the number of END IF statements from the start of the program unit up to but not including S.

An "IF block" consists of all the executable statements following a Block IF statement up to, but not including, the next ELSE IF, ELSE or END IF statement with the same IF-level as the Block IF.

If the logical expression in a Block IF statement evaluates to true, execution continues with the IF block.

If the logical expression is false, execution continues with the next ELSE IF, ELSE or END IF statement with the same IF-level as the Block IF.

The following source fragment illustrates the features provided by the Block IF and its associated statements.

```
IMPLICIT LOGICAL (L)
 ..
IF (LOG1) THEN
   X = 1.0
   IF (LOG2) THEN
      Y = 2.0
   ELSE
      Y = 0.0
   ENDIF
ELSE IF (LOG3) THEN
   X = 3.0
ELSE
   X = 0.0
END IF
```

5.2.5 ELSE IF statement

else-if-statement = ELSE IF (logical-expression) THEN

An "ELSE IF block" consists of all the executable statements following an ELSE IF statement up to, but not including, the next ELSE IF, ELSE or END IF statement with the same IF-level as the ELSE IF.

If the logical expression in an ELSE IF statement evaluates to true, execution continues with the ELSE IF block.

If the logical expression is false, execution continues with the next ELSE IF, ELSE or END IF statement with the same IF-level as the ELSE IF.

5.2.6 ELSE statement

else-statement = ELSE

An "ELSE block" consists of all the executable statements following a Block IF statement up to, but not including, the next END IF statement with the same IF-level as the ELSE.

Execution of an ELSE statement has no effect.

5.2.7 END IF statement

end-if-statement = END IF

Execution of an END IF statement has no effect.

Each Block IF in a program unit must have a matching END IF statement, i.e. one with the same IF-level.

5.2.8 CALL statement

A CALL statement causes control to be transferred to a named subroutine; optionally, a list of arguments may be passed at the same time. Control returns to the statement next in sequence after the CALL statement when a RETURN or END statement is executed in the subroutine, or to a label in the program unit containing the CALL statement if an alternate return specifier is activated.

```
call-statement = CALL subroutine-name
                     [ ( [actual-argument-list] ) ]
actual-argument-list = actual-argument {, actual-argument}
actual-argument = expression ¦ variable-element ¦
                  array-name ¦ procedure-name ¦
                  alternate-return-specifier
alternate-return-specifier = * statement-label
```

In this context, "expression" (see section 6) must be understood to mean any expression other than "variable-element".

The actual arguments must agree in number, order, kind and type with the dummy arguments in the SUBROUTINE statement (see 2.2) at the start of the subroutine being called.

By agreement in "kind" of argument is meant the following. As is clear from 2.2, a dummy argument can name a variable or an array or a procedure (i.e. a subroutine or function). If the dummy argument is a variable-name, the corresponding actual argument may not be a procedure-name; furthermore, if the dummy argument is defined (that is, has a value assigned to it) in the subroutine, then the actual argument must be a variable-element (i.e. a variable or an array element) or an array-name. If the dummy argument is an array-name, the corresponding actual argument must be an array-name or an array-element. If the dummy argument is a subroutine- or function-name, then the actual argument must also be a subroutine- or function-name, respectively.

By agreement in "type" of argument is meant the following. If the dummy argument is a variable- or function-name, then the actual argument must be of exactly the same type (REAL matching REAL, INTEGER*1 matching INTEGER*1, and so on). Further, if the dummy argument is a variable-name and the actual argument is an expression (other than "variable-element"), then the dummy argument cannot be INTEGER*1, INTEGER*2, LOGICAL*1 or LOGICAL*2. This is because actual argument expressions which are more general than "variable-element" are evaluated, placed in a temporary store, and the address of that temporary location passed to the called procedure. In the case of integer or logical expressions, the temporary location contains an INTEGER*4 or LOGICAL*4 result, respectively; so that, by the above-mentioned type-matching rule, the corresponding dummy argument must be of type INTEGER*4 or LOGICAL*4, respectively.

As an example of CALL statements, suppose a subroutine subprogram begins:

```
SUBROUTINE  S(ARRAY, IARG, LARG)
REAL  ARRAY(100)
INTEGER  IARG
LOGICAL  LARG
...
```

then, in a program unit with the following specification statements:

```
REAL  A(10), X
INTEGER  I(10), J
LOGICAL*1  L1
...
```

the following CALLs of the subroutine S are all legal:

```
CALL  S(A, J, .NOT. L1)
CALL  S(A(9), 3, .TRUE.)
CALL S(A(J), (300-J), L .OR. J .LT. 1)
```

and the following CALLs are all illegal:

```
C   Wrong number of arguments
      CALL  S(A, I)

C   Wrong kinds of arguments
      CALL  S(X, MAX, S)

C   Wrong types of arguments
      CALL  S(I, (J.EQ.0), L1)
```

In section 7.2 are listed a number of "intrinsic functions" which are often useful when a value has to be coerced to the correct type to match a dummy argument. The function FLOAT, for instance, can be applied to an integer value to convert it to real type, and INT performs the reverse process. Continuing the example above, INT(X) would be legal as the second argument of a call, whereas X alone is wrong.

5.2.9 RETURN statement

The purpose of a RETURN statement is to leave the subprogram containing it and return control to the calling program unit.

```
return-statement = RETURN [ integer-expression ]
integer-expression = expression
```

A RETURN statement is not allowed in a main program.

In a subroutine subprogram, a RETURN may optionally be followed by an integer expression. If this expression e lies in the range

```
1 <= e <= n
```

where n is the number of asterisks in the SUBROUTINE (or ENTRY) statement for this execution of the subprogram, then the effect of the RETURN is to transfer control to the label of the e'th alternate return specifier in the CALL statement for this execution of the subprogram.

In a function subprogram, execution of a RETURN (or END) statement makes the value of the function available to the calling program unit. No alternate return expression may be given.

Example (as the subsidiary statement of a logical IF):

```
IF (X .GT. 1E37)  RETURN 1
```

5.2.10 PAUSE statement

A PAUSE statement causes execution to be suspended with, optionally, the output of up to 5 digits, or a character constant, to the console device:

```
pause-statement = PAUSE [ digit-string ¦ character-constant ]
```

Depending on the operator's response, the program will be resumed (at the statement following the PAUSE) or terminated. For more details, see Part III, section 4.2.5.

Example:

```
PAUSE  'Initialisation complete'
```

5.2.11 STOP statement

A STOP statement causes execution to be terminated with, optionally, the output of up to 5 digits, or a character constant, to the console.

```
stop-statement = STOP [ digit-string ¦ character-constant ]
```

Example:

```
STOP
```

5.2.12 CONTINUE statement

This is the "no-op" statement: it does nothing.

```
continue-statement = CONTINUE
```

Its most common use (and this is good Fortran coding practice) is as the recipient of a statement label; programs are thereby made more maintainable and often more readable.

As an example, a CONTINUE may be used to terminate a DO loop:

```
      DO 120  I = 1, 1000
      A(I) = A(I) / FLOAT(I)
120   CONTINUE
```

5.2.13 DO statement

A DO statement is used to define a loop.

```
do-statement = DO statement-label [ , ] do-control
do-control = control-variable = initial-value ,
                          terminal-value [ , increment-value ]
control-variable = variable-name
initial-value = arithmetic-expression
terminal-value = arithmetic-expression
increment-value = arithmetic-expression
```

The control-variable must be of integer-, real- or double-precision type, as must the initial-, terminal- and increment-value expressions.

If an increment value is not supplied, the value 1 is assumed.

The statement-label must be the label of a statement (called the terminal statement) which is in same program unit as, but physically following, the DO statement.

The terminal statement may not be an unconditional-goto-statement, assigned-goto-statement, arithmetic-if-statement, block-if-statement, else-if-statement, else-statement, end-if-statement, return-statement, stop-statement, end-statement, or another do-statement. If the terminal statement is a logical-if-statement, it may contain any executable statement except a do-statement, block-if-statement, else-if-statement, else-statement, end-if-statement, end-statement or another logical-if-statement.

When a DO statement is executed, the following occurs. The initial-, terminal- and increment-values are computed - say e1, e2 and e3, respectively. (e3 must not be zero.) The control-variable is given the value e1, converted if necessary to the type of the control-variable as in an arithmetic assignment. An "iteration count" is computed, as the value of:

```
MAX (INT ( (e2 - e1 + e3)/e3), 0)
```

Now "loop control processing" is performed, as follows. If the iteration count is zero, the body of the DO loop is skipped. Otherwise, processing continues with the statement following the DO statement.

When the terminal statement is reached, it is executed, and then the following "incrementation processing" is performed. The control-variable is incremented by e3. The iteration count is decremented by one. Then control jumps back to the "loop control processing" of the DO statement (i.e. test the iteration count for zero, etc).

Example:

```
      DO  100  I = IFIRST, ILAST
      ISUM2 = ISUM2 + I*I
100   ISUM3 = ISUM3 + I**3
```

5.2.14 END statement

The END statement is the last line of every program unit.

```
end-statement = END
```

The characters "END" must be written in columns 7 thru 72 of an initial line, and there may not be any continuation lines.

If executed in a subroutine or function subprogram, an END statement has the same effect as RETURN (see 5.2.9). If executed in a main program, it has the same effect as STOP (see 5.2.11).

5.3 Input-output statements

Input-output statements provide for the transfer of values between internal storage and external devices or files, or between internal storage and internal files, and for the control and positioning of devices or files.

Both sequential and random (or "direct") processing of files is supported.

There are 9 kinds of input-output statement:

```
input-output-statement = read-statement ¦ write-statement ¦
                         print-statement ¦ backspace-statement ¦
                         endfile-statement ¦ rewind-statement ¦
                         open-statement ¦ close-statement ¦
                         inquire-statement
```

All input-output statements, apart from PRINT and INQUIRE, require a "unit specifier".

```
unit-specifier = [ UNIT = ] unit-identifier
unit-identifier = integer-expression ¦ * ¦
                  array-name ¦ character-field
```

A unit-identifier which is an integer expression identifies an external unit. The value of the expression must be in the range 0 to 255. At any one time, a number of devices or files are "active", in the sense of being known to the program, and each of these is associated with one or more unit numbers; a unit number can be associated with at most one device or file at any one time.

A unit-identifier which is an asterisk identifies one of two devices which are "pre-connected" for sequential formatted access, one for input and one for output.

A unit-identifier which is an array-name or character-field (i.e. character-type variable, array-element or substring) refers to an "internal file". If an array-name is used, the array must be of character type, and the internal file consists of as many records as there are elements in the array. If a character-type variable, array element or substring is used, the internal file consists of just one record. In either case, the record size is equal to the length of the character item.

READ, WRITE and PRINT statements make use of the concepts of "format specifier" and "format identifier":

```
format-specifier = [ FMT = ] format-identifier
format-identifier = statement-label ¦ integer-variable ¦
                    array-name ¦ character-expression ¦ *
```

The presence of a format identifier signifies a "formatted" input-output operation.

If a statement-label is specified, it must be the label of a FORMAT statement, and the corresponding format-specification is used.

If an integer variable-name is specified, the variable must, at the time of execution of the input-output statement, have been ASSIGNed the label of a FORMAT statement.

If the format-identifier is an array name, the array must be of character type, and the contents of the array, taken in order from its beginning, must constitute a valid format-specification as defined in 4.3, beginning with a left parenthesis and ending with a right parenthesis. (The contents of the array beyond the right parenthesis are immaterial.) The format specification may have been inserted into the array by means of a DATA statement, or by means of a READ statement with an A-descriptor (see 4.3.4), for example.

If the format-identifier is a character expression, the value of the expression must represent a valid format format-specification (see 4.3).

"Formatted" input-output operations take place under what is known as "format control", according to the following prescription. The start of execution of a formatted input-output statement initiates format control. Each action of format control depends on information provided jointly by the next io-element (if one exists) and the next descriptor. Records are read or written only as the format specification demands; in particular, each slash (/) causes a new record to be started (cf. 4.3.9). During an input operation, any unprocessed characters from the current record are skipped at the time the next record is started. Whenever format control encounters a D, E, F, G, I, L or A descriptor (see 4.3), it determines whether there is a corresponding io-element specified by the io-list (see 5.3.1); if so, appropriate information is transmitted between the internal element and the record, and format control proceeds; if not, format control terminates. Whenever format control reaches the last outer right parenthesis of the format specification, it determines whether another io-element is specified; if so, a new record is started, and control reverts to the beginning of the "group" (see 4.3) terminated by the last preceding right parenthesis, or if none exists, then to the first left parenthesis of the format specification; if not, format control terminates. When format control terminates, for an input operation any remaining characters in the current record are skipped, and for an output operation the current record is written.

If the format-identifier is an asterisk, "list-directed" input/output is selected. In this case, there is no "format control" (see above), but rather the type of the io-list item determines the processing.

For list-directed input, each external record contains one or more values, separated by blanks, tab characters, commas or semi-colons. In addition, a value may be preceded by a repeat count and an asterisk, as in:

```
12 * 3.142
```

which is equivalent to 12 successive occurrences of 3.142 (separated by spaces). The length of each input record may not exceed the maximum variable-length formatted record size (see section 8.1.1). For numeric items, the valid form of external representation is as for F-descriptor input editing (see 4.3.1.3). A complex value must be represented by a pair of real-type values separated by a comma, all enclosed in parentheses. For logical items, the valid form of external representation is as for L-descriptor input editing (see 4.3.3). In the case of character items, the external representation consists of a non-empty string of characters enclosed in apostrophes; such a literal may extend over more than one line, but is limited in length to not more than 255 characters.

In list-directed output, records are not more than 81 characters in length, the first of which is a blank to give single-spaced printing (cf. 8.1.2.1). Items are separated by blanks. Integer-type items are written with an I edit descriptor (see 4.3.2), real- and double-precision items are written with either E or F edit descriptors (see 4.3.1), depending on the magnitude of the value. Complex-type values are written as a pair of real values separated by commas and enclosed in parentheses. Logical-type values are written as T (for true) or F (for false). Character-type values are written without apostrophes.

The individual input-output statements are described in the following subsections.

5.3.1 READ statement

A READ statement causes record(s) to be read from an external device or file, or an internal file, and the values optionally to be assigned to variables and/or array elements and/or arrays.

```
read-statement = < READ ( read-write-control ) [io-list] ¦
                   READ format-identifier [,io-list] >
read-write-control = unit-specifier
                     [, format-specifier]
                     [, REC = integer-expression]
                     [, IOSTAT = integer-variable-element]
                     [, END = statement-label]
                     [, ERR = statement-label]
integer-variable-element = variable-element
io-list = io-item {, io-item}
io-item = io-element ¦ io-implied-do
io-element = variable-element ¦ array-name ¦
             substring ¦ expression
io-implied-do = ( io-list , do-control )
```

The second form of READ statement is equivalent to the following variant of the first form:

```
READ (*, format-identifier) [io-list]
```

If a format-specifier (see 5.3) is present, the statement is a formatted READ, and execution of the statement proceeds under format control (see 5.3). If the format-specifier has no "FMT=", it must appear immediately after the unit-specifier.

If no format-specifier is present, the statement is an unformatted READ. In this case, the next record is read from the external device or file. This must be an unformatted record, that is, it must have been previously written by an unformatted WRITE statement (or by equivalent non-Fortran methods). If there is an io-list, the values in the record are assigned to the sequence of io-elements specified by the io-list.

The remaining 4 options in "read-write-control" are each "keyword parameters", and may be specified in any order.

If REC= is specified, the integer expression determines the record number at which the READ operation is to commence. This value must be greater than zero. (The first record in the file is numbered 1.) This parameter may be used for either formatted or unformatted operations. The file must be an external file, and have been defined to contain fixed-length records by use of the RECL= parameter in the OPEN statement (see 5.3.7).

If IOSTAT= is specified, the integer variable or array element (which must be of type INTEGER*4) will be set to a status value on completion of the READ statement. Zero indicates successful completion, a negative value end-of-file and a positive value an error condition (the exact values used are implementation-specific, and are described more fully in section 8 and Appendix C).

If END= or ERR= is specified, the statement label must be the label of an executable statement in the same program unit. Control will be transferred to this label if, during execution of this READ statement, end-of-file is encountered (END=) or an error condition is detected (ERR=), respectively.

In general, an "io-list" is made up of variables, array-elements, array-names, substrings and general expressions (these are the "io-elements"). In a READ statement, however, substrings and expressions (other than variable-element) are not allowed. Variables and array-elements stand for themselves, while an array-name stands for all its elements (in the order dictated by the subscript value function, see 3.4).

In an "io-implied-do", the "do-control" causes all the items in the "io-list" to be selected repeatedly, in order from left to right, the repetition being governed by the initial-, terminal- and increment-values just as for a DO statement (cf. 5.2.13).

As an example of a formatted READ, the following will cause a varying number of integers to be read in, the number being specified by the first digit in the record:

```
      INTEGER*1  I,LENGTH
      INTEGER  IARR(9)
      ...
      READ  (1, 800)  LENGTH, (IARR(I), I = 1, LENGTH)
800   FORMAT(I1, 9I10)
```

As an example of an unformatted READ, the following fragment will obtain the 12000th complex value from a work file:

```
      COMPLEX  C
      ...
      OPEN  (6, RECL=8, ACCESS='DIRECT')
      ...
      READ  (6, REC=12000) C
```

5.3.2 WRITE statement

A WRITE statement causes record(s) to be written to an external device or file, or to an internal file.

write-statement = WRITE (read-write-control) [io-list]

"Read-write-control" and "io-list" have the same syntax as in a read-statement.

If a format-specifier is present, the statement is a formatted WRITE, and execution of the statement proceeds under format control (see 5.3). If the format-specifier has no "FMT=", it must appear immediately after the unit-specifier.

If no format-specifier is present, the statement is an unformatted WRITE. In this case, an io-list must be present, and one record is written to the external device or file. The record consists of the sequence of values of the elements specified by the io-list.

The meaning of the remaining 4 options in read-write-control is as described in 5.3.1, except that the END= exit can never be taken, and should not therefore be specified, for a WRITE.

An example of a formatted WRITE is:

```
      INTEGER*2  I, J, IUNIT
      ...
      WRITE (IUNIT, 900) I, J
900   FORMAT(1X, 'Value of I = ', I5, ', and of J = ', I5/)
```

An example of an unformatted WRITE (using direct access) is:

```
      INTEGER  NUM
      DOUBLE PRECISION  DARR(100)
      CHARACTER*14 FNAME
      ...
      OPEN (56, FILE = FNAME, RECL = 800, ACCESS = 'DIRECT')
      ...
      WRITE (56, REC = NUM) DARR
```

5.3.3 PRINT statement

print-statement = PRINT format-identifier [, io-list]

This statement has precisely the same effect as the following WRITE statement:

```
      WRITE (*, format-identifier) [io-list]
```

Example:

```
      PRINT  *, 'Computed value of x is: ', X
```

5.3.4 BACKSPACE statement

A BACKSPACE statement has the effect of positioning an external file so that what had been the preceding record becomes the next record.

```
backspace-statement = < BACKSPACE unit-identifier ¦
                        BACKSPACE ( pos-control ) >
pos-control = unit-specifier
              [, IOSTAT = integer-variable-element]
              [, ERR = statement-label]
```

The BACKSPACE statement is applicable to a file open for sequential, not for direct, input or output. It may be used to backspace over an endfile record (see 5.3.5). If the unit is already at its initial position, the statement has no effect.

Example: BACKSPACE (IUNIT)

5.3.5 ENDFILE statement

The ENDFILE statement causes an output file to be marked as "at end-of-file".

```
endfile-statement = < ENDFILE unit-identifier ¦
                      ENDFILE ( pos-control ) >
```

The ENDFILE statement is applicable to a file open for sequential output - not for direct access output, nor to a file open for input. The output buffer is flushed, and a notional "endfile record" is written (no physical bytes are in fact transferred). No further output operation may be performed on the file without an intervening BACKSPACE (or REWIND) statement.

Example: ENDFILE (UNIT=IU, IOSTAT = IS)

5.3.6 REWIND statement

A REWIND statement causes the external device or file to be repositioned at its initial position.

```
rewind-statement = < REWIND unit-identifier ¦
                     REWIND ( pos-control ) >
```

The REWIND statement is applicable to a file open for sequential, not direct, input or output. In the case of output, the file's buffer is flushed. If the unit is already at its initial position, the statement has no effect.

Example: REWIND 6

5.3.7 OPEN statement

The OPEN statement permits files to be named, and also allows a fixed record length to be specified for the purpose of direct (or "random") access.

```
open-statement = OPEN ( open-control )
open-control = unit-specifier
               [, IOSTAT = integer-variable-element]
               [, RECL = integer-expression]
               [, FILE = character-expression]
               [, STATUS = character-expression]
               [, ACCESS = character-expression]
               [, FORM = character-expression]
               [, BLANK = character-expression]
               [, ERR = statement-label]
```

Only the "unit-specifier" is mandatory; zero or one of each of the other specifiers may be given.

If no FILE= parameter is present, the file is treated as a "work" file and is given a default name which is generated by the system (see 8.1.1.3).

If RECL= is specified, that defines the file to consist of fixed-length (formatted or unformatted) records, of the stated length. (See 8.1.2.3 for details.) The file must be a filestore (e.g. disc) file, and be processed using direct access (ACCESS='DIRECT').

The possible values for STATUS are: 'OLD', 'NEW', 'SCRATCH' or 'UNKNOWN'. The effect of specifying one of these is described in section 8. If no STATUS= parameter is specified, the effect is the same as STATUS='UNKNOWN'.

The possible values for ACCESS are: 'SEQUENTIAL' or 'DIRECT'. If no ACCESS parameter is specified, 'SEQUENTIAL' is assumed.

The possible values for FORM are: 'FORMATTED' or 'UNFORMATTED'. If no FORM parameter is specified, 'FORMATTED' is assumed for sequential access, and 'UNFORMATTED' in the case of direct access.

The possible values for BLANK are: 'NULL' or 'ZERO'. If BLANK='ZERO' is specified, all blanks (except leading blanks) in numeric input fields are treated as zeros; if BLANK='NULL' is specified, such blanks are ignored. If no BLANK= option is given, BLANK='NULL' is assumed. See also section 4.3.13.

In the following example, the OPEN statement defines unit 10 to be associated with a disc file called NUMBERS which consists of fixed-length records of size 128 bytes:

```
OPEN (10, FILE = 'FLP1_NUMS', RECL = 128, ACCESS = 'DIRECT')
```

5.3.8 CLOSE statement

CLOSE causes the connection (established by OPEN) between the unit and the file to be severed: to access the file again, an explicit OPEN must be performed.

```
close-statement = CLOSE ( close-control )
close-control = unit-specifier
                [, IOSTAT = integer-variable-element]
                [, STATUS = character-expression]
                [, ERR = statement-label]
```

The possible values for STATUS are: 'KEEP' or 'DELETE'. If the latter is specified, the file will be closed and then erased.

Since all open files are closed automatically by the run-time system at program termination, it is only necessary to execute a CLOSE statement if the unit is to be allocated to a different device or file.

Example:

```
CLOSE (INPUT, STATUS = 'DELETE', ERR = 999)
```

5.3.9 INQUIRE statement

The INQUIRE statement enables properties of a particular named file, or of a unit, to be made available to a program.

```
inquire-statement = INQUIRE ( inquire-control )
inquire-control = < unit-specifier ¦ FILE = character-expression >
                  [, IOSTAT = integer-variable-element]
                  [, RECL = integer-variable-element]
                  [, NEXTREC = integer-variable-element]
                  [, NUMBER = integer-variable-element]
                  [, EXIST = logical-variable-element]
                  [, OPENED = logical-variable-element]
                  [, NAMED = logical-variable-element]
                  [, NAME = character-variable-element]
                  [, ACCESS = character-variable-element]
                  [, SEQUENTIAL = character-variable-element]
                  [, DIRECT = character-variable-element]
                  [, FORM = character-variable-element]
                  [, FORMATTED = character-variable-element]
                  [, UNFORMATTED = character-variable-element]
                  [, BLANK = character-variable-element]
                  [, ERR = statement-label]
logical-variable-element = variable-element
character-variable-element = variable-element
```

Either a unit-specifier or a FILE= specifier must be present (but not both); zero or one of each of the other specifiers may be present.

The IOSTAT, RECL, NEXTREC and NUMBER specifiers require an INTEGER(*4) variable or array element to be given, which will be set to a value by the INQUIRE statement; the value represents, respectively, the input/output status (zero for OK), the record length (if a direct access file), the next record number (if a direct access file) and the unit identifier (if an external file).

The EXIST, OPENED and NAMED specifiers require a LOGICAL(*4) variable or array element to be given, which will be set to true of false by the INQUIRE statement; the value represents, respectively, whether the specified unit or file exists, whether the file or unit is connected, and whether the file has a name.

The NAME specifier requires a character variable or array element to be given, which will be set to the file name by the INQUIRE statement.

The ACCESS specifier can return the values 'SEQUENTIAL' or 'DIRECT'.

The FORM specifier can return the values 'FORMATTED' or 'UNFORMATTED'.

The BLANK specifier can return the values 'NULL' or 'ZERO'.

The remaining specifiers which require a character variable or array element to be supplied are: SEQUENTIAL, DIRECT, FORMATTED and UNFORMATTED; in each case, the INQUIRE statement can return either 'YES' or 'NO' or 'UNKNOWN'.

Examples:

```
IMPLICIT CHARACTER*20 (C), LOGICAL (L)
 ..
INQUIRE (10, IOSTAT = IO, FORM = CFORM)
INQUIRE (FILE = CFNAME, OPENED = LOPEN)
```

6 EXPRESSIONS

An expression describes the application of operators (such as +) to operands (such as variables and constants). At execution time it possesses a value, which is of a particular type.

If an expression involves several different operators, the order in which the operations should be performed is governed by the "binding strength" of the operators: operations with greatest binding strength are performed first. The following list shows operators of equal strength on the same line, with operators of greater strength on earlier lines:

Type of operator	Operator
Arithmetic	**
Arithmetic	* /
Arithmetic	+ -
Character	//
Relational	.LT. .LE. .EQ. .NE. .GE. .GT.
Logical	.NOT.
Logical	.AND.
Logical	.OR.
Logical	.EQV. .NEQV.

For example, in the expression (I + J * K), the * is performed before the +. (This accords with normal algebraic conventions.) If the order of evaluation needs to be different from that dictated by the implicit binding strengths, parentheses can be used: a parenthesised expression is always evaluated completely before performing any operation in which it is an operand. For instance, in the expression ((I + J) * K), the + is performed before the *.

Not all combinations of operators and operand types are permissible (e.g. .EQ. may not be used with logical-type operands).

There are three kinds of expression:

```
expression = arithmetic-expression ¦ logical-expression ¦
             character-expression
```

These are defined in sections 6.1 thru 6.3, respectively.

6.1 Arithmetic expressions

Expressions of this kind have values which are integer-, real-, double-precision- or complex-type.

```
arithmetic-expression = [sign] arithmetic-term
                        { <+|-> arithmetic-term }
arithmetic-term = arithmetic-factor { <*|/> arithmetic-factor }
arithmetic-factor = arithmetic-primary [ ** arithmetic-factor ]
arithmetic-primary = variable-element | function-reference |
                     arithmetic-constant | constant-name |
                     ( arithmetic-expression )
```

The first form of arithmetic-primary is a "variable-element", which is syntactic shorthand for "variable or array-element":

```
variable-element = variable-name | array-element
array-element = array-name  ( subscript {, subscript} )
subscript = integer-expression
```

The second form of arithmetic-primary is a "function-reference", which is treated in section 7.

The third and fourth forms of arithmetic-primary are literal and named constants, defined in sections 1.4.3 and 3.10 respectively.

The fifth and last form which an arithmetic-primary can take is any arithmetic expression enclosed in parentheses. In this way expressions of arbitrarily complex nested form can be written.

An arithmetic-primary is of type integer, real, double-precision or complex.

For each of the 5 arithmetic operators, an operand may be combined with another of any arithmetic type, with the exception that a double-precision and a complex operand may not be combined with one another. The result type is the same as the "senior" of the two operand types, where the arithmetic types are ordered as follows, in increasing seniority:

```
integer-type
real-type
double-precision-type
complex-type
```

Examples:

```
arithmetic-primary:       A(I)
arithmetic-factor:        A(I)**2
arithmetic-term:          A(I)**2 / B
arithmetic-expression:    A(I)**2 / B + 3.1 * G(Q) - 5.
```

6.2 Logical expressions

Expressions of this kind have logical-type values.

```
logical-expression = logical-disjunct
                     { <.EQV.|.NEQV.> logical-disjunct }
logical-disjunct = logical-term { .OR. logical-term }
logical-term = logical-factor { .AND. logical-factor }
logical-factor = [ .NOT. ] logical-primary
logical-primary = variable-element | function-reference |
                  logical-constant | constant-name |
                  relational-expression | ( logical-expression )
```

The first form of logical-primary is a "variable-element", which is described in 6.1 above.

The second form of logical-primary is a "function-reference", which is treated in section 7.

The third and fourth forms of logical-primary are literal and named constants, defined in sections 1.4.3.5 and 3.10 respectively.

The fifth form of logical-primary is a "relational-expression", which is a comparison of two arithmetic-type or two character-type expressions, and has a logical-type value: true if the expressions stand in the stated relation to one another, otherwise false.

```
relational-expression = arithmetic-relational-expression |
                        character-relational-expression
arithmetic-relational-expression = arithmetic-expression
                                   rel-op arithmetic-expression
character-relational-expression = character-expression
                                  rel-op character-expression
rel-op = .LT. | .LE. | .EQ. | .NE. | .GE. | .GT.
```

In an arithmetic-relational-expression, the two arithmetic expressions can each be of type integer, real or double-precision (in any combination); if the operator is .EQ. or .NE., then one or both of the operands may, also, be complex.

The meanings of the relational operators are:

.LT.	less than
.LE.	less than or equal to
.EQ.	equal to
.NE.	not equal to
.GE.	greater than or equal to
.GT.	greater than

The sixth and last form which a logical-primary can take is any logical expression enclosed in parentheses.

The type rules are simple: a logical-primary must be of logical-type, and the result of any of the 5 logical operations (.NOT., .AND., .OR., EQV. or .NEQV.) is also of logical-type.

Examples:

```
logical-primary:    L(I)
logical-factor:     .NOT. L(I)
logical-term:       .NOT. L(I) .AND. L2
logical-disjunct:   .NOT. L(I) .AND. L2 .OR. I .LT. J
logical-expression: .NOT. L(I) .AND. L2 .OR. I .LT. J .EQV. L2
```

6.3 Character expressions

A character expression is formed, in general, by concatenating a number of character-type "primaries", the concatenation operator being //. In the simplest case, the expression consists just of a single primary (a character constant, for example).

```
character-expression = character-primary {// character-primary}
character-primary = variable-element ¦ substring ¦
                    function-reference ¦ character-constant ¦
                    constant-name ¦ ( character-expression )
```

The first form of character-primary is a "variable-element", which is described in section 6.1 above.

The second form of character-primary is a "substring". This is a portion (which may be the whole) of a character datum, the latter being a character-type variable or array element:

```
substring = variable-element ( [substring-expression] :
                               [substring-expression] )
substring-expression = integer-expression
```

If the first substring-expression is omitted, 1 is assumed. If the second substring-expression is omitted, the length of the character variable or array-element is assumed. The first substring-expression, if present, must be less than or equal to the second one.

The third form of character-primary is a "function-reference", which is treated in section 7.

The fourth and fifth forms of character-primary are literal and named constants, defined in sections 1.4.3.6 and 3.10 respectively.

The sixth and last form which an character-primary can take is any character expression enclosed in parentheses.

7 FUNCTION REFERENCES

Function references are one of the basic ingredients of arithmetic, logical and character expressions (see 6.1, 6.2 and 6.3, respectively). A function reference consists of the function name followed by a (possibly empty) list of actual arguments enclosed in parentheses:

function-reference = function-name ([actual-argument-list])

The "actual-argument-list" is as defined for the CALL statement in 5.2.8, except that alternate return specifiers are not permitted.

Functions are of 3 kinds: statement functions, Intrinsic functions and External functions. The following subsections contain further details about referencing these 3 kinds of function.

7.1 Statement functions

Definition of statement functions was treated in section 4.1. When a statement function is referenced, the actual arguments must agree in number, order and type with the dummy arguments in the statement function's definition. Each actual argument must be an expression (see section 6) of the same type as the corresponding dummy argument.

For example, if the statement function is defined by:

```
INTEGER  K
REAL  AFUNC, X, PI
DOUBLE PRECISION  D, D1
...
AFUNC (K, X, D) = 2.0 + X * IDINT(D) ** K
```

then the right-hand-side of the following arithmetic assignment is a valid statement function reference:

```
I = AFUNC (3, PI, DSQRT(D1))
```

7.2 Intrinsic functions

Intrinsic functions have names, argument types, result types and meanings which are predefined. (They are, to some extent, like predefined statement functions.) The Intrinsic functions are enumerated in the following table.

Table of Intrinsic Functions

Generic name	Specific name	No. of args.	Arg. type	Result type	Meaning
INT	-	1	Int	Int	Conversion to INTEGER
	INT	1	Real	Int	
	IFIX	1	Real	Int	
	IDINT	1	Doub	Int	
	-	1	Comp	Int	
REAL	REAL	1	Int	Real	Conversion to REAL
	FLOAT	1	Int	Real	
	-	1	Real	Real	
	SNGL	1	Doub	Real	
	-	1	Comp	Real	
DBLE	-	1	Int	Doub	Conversion to DOUBLE PREC.
	-	1	Real	Doub	
	-	1	Doub	Doub	
	-	1	Comp	Doub	
CMPLX	-	1 or 2	Int	Comp	Conversion to COMPLEX
	-	1 or 2	Real	Comp	
	-	1 or 2	Doub	Comp	
	-	1	Comp	Comp	
-	ICHAR	1	Char	Int	Conversion to INTEGER
-	CHAR	1	Int	Char	Conversion to CHARACTER*1
AINT	AINT	1	Real	Real	Truncation
	DINT	1	Doub	Doub	
ANINT	ANINT	1	Real	Real	Nearest whole number
	DNINT	1	Doub	Doub	
NINT	NINT	1	Real	Int	Nearest integer
	IDNINT	1	Doub	Int	
ABS	IABS	1	Int	Int	Absolute value
	ABS	1	Real	Real	
	DABS	1	Doub	Doub	
	CABS	1	Comp	Real	

Table of Intrinsic Functions (ctd)

Generic name	Specific name	No. of args.	Arg. type	Result type	Meaning
MOD	MOD	2	Int	Int	arg1 - int(arg1/arg2)*arg2
	AMOD	2	Real	Real	
	DMOD	2	Doub	Doub	
SIGN	ISIGN	2	Int	Int	abs(arg1) if arg2 >= 0
	SIGN	2	Real	Real	- abs(arg1) if arg2 < 0
	DSIGN	2	Doub	Doub	
DIM	IDIM	2	Int	Int	arg1 - min(arg1,arg2)
	DIM	2	Real	Real	
	DDIM	2	Doub	Doub	
-	DPROD	2	Real	Doub	Double precision product
MAX	MAX0	>=2	Int	Int	Largest value
	AMAX1	>=2	Real	Real	
	DMAX1	>=2	Doub	Doub	
-	AMAX0	>=2	Int	Real	Largest value
	MAX1	>=2	Real	Int	
MIN	MIN0	>=2	Int	Int	Smallest value
	AMIN1	>=2	Real	Real	
	DMIN1	>=2	Doub	Doub	
-	AMIN0	>=2	Int	Real	Smallest value
	MIN1	>=2	Real	Int	
-	LEN	1	Char	Int	Length of string
-	INDEX	2	Char	Int	Location of arg2 in arg1
-	AIMAG	1	Comp	Real	Imaginary part
-	CONJG	1	Comp	Comp	Complex conjugate
SQRT	SQRT	1	Real	Real	Square root
	DSQRT	1	Doub	Doub	
	CSQRT	1	Comp	Comp	
EXP	EXP	1	Real	Real	Exponential
	DEXP	1	Doub	Doub	
	CEXP	1	Comp	Comp	
LOG	ALOG	1	Real	Real	Natural logarithm
	DLOG	1	Doub	Doub	
	CLOG	1	Comp	Comp	

Table of Intrinsic Functions (ctd)

Generic name	Specific name	No. of args.	Arg. type	Result type	Meaning
LOG10	ALOG10	1	Real	Real	Common logarithm
	DLOG10	1	Doub	Doub	
SIN	SIN	1	Real	Real	Sine
	DSIN	1	Doub	Doub	
	CSIN	1	Comp	Comp	
COS	COS	1	Real	Real	Cosine
	DCOS	1	Doub	Doub	
	CCOS	1	Comp	Comp	
TAN	TAN	1	Real	Real	Tangent
	DTAN	1	Doub	Doub	
ASIN	ASIN	1	Real	Real	Arcsine, with result in
	DASIN	1	Doub	Doub	range -pi/2 to +pi/2
ACOS	ACOS	1	Real	Real	Arccosine, with result in
	DACOS	1	Doub	Doub	range 0 to pi
ATAN	ATAN	1	Real	Real	Arctangent, with result in
	DATAN	1	Doub	Doub	range -pi/2 to +pi/2
ATAN2	ATAN2	2	Real	Real	Arctangent(arg1/arg2), with
	DATAN2	2	Doub	Doub	result in range -pi to +pi
SINH	SINH	1	Real	Real	Hyperbolic sine
	DSINH	1	Doub	Doub	
COSH	COSH	1	Real	Real	Hyperbolic cosine
	DCOSH	1	Doub	Doub	
TANH	TANH	1	Real	Real	Hyperbolic tangent
	DTANH	1	Doub	Doub	
-	LGE	2	Char	Log	arg1 >= arg2
-	LGT	2	Char	Log	arg1 > arg2
-	LLE	2	Char	Log	arg1 <= arg2
-	LLT	2	Char	Log	arg1 < arg2

(In the "type" columns, the following abbreviations have been employed: Int for INTEGER, Doub for DOUBLE PRECISION, Comp for COMPLEX, Log for LOGICAL and Char for CHARACTER.)

The actual argument(s) in an intrinsic function call must agree in number and type with the specification in the above table, and may be any expressions of the stated type. It is an error to supply a second argument of zero to the functions AMOD, MOD or DMOD; or, more generally, to supply an argument to a transcendental function for which the result is not mathematically defined (for example: a zero argument to LOG).

A function reference is classified as an intrinsic function reference if the name of the function is one of the names in the above table, and is not an array name, statement function name, subroutine name or dummy argument name. Note that IMPLICIT statements have no effect on the result-types of intrinsic functions.

If a specific Intrinsic function name is passed as an actual (procedure-name-type) argument in a CALL statement or function reference, then that name must appear in an INTRINSIC statement (see 3.9) in the calling program unit. The following specific Intrinsic names may not be passed as actual arguments: INT, IFIX, IDINT, REAL, FLOAT, SNGL, ICHAR, CHAR, MAXO, AMAX1, DMAX1, AMAXO, MAX1, MINO, AMIN1, DMIN1, AMINO, MIN1, LGE, LGT, LLE, LLT. A generic Intrinsic name may not be passed as an actual argument.

Examples of Intrinsic function references (assuming all variables to have their default types):

```
I = MAX1(X/Y, 1.0) + MOD(J, 10)
X = DIM( ABS(Z), FLOAT(K))
```

7.3 External functions

Function references which are not statement function or intrinsic function references are classified as external function references. The reference is either to a Fortran function subprogram of that name, or to an external module coded in some other language but presenting the same interface. In the former case, the actual arguments must agree in number, order, kind and type with the dummy arguments in the FUNCTION (or ENTRY) statement at the start of the subprogram. The remarks in section 5.2.8 about agreement in "kind" and "type" apply here, too.

The permissible kinds of actual argument are as defined for the CALL statement in 5.2.8, except that alternate return specifiers are not allowed.

Examples:

```
X = F(Y-Z)
J = IFUNC( )
```

8 IMPLEMENTATION-DEPENDENT ASPECTS

8.1 Fortran-77 files and QDOS

8.1.1 Files and records

A Fortran-77 file is composed of "records" arranged in sequence, just one record being accessible at any one time. A record may be

formatted or unformatted,
variable-length or fixed-length.

Formatted and unformatted records cannot be mixed in the same file, nor can variable-length and fixed-length. Formatted records are written and read by the formatted forms of the WRITE and READ statements, and unformatted records by the unformatted forms.

In the case of files allocated on external media such as a disc, variable-length records can only be processed sequentially, and fixed-length records can only be processed using direct access, i.e. with ACCESS='DIRECT' in the OPEN statement and using the direct-access forms (REC = ...) of READ and WRITE statements. In the case of "internal" files, however, which are allocated to a program's data, the records can only be fixed-length and accessed sequentially.

In this implementation, fixed- and variable-length formatted records are limited in size to a maximum of 200 characters. Fixed-length unformatted records are limited in size to a maximum of 32767 bytes. Variable-length unformatted records can be arbitrarily long.

Fortran files may be classified into the following types:

pre-connected
external
work
internal.

8.1.1.1 Pre-connected files

Such files, referenced by a program in READ, WRITE and PRINT statements specifying (or, in the case of PRINT, implying) UNIT=*, must be used with formatted I/O and accessed sequentially. No other types of I/O statement may be used to reference them. In particular, no OPEN statement is needed. Formatted numerical input fields are processed by default as if the OPEN option BLANK='NULL' applied.

Pre-connected files are named and opened automatically before program execution begins. The default pre-connected files are associated with the program's window, and this will be used for UNIT=* references if no explicit files are specified by the user for pre-connection (see part III, section 4.2).

8.1.1.2 External files

Except for the pre-connected files (see above), External files require an OPEN statement with a unit-identifier which is an integer expression. If no FILE= parameter is supplied, a work file is implied (see 8.1.1.3); otherwise, a file name of up to 36 characters must be given. This may represent the name of a filestore-type file, such as one on disc or microdrive, or a device-type file such as SER, SER1, etc. Files to be accessed by direct-access operations must be allocated to devices that support such operations.

8.1.1.3 Work files

If an OPEN statement has no FILE= parameter, a workfile is implied, with a name which will be chosen by the library software at run time. For such files, STATUS cannot be specified as 'OLD' or 'NEW'.

All workfiles used by a program at run-time are allocated to the same device, which is defaulted but can be configured by the user (see part III, section 5). In particular, the use may choose the device at run-time. Workfiles are deleted automatically by a CLOSE statement or when the program terminates.

Workfile names chosen by the library software have the following ASCII character format:

<device>_F77$_<job id>_<sequence no>

where <device> is the chosen/default device name, e.g. MDV2, <job id> is the QDOS job id for the program (8 hex characters), <sequence no> is a unique sequence number, starting at 0001 and incrementing by 1 for each workfile opened (4 hex characters).

8.1.1.4 Internal files

Internal files are distinguished from External files by having unit-identifiers which are array-names or character-fields (see 5.3). They are memory-allocated "files" containing fixed-length formatted records. READ, WRITE and PRINT are the only I/O statements which may be used with internal files. List-directed input/output (see 5.3) is not permitted. Only sequential access is allowed. The maximum record length possible with an internal file depends entirely on the declaration of the character variable or array used.

8.1.1.5 File existence

This concept is largely involved with the INQUIRE and OPEN statements. A closed file will be said to exist if an open-input operation on the file succeeds; otherwise, the file will be said to "not exist". If the file is already open, then it must exist.

8.1.1.6 File access

Both sequential and direct access are supported for all filestore type files, and sequential access for device-type files. Files for sequential access consist of variable-length records, files for direct access consist of fixed-length records.

It is the user's responsibility to only apply those I/O operations to a file or device that it is capable of accepting.

8.1.2 File formats

Fixed length records and variable length unformatted records can only exist on filestore devices. Variable length formatted records can also be written to a printer or the screen, and read from the keyboard.

8.1.2.1 Variable-length formatted records

A Fortran program typically uses a file of variable-length formatted records for input of data or display of results. In this kind of file, each "record" becomes one line of input or output. On filestore, the file layout follows the conventions of the operating system for text files.

When reading such records from a filestore (e.g. disc) file, end-of-file will be set either by encountering the physical end of the file or by a line consisting of <CTRL-Z> (decimal code 26) followed by <LINEFEED> (decimal code 10).

To signal end-of-file when inputting from the keyboard, a separate line consisting of <CTRL-SHIFT-QUOTE> followed by <ENTER> must be supplied, i.e. decimal codes 130 followed by 10.

On output, the first character of the record is interpreted to determine printer carriage-control according to the standard Fortran conventions:

Blank	Advance one line
0	Advance two lines (double-spacing)
1	Advance to top of page
+	No advance

Any other character is treated as blank, i.e. producing single spacing. In all cases the first character does not appear in the actual output. If such a file is produced by list-directed output, each line of the file will begin with a blank character to give single-spacing when printed.

Note that this "interpretation" is performed whether the records are being output to filestore or to an actual printer. Thus a file intended for printing may either be printed directly, or first output to filestore and then spooled or copied to a printer device.

Note also that the control characters produced as a result of carriage-control interpretation are output at the beginning of the corresponding line, i.e. before outputting the remaining characters of the line.

8.1.2.2 Variable-length unformatted records

A file of variable-length unformatted records must be a filestore file. Each Fortran "record" is divided into logical records, consisting of 2 control bytes and user data. It follows that small records may be associated with a relatively large amount of control information. In all cases, however, filestore is used as efficiently as possible; and there is effectively no limit on the size of the Fortran records.

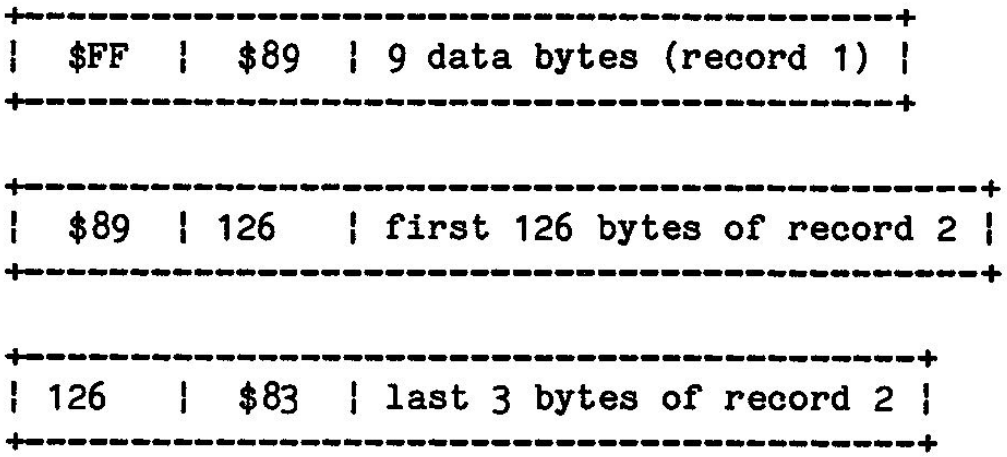
Each Fortran record consists of one or more contiguous logical records not exceeding 128 bytes in size, constructed as follows:

where C1 and C2 are control bytes, and the user data does not exceed 126 bytes (it can be of zero length).

C1: equal to $FF if this is the start of the file; otherwise it contains the number of user data bytes in the previous logical record, with bit 7 set if this is the first logical record of the Fortran record.

C2: bit 7 is set if this is the last logical record of the Fortran record, otherwise it is zero; bits 6 to 0 contain the length of the user data (0 to 126 inclusive).

For example, a file consisting of 2 Fortran records with 9 and 129 bytes of data respectively would consist of the following sequence of 144 bytes:



8.1.2.3 Fixed-length records

Fixed-length records are restricted to filestore files, and are limited in size to 200 characters (formatted) and 32767 bytes (unformatted). The Fortran records are stored adjacent to one another, with no control information or "slack" bytes. The RECL= value used for reading such a file must be such that the file size is an exact multiple of the record size.

The first character of fixed-length formatted records is not interpreted as a printer control character.

8.1.3 Unit numbers

A unit number is associated with a file or device, and is the means by which a program refers to that file. Unit numbers can be in the range 0 to 255 inclusive. The OPEN statement (see 5.3.7) allows any valid unit number to be connected to a named filestore file (e.g. MDV1_DATAFILE) or to a device (e.g. CON_20x50 or SER1).

At most 15 units (excluding the 2 pre-connected units) may be open at one time.

8.1.4 Random access

Random access files are always OPENed with ACCESS='DIRECT' and RECL= giving the record size in characters or bytes. Section 8.1.2.3 gives the maximum allowed record sizes.

If the OPEN statement specifies STATUS='OLD', the file must exist, and it is possible to update the file by a mixture of READ and WRITE operations quoting REC= to address the records directly.

If the OPEN statement specifies STATUS='SCRATCH', a workfile with a library-chosen name (see section 8.1.1.3) will be created if it does not already exist or overwritten if it does. If, however, STATUS='NEW', the specified file must not already exist. Then, in either case, the first operation on the file must be WRITE.

It is quite permissible to write records to the file in any desired order using the REC= parameter, but it is the user's responsibility to ensure that no record is read before it has been written. If by the time the file is CLOSEd an incomplete set of records has been written by the program, all unwritten records will actually exist in the file but they will not, of course, contain any user data. They will in fact contain binary zero bytes or ASCII blank characters according as the records are unformatted or formatted respectively.

8.1.5 Operations on External and Work files

These are the operations of opening, closing and deleting files on filestore media.

8.1.5.1 Opening files

An OPEN statement associating a unit number with a file must be executed before any data transfer I/O statement refers to that unit. OPEN may be used to effect one of the following operations:

(a) Associate the unit number with an already existing file in preparation for formatted or unformatted, direct or sequential, I/O. The file may be read, updated or completely overwritten prior to a REWIND or CLOSE on the same unit.

(b) Associate the unit number with a file that is to be created, i.e. that does not already exist, in preparation for formatted or unformatted, direct or sequential, I/O. The file may be written or updated prior to a REWIND or CLOSE on the same unit.

(c) Perform either of operations (a) or (b) when the unit number is currently connected to another file. The latter file is first closed with an internal CLOSE(unit) operation, and then the new association is made as before.

(d) Alter the BLANK= specifier for the file already connected to the unit number.

It is important to realize that an OPEN statement does not cause a physical file-open operation. The latter takes place when the first data transfer I/O statement for the associated unit is executed. For example, consider the fragment:

```
.
.
OPEN (u,FILE='MDV2_DATA_IN',STATUS='OLD',ERR=100)
READ (u,*,ERR=200)
.
.
```

Then if the file DATA_IN does not exist on the cartridge in MDV2 at the time these statements are executed, the statement with label 200 will receive control rather than the statement with label 100.

In the sense of the Fortran-77 Standard, the term "connected" when applied to a file means that a successful file-open operation has been performed.

The following table shows the open modes used for various OPEN options:

Status	Open mode
'NEW'	New exclusive
'SCRATCH'	New overwrite
'OLD' (Input)	Old exclusive
'OLD' (Output)	New overwrite
'OLD' (Direct)	Old exclusive

The effect of omitting STATUS= in the OPEN statement is the same as specifying STATUS='UNKNOWN', and causes the status to be set at run time according to the following rules:

(a) If FILE= is omitted, then STATUS is set to 'SCRATCH'.

(b) If FILE= is supplied, then if the file exists STATUS is set to 'OLD' else to 'NEW'.

To send a formatted output file directly to the printer, the OPEN statement must specify STATUS='NEW', e.g:

```
OPEN (iprin, File = 'SER1', Status = 'NEW')
```

An alternative method is to write to unit "*", and reply SER or SER1 to the "Standard output file?" question at program start-up (see Part III, section 4.2).

8.1.5.2 Closing files

The CLOSE statement breaks the connection between a unit number and the associated file, so that the unit number may no longer be used to access that file, but may, if desired, be associated with another file by means of an OPEN statement.

8.1.5.3 Deleting files

There is no single Fortran-77 statement that will cause a particular file to be deleted, but the following sequence may be used:

```
OPEN (u,FILE='filename',STATUS='OLD')
CLOSE (u,STATUS='DELETE')
```

This sequence will not cause an error if the file does not exist.

8.1.6 Input/output restrictions

There is a general restriction on the execution of I/O statements, namely, that no function reference made during an I/O statement may cause another I/O statement to be executed.

8.2 Additional library routines

There are a number of routines in the Fortran library which can be called on when required. Use in a program causes the appropriate selection to be made during the link-edit process.

8.2.1 GETCOM (options)

The subroutine GETCOM may be called at any time during a program, to obtain the program option string.

The parameter "options" must be a character variable or array element, and the program option string will be copied to this as if an assignment to the character variable was being made. Thus if the option string is shorter than the receiving variable, the latter is filled out with blanks, but if the option string is longer, truncation occurs. If there is no program option string, the receiving variable will be set to all blanks.

```
Example:        CHARACTER SHORT*6, LONG*16
                ..
                CALL GETCOM (SHORT)
                CALL GETCOM (LONG)
```

If the program option string is ' Weather Report', then the variables SHORT and LONG will be set to ' Weath' and ' Weather Report '.

8.2.2 RANDOM (i)

The function RANDOM yields at each call a pseudo-random real value, uniformly distributed in the range 0.0 to 1.0. The parameter i must be an INTEGER*4 variable or expression. If (i.LE.0) the next value in the pseudo-random sequence is returned; if (i.GT.0) then the value of i is taken as a new "seed", and another sequence started.

```
Example:   X = RANDOM (0)
```

8.2.3 IADDR (variable)

This function has as argument any variable reference, and returns an integer result, being the 32-bit absolute address of the variable.

A particular use for this function is in supplying address-type parameters to some of the QDOS function calls (cf. 8.2.12).

```
Example:        IV = IADDR(V(I,J))
```

8.2.4 IPEEK (iaddr)

The INTEGER*1 function IPEEK returns the value of the byte at memory location iaddr. The parameter iaddr must be an INTEGER*4 variable or expression, representing an absolute machine address.

```
Example:    INTEGER*1  IPEEK
            ...
            IVAL = IPEEK (128+J)
```

8.2.5 POKE (iaddr,ival)

Subroutine POKE stores ival (truncated to one byte if necessary) at memory location iaddr. Both parameters are INTEGER*4 variables or expressions, with iaddr representing an absolute machine address.

```
Example:    CALL POKE (192+J,IVAL)
```

8.2.6 EXECPG (command_string,return_code)

EXECPG allows a user program to dynamically execute other separately linked Fortran-77 programs.

The currently executing Fortran program, the "parent", uses EXECPG to execute a separately linked Fortran program, the "child". A string, the "program options", can be specified for passing to the child program, which can obtain the string by means of subroutine GETCOM (see 8.2.1).

While the child program runs, the parent program is suspended, and only continues when the child terminates. The child itself may cause child programs of its own to be executed in the same way. The parent program and each child program run by it are separate QDOS jobs, these jobs together forming a dependent job tree.

When the child terminates, a numeric return code is passed back to its parent (see 8.2.7).

EXECPG requires the following parameters -

(a) A character expression "command_string" in the form

```
'<child_program_file>[<program_options>]'
```

Examples:

1. The command string 'MDV1_MENU' specifies program MENU on MDV1, and a null program options string is passed.

2. The command string 'MDV2_PRINT_BIN MYFILE_DAT' specifies that program PRINT_BIN on MDV2 is to be executed, and the options string ' MYFILE_DAT' is to be made available to the program (which uses subroutine GETCOM to get this string). Note that the space separating the options string from the program name is part of the string.

(b) An INTEGER*4 variable "return_code" that will receive the child's return code when it terminates (see 8.2.7).

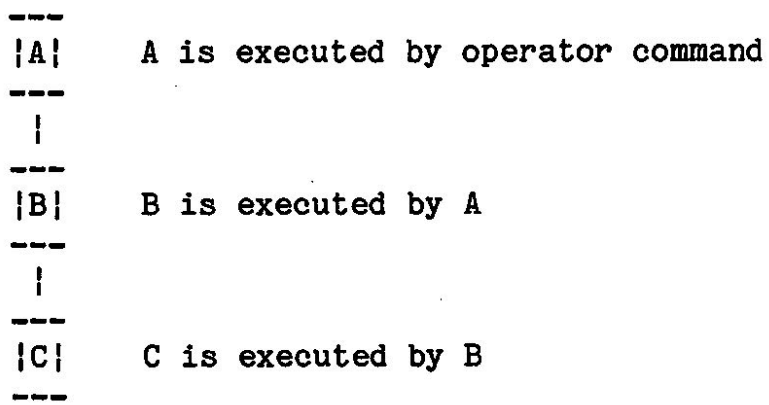
Example:

A parent program could include the following statements -

```
.
.
INTEGER*4 rc
.
.
CALL  EXECPG ('MDV2_HOME PARK',rc)
IF (rc) 70, 80, 90
.
```

The statements cause program HOME to be loaded from MDV2 and executed. A call of GETCOM in HOME will yield the 5-character string ' PARK'. When HOME terminates, the return code it sets is stored in variable rc, which the parent then tests. Normally, a negative return code will indicate an error condition (see 8.2.7).

A hierarchy of programs can be executed in this manner, as in the following example.



Program A is loaded and executed by an operator command, and uses EXECPG to load and execute B. B, in turn, uses EXECPG to load and execute C. Thus when C has control, all three programs are in memory. A and B are known as parent and child programs respectively, as are B and C. Each of A, B and C is a separate QDOS job with B dependent on A and C dependent on B.

When C terminates, it is deleted from memory and control returns to B, which then continues execution. When B terminates, it is deleted from memory and control returns to A, which then continues execution. When A terminates, control passes back to the operating system in the usual way.

This example demonstrates that at each level of the program hierarchy, there is just one program loaded. Thus suppose for example that B executes further programs D and E after C, then D and E are successively loaded, executed and deleted from memory.

The user is of course limited by the amount of memory available for user programs, and to make maximum use of this the Linker option (see Part III) for specifying stack/work area size should be used when linking programs to be dynamically executed.

A parent program's standard input/output files (even if not yet accessed) are also available to each of its child programs, through any number of levels, via UNIT = *.

8.2.7 EXITPG (return_code)

EXITPG may be used by a program to terminate its execution at any time, and to pass a non-negative return code back to the calling parent program (if any).

If a program terminates normally without calling EXITPG, a return code of zero will be passed back. Sometimes, the run-time library will detect an error condition and force termination of a child program with one of the negative return codes below.

The return_code given to EXITPG can be any integer expression and should be in the range 0 to 127 inclusive. The library-generated negative return codes that can be generated are as follows:-

-1	child terminated by run-time error
-2	error in format of command string
-3	child program file could not be opened, or error reading child program's file header
-4	child program file not marked as executable
-5	failure to load child program file
-6	child program incompatible with PRL software
-7	insufficient memory to run child program
-8	error during initialization of child program
-9	no PRL installed, or PRL is corrupt
-10	error opening/reading/writing child program's window
-11	only one channel passed to child program
-12	error when linking child program
-13	unable to create a job to run child program

For an explanation of run-time errors, see Appendix C; for further details of the other errors listed above, see Part III section 4.4

Example:

```
CALL EXITPG (8)
```

causes the issuing program to terminate with a return code of 8.

8.2.8 AFFIRM (prompt)

AFFIRM is a LOGICAL function which is given the argument "prompt", a character expression intended for display on the screen. The user must respond with either Y (or y) for an affirmative response or N (or n) for a negative response, and the function then returns the corresponding value .TRUE. or .FALSE. respectively.

Example:

```
LOGICAL AFFIRM
.
.
IF (AFFIRM ('More data files')) THEN
  .
  .
ENDIF
.
.
```

The effect of these statements is to display "More data files? " on the screen (the "? " is appended automatically) and flash the cursor to invite input. The reply is echoed on the screen and the function value returned as described above. If an invalid reply is entered, the cursor continues to flash until a valid reply is given.

8.2.9 IHANDL (unit_number)

IHANDL is an INTEGER*4 function which is given a unit number expression as argument. If the unit number is connected to a file which has been physically opened, i.e. at least one READ, WRITE or PRINT statement for the unit has been successfully executed, the function will return the 32-bit QDOS channel ID for the unit in question; otherwise, it returns the value -1.

Example: `IC = IHANDL (10)`

8.2.10 DATE (year,month,day)

Subroutine DATE puts the current date into the 3 arguments, which must be INTEGER*4 variables (or array elements).

Example: CALL DATE(IYEAR,MONTH,IDAY)

8.2.11 TIME (hours,mins,seconds,hundredths)

Subroutine TIME puts the current time into the 4 arguments, which must be INTEGER*4 variables (or array elements).

Example: CALL TIME(IHOURS,MINS,ISECS,IHUNDS)

8.2.12 TRAP (trap_no,regarray)

This subroutine enables QDOS trap calls to be made from a Fortran program. The first parameter is an integer expression which is truncated to 4 bits to derive the trap number in the range 0-15. The second parameter is the name of an INTEGER*4 array having the layout given by the supplied INCLUDE file TRAPREG_FOR. The user is responsible for setting all necessary parameters required by the QDOS call to be issued including, of course, the call number. The subroutine loads the machine registers from regarray before issuing the TRAP machine instruction. On return, the machine registers are stored into regarray where they can be examined by the user program.

Example:

```
*       Read the display mode (TRAP $1, call 10 hex)

        D0 = 16
        D1 = -1
        D2 = -1
        CALL TRAP (1,MCREGS)

*  The display mode and display type are returned in the l.s. 8 bits
*  of registers D1 and D2 respectively. Normally, at this point one
*  would test the contents of D0 to detect any error condition.
```

Note: Care should be taken that the QDOS trap call does not interfere with the operation of the Fortran run-time routines, in particular those concerning files.

8.2.13 MODE (highres)

Subroutine MODE sets the QL screen to either high resolution 4-colour mode or low-resolution 8-colour mode. The argument must be a logical expression or a LOGICAL*4 variable or array element. For example, the following statement will set the display to 4-colour mode:

```
        CALL MODE(.TRUE.)
```

8.2.14 Window routines

This group of subroutines enables the programmer to associate a screen output window with a Fortran unit number, and to set and inquire about its attributes. Where there is a related SuperBASIC command, this is stated, and the QDOS documentation for that command should be consulted for more details.

A program may create several windows, but there is an overall limit of 15 on the number of units (windows and other Fortran files) which may be open simultaneously (see 8.1.3).

All the routines in this section and in sections 8.2.15 thru 8.2.17 operate on "window"s. Note that the window must be specified explicitly in each subroutine call - there is no equivalent of the SuperBASIC idea of the "default channel".

Before calling any other subroutine referencing a window, the window must first be opened, i.e. associated with a Fortran unit number, by a call of WOPEN.

It is possible to output text to a window using formatted WRITE stamements, but since the window is for output only, READ statements are impermissible.

8.2.14.1 WOPEN (unit, width, height, Xorigin, Yorigin)

Related SuperBASIC keyword: OPEN

Opens a screen output window. All five arguments must be integer expressions. The last four arguments are in pixel units. For example, the call

```
CALL wopen (10, 448, 180, 32, 16)
```

associates unit 10 with the QL console device SCR_448x180a32x16 (which is in fact the default SCR_ device).

8.2.14.2 WINDOW (unit, width, height, Xorigin, Yorigin)

Related SuperBASIC keyword: WINDOW

Allows the user to change the size and/or position of a window. All five arguments must be integer expressions, the last four representing pixel values.

Example: `CALL WINDOW (iwind, 30, 40, 10, 10)`

8.2.14.3 WSTATC (unit, width, height, Xcursor, Ycursor)

Related SuperBASIC keyword: none

An inquiry subroutine, which returns the window size and cursor position, both in terms of character coordinates. The last four arguments must be INTEGER*4 variables/array-elements. Note that the top left cursor position is 0,0.

Example: CALL wstatc(10, iw, ih, ix, iy)

8.2.14.4 WSTATP (unit, width, height, Xcursor, Ycursor)

Related SuperBASIC keyword: none

An inquiry subroutine, which returns the window size and cursor position, both in terms of pixel coordinates. The last four arguments must be INTEGER*4 variables/array-elements. Note that the top left cursor position is 0,0.

Example: CALL wstatc(iwind, iw, ih, ix, iy)

8.2.14.5 RECOL (unit, c1, c2, c3, c4, c5, c6, c7, c8)

Related SuperBASIC keyword: RECOL

Changes the colour values for this window. Each of the last 8 arguments must be integer expressions in the range 0 (= black) thru 7 (= white).

Example: CALL recol (iwind, 2,3,4,5,6,7,1,0)

8.2.14.6 BORDER (unit, width, colour)

Related SuperBASIC keyword: BORDER

Adds to the window a border of the specified size and colour. The arguments must be integer expressions. "width" is in pixels. "colour" is in the range 0 to 255; considered as an 8-bit byte, the bottom three bits define the base colour (0 = black thru 7 = white), the next three bits give the exclusive OR of this base colour and the stippling colour (if these three bits are zero, the colour is therefore solid), and the top two bits select one of the four stipple patterns. The special value $80 for "colour" will produce a "transparent" border, i.e. the previous border is unchanged.

Example: CALL BORDER (10, I1-I2, $40+J*NEWCOL)

8.2.14.7 INK (unit, colour)

Related SuperBASIC keyword: INK

Sets the ink colour for the given window. Both arguments must be integer expressions. The second argument should be in the range 0 to 255, the associated colour being as described in 8.2.14.6.

Example: call ink (iw3, icolor[j])

8.2.14.8 PAPER (unit, colour)

Related SuperBASIC keyword: PAPER

Sets the paper colour for the given window. Both arguments must be integer expressions. The second argument should be in the range 0 to 255, the associated colour being as described in 8.2.14.6.

Example: call paper (10, icol)

8.2.14.9 STRIP (unit, colour)

Related SuperBASIC keyword: STRIP

Sets the strip colour for the given window. Both arguments must be integer expressions. The second argument should be in the range 0 to 255, the associated colour being as described in 8.2.14.6.

Example: call strip (10, 7-icol)

8.2.14.10 BLOCK (unit, width, height, Xorigin, Yorigin, colour)

Related SuperBASIC keyword: BLOCK

Fills a block of the specified size and position with the given colour. All arguments must be integer expressions. The middle four arguments are in pixel units. The meaning of "colour" is as described in 8.2.14.6.

Example: call block (iwind, 10,10, 5,5, 7)

8.2.14.11 CLS (unit, part)

Related SuperBASIC keyword: CLS

Clears the window to the current PAPER colour. Both arguments are integer expressions. The second argument determines how much of the window area will be cleared, according to the scheme:

part = 0	whole window
part = 1	top excluding the cursor line
part = 2	bottom excluding the cursor line
part = 3	whole of the cursor line
part = 4	right end of cursor line (including cursor)

Example: CALL CLS (10, 0)

8.2.14.12 PAN (unit, distance, part)

Related SuperBASIC keyword: PAN

Pans the window "distance" pixels to the right; if distance is negative, pans to the left. All arguments are integer expressions. The last argument determines how much of the window area will be panned, according to the scheme:

part = 0	whole window
part = 3	whole of the cursor line
part = 4	right end of cursor line (including cursor)

Example: CALL PAN (10, IXNEW-IXOLD, 4)

8.2.14.13 SCROLL (unit, distance, part)

Related SuperBASIC keyword: SCROLL

Scrolls the window "distance" pixels downwards; if distance is negative, scrolls upwards. All arguments are integer expressions. The last argument determines how much of the window area will be scrolled, according to the scheme:

part = 0	whole window
part = 1	top excluding the cursor line
part = 2	bottom excluding the cursor line

Example: CALL SCROLL (10, IYNEW-IYOLD, 0)

8.2.15 Print style routines

This group of subroutines enables the programmer to influence the style in which text is output to a window. In each case, there is an associated SuperBASIC command, and the QDOS documentation should be consulted for further details.

8.2.15.1 CSIZE (unit, width, height)

Related SuperBASIC keyword: CSIZE

Sets the character size for the window. All arguments are integer expressions. "Width" should be in the range 0 to 3, and "height" should be 0 or 1.

Example: call csize (iwind, 0, 0)

8.2.15.2 FLASH (unit, onoff)

Related SuperBASIC keyword: FLASH

Sets the FLASH state on or off. The second argument must be a logical expression; if it is true, the flash state is set on, else off.

Example: CALL FLASH (10, .TRUE.)

8.2.15.3 UNDER (unit, onoff)

Related SuperBASIC keyword: UNDER

Turns underlining on or off. The second argument must be a logical expression; if it is true, underline is set on, else off.

Example: CALL UNDER (10, .NOT. L)

8.2.15.4 OVER (unit, mode)

Related SuperBASIC keyword: OVER

Both arguments are integer expressions. The second argument selects the type of over-printing required, according to the scheme:

```
mode = -1       print in INK over previous screen contents
mode = 0        print INK on STRIP
mode = 1        print in INK on transparent STRIP
```

Example: call over (10, 1)

8.2.16 Cursor positioning routines

This group of subroutines enables the programmer to position the cursor within a window. In each case, there is an associated SuperBASIC command, and the QDOS documentation should be consulted for further details.

8.2.16.1 ATC (unit, line, column)

Related SuperBASIC keyword: AT

Positions the cursor in the window using character coordinates, with 0,0 corresponding to the top left corner of the window. All arguments are integer expressions.

Example: CALL ATC (iwind, 0, 0)

8.2.16.2 ATP (unit, Xpos, Ypos)

Related SuperBASIC keyword: CURSOR (with 2 parameters)

Positions the cursor in the window using pixel coordinates, with 0,0 corresponding to the top left corner of the window. All arguments are integer expressions.

Example: CALL ATP (iwind, 20, 30)

8.2.16.3 ATG (unit, Xorigin, Yorigin, Xoffset, Yoffset)

Related SuperBASIC keyword: CURSOR (with 4 parameters)

Positions the cursor in the window, using a combination of graphics coordinates (the second and third arguments, which must be real expressions) and pixel offsets (the fourth and fifth arguments, which must be integer expressions).

Example: CALL ATG (iwind, gx, gy, 0, 0)

8.2.17 Graphics drawing routines

This group of subroutines enables the programmer to output points, lines and arcs to a window. The facilities provided correspond to those of the SuperBASIC graphics procedures, and in each case the corresponding SuperBASIC keyword is given. The QDOS documentation should be consulted for further details.

8.2.17.1 FILL (unit, onoff)

Related SuperBASIC keyword: FILL

Turns the graphics fill on or off. The second argument is a logical expression; if it is true, fill is set on, else off.

Example: call fill (10, .TRUE.)

8.2.17.2 SCALE (unit, scalefactor, Xorigin, Yorigin)

Related SuperBASIC keyword: SCALE

Allows the scale factor used by the graphics procedures (POINT, LINE, ARC, CIRCLE and ELIPSE) to be altered. The last 3 arguments are real expressions.

Example: call scale (10, 0.5, 0.1, 0.1)

8.2.17.3 POINT (unit, X, Y)

Related SuperBASIC keyword: POINT

Plot a point at the specified position relative to the graphics origin. The last two arguments are real expressions.

Example: call plot(10, x, 0.0)

8.2.17.4 LINE (unit, Xfrom, Yfrom, Xto, Yto)

Related SuperBASIC keyword: LINE

Draw a straight line between two points, whose locations are specified in absolute graphics coordinates. The last four arguments are real expressions.

Example: call line (iwind, 0.0, 0.0, xdest, ydest)

8.2.17.5 ARC (unit, Xfrom, Yfrom, Xto, Yto, angle)

Related SuperBASIC keyword: ARC

Draw an arc of a circle between two points, whose locations are specified in absolute graphics coordinates. The last five arguments are real expressions. The last argument gives the angle subtended by the arc, in radians. The following will draw a semi-circle, for example:

```
CALL ARC (10, XL, YL, XR, YR, 3.141593)
```

8.2.17.6 CIRCLE (unit, Xcentre, Ycentre, radius)

Related SuperBASIC keyword: CIRCLE (with 3 parameters)

Draw a circle whose centre and radius are given in graphics coordinates. The last three arguments are real expressions.

Example: CALL CIRCLE (10, XCEN, YCEN, SIN(GRAD))

8.2.17.7 ELIPSE (unit,Xcentre,Ycentre,majoraxis,eccentricity,angle)

Related SuperBASIC keyword: CIRCLE (with 5 parameters)

Draw an ellipse whose centre and major axis are given in graphics coordinates, with a specified eccentricity and orientation. The last five arguments are real expressions. The "eccentricity" is the ratio between the major and minor axis. The "angle" is the orientation of the major axis relative to the vertical, in radians.

Example: call elipse (10, xcen, ycen, 12.0, 0.5, 0.0)

8.3 Storage allocation

8.3.1 Overall layout

Object programs can in general contain requirements for the following kinds of storage.

Program code.
Constants (literals).
Static data areas.
COMMON data blocks.
Stack/work area.

All program variables, whether in COMMON or not, are allocated static data space. Thus any SAVE statements in the program source are effectively redundant.

In the object code from the compiler, the static data for each module, and each COMMON block, is located on a word boundary. The stack is kept word-aligned throughout execution of the program.

The result of compiling a program unit is a relocatable module in Sinclair object format which consists of a number of "sections". There are up to four sections generated:-

(1) .CODE (if the program unit contains executable statements), which contains object code and constants (integer, real, character, etc.). The code generated per program unit cannot exceed 32K bytes.

(2) .ATAB, which contains control information enabling run-time access to COMMON blocks and linked-in routines such as other program units and library procedures

(3) .INIT, which contains control information enabling the run-time initialization of COMMON blocks and local variables (corresponding to DATA statements in the source program).

(4) .NAME, which by default contains just the main program's name. If the /N compile-time option is selected, it also contains the names of files and subroutines compiled, for utilization in the event of a run-time error, when producing a subroutine call trace-back.

Each COMMON block is converted into a COMMON section with the same name as the common block, the name .BLANK being used for blank common. A program unit's local static data also becomes a COMMON section, but is only addressable by that program unit. Note however that a program file contains no COMMON areas within it, since the COMMON DUMMY directive is used at link-time. The actual COMMON areas used by a program are created and initialised dynamically at run-time.

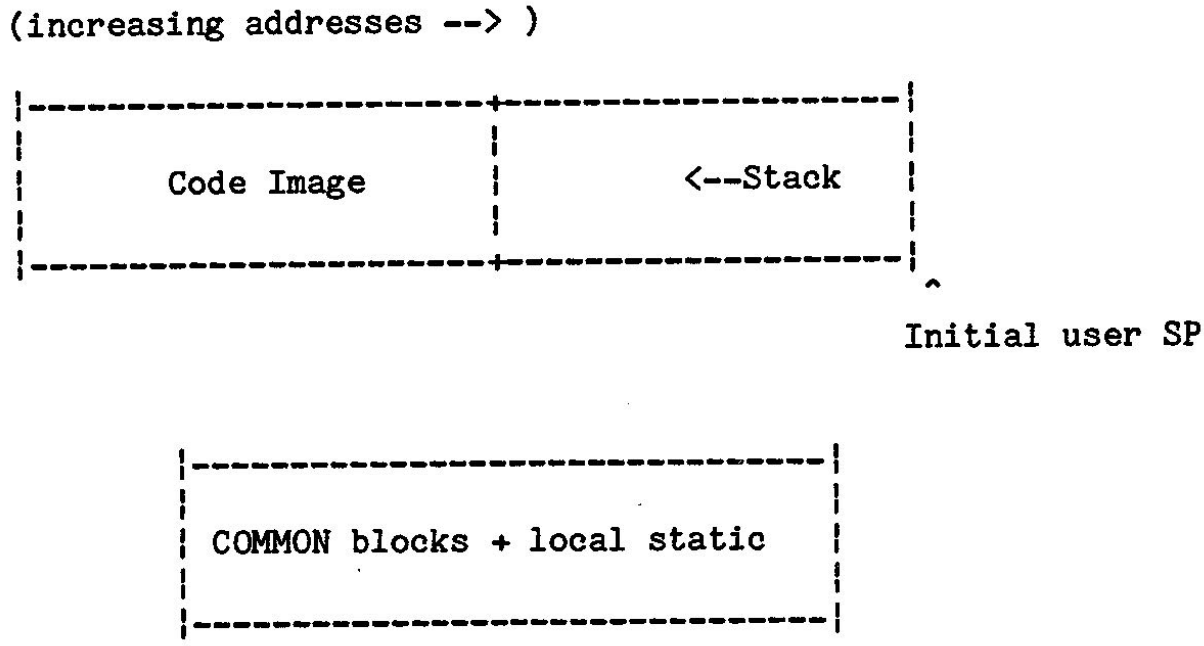
The user stack pointer SP is set to the highest address in the workspace area plus one, and the stack "grows" from higher to lower addresses. Almost all of this area is used for the stack, and if at run-time the library finds that the stack is about to overflow, execution of the user program is terminated. The program must be re-run with a larger amount of workspace (see part III section 5).

Depending on a program's requirements, extra memory areas may be allocated dynamically at run-time by the library heap manager.

The detailed layout of a program in memory is as follows:-

There are two separately allocated memory areas:

(1) the executable code image and heap/stack area
(2) the COMMON blocks and local static data.



In addition, there is one further area allocated dynamically by the library, containing global run-time information shared by the top-most job and its dependent child jobs. The size of this area is a few hundred bytes.

(The description above relates to an individual object program, which may be run as a job by operator command or as a child by a parent program. The Prospero Resident Library, or PRL, is a fixed area of code, of rather less than 16K bytes, which is distinct from the areas just described. Only one copy of the PRL is in the machine at one time - it is typically in ROM, in fact - and it is shared between any active jobs.)

8.3.2 Formats of variables

Variables of type INTEGER (INTEGER*4) occupy 4 bytes, arranged most-significant to least-significant in ascending addresses. Type INTEGER*2 occupies 2 bytes, with the normal high-low convention of the 68000; the range of values accommodated is -32768 to 32767. Type INTEGER*1 occupies 1 byte, the range of values being -128 to 127.

The types LOGICAL (LOGICAL*4), LOGICAL*2 and LOGICAL*1 are similar to the corresponding integer types, with the values .FALSE. and .TRUE. being represented by 0 and 1 respectively in the least significant byte only, the other bytes (if any) being unused.

REAL values occupy 4 bytes in a format corresponding to the proposed IEEE Standard. The 32 bits are made up as follows (from most to least significant):

1-bit sign
8-bit binary exponent, biassed by 127
23-bit mantissa, with an implied 1 in the most significant (24th) bit position

DOUBLE PRECISION values occupy 8 bytes in the IEEE format:

1-bit sign
11-bit binary exponent, biassed by 1023
52-bit mantissa, with an implied 1 in the most significant (53rd) bit position

In both formats, the implied binary point is between the implied '1' bit and the most significant actual bit of the mantissa. Thus the value 1.0, for example, is represented by the following bit-patterns:

32-bit	$3F800000
64-bit	$3FF0000000000000

Values of COMPLEX type occupy 8 bytes and are represented by two REAL numbers, the imaginary part being in higher addresses than the real part.

CHARACTER values occupy the declared number of bytes, with one character stored per byte, in ascending locations.

Arrays are arranged with the element having the lowest subscript value in the lowest address. Array elements are stored in successive locations according to the "subscript value" (see section 3.4).

Variables and arrays of type other than INTEGER*1, LOGICAL*1 and CHARACTER are word-aligned.

8.4 Interfacing to assembler

8.4.1 Use of assembly language

To use machine features not available through the Fortran language, for example interrupts, routines may be written in assembly language and combined with the generated code during the link-edit process.

8.4.2 Choice of assembler

The Fortran compiler generates relocatable object code. Assembler language modules may be processed by any assembler which generates the same format, and linked with the other components of the program. In particular, the GST Macro Assembler will be found satisfactory.

8.4.3 XDEF/XREF linkage

8.4.3.1 Calling assembler from Fortran

An assembler-coded routine can be called from Fortran in the normal way as a subroutine with a CALL statement, or may be invoked as an external function if it returns a value. In the assembler module, the name is quoted in an XDEF directive, or made global in some equivalent way. More than one routine can be in the module. Return is made by an RTS instruction or equivalent.

To make these remarks more specific, consider the Fortran fragment:

```
      ..
      CALL  SUBA(X,Y)
      ..
```

The assembler code for a routine SUBA which is called in this way should be structured as shown below.

```
*.........................
* Fortran calling assembler
*.........................

      XDEF    SUBA

      SECTION .CODE

SUBA
      MOVEA.L 4(SP),A0        Get address of argument Y
      MOVEA.L 8(SP),A1        Get address of argument X
      ..
      ..
      MOVE.L  (SP)+,A0        Caller's link
      ADDA.W  #8,SP           Remove SUBA's arguments from stack
      JMP     (A0)            Return to Fortran
```

8.4.3.2 Calling Fortran from assembler

A Fortran subroutine or function can be called from assembler code. The subprogram name must be quoted in an XREF directive (or equivalent), and called by a BSR.L instruction (this assumes that the distance between the BSR and the called routine is expressible as a long BSR operand; if this is not the case, another technique must be used which is explained below). If the subprogram requires arguments (see 8.4.6) they must be pushed on the stack before the call.

To make these remarks more specific, consider the Fortran fragment:

```
      SUBROUTINE SUBF(I,J)
      ..
      END
```

The assembler code for a routine which calls subroutine SUBF, should be structured as shown below. In particular, in the large program example, the .ATAB section name must be used, so that code base addresses will be relocated correctly.

```
*.................................................
* Assembler calling Fortran (small program example)
*.................................................

        XREF    SUBF

        SECTION .CODE

        ..
        ..                          Set up SUBF's arguments on stack
        PEA     I
        PEA     J
        BSR     SUBF                Direct call to Fortran subroutine

*             (SUBF must be within 32K bytes of this BSR)
        ..
        ..

I       DS.L    1
J       DS.L    1

        ..
```

```
*.................................................
* Assembler calling Fortran (large program example)
*.................................................
```

This linkage can always be used, but must be used if the distance between the calling and target routines exceeds 32K bytes.

```
        XREF.L  SUBF            (The .L is essential!)
        XREF    .ATABS          .ATABS is a reserved public symbol

        SECTION .ATAB           .ATAB is a reserved section name

JMPSUBF JMP     SUBF            Direct 6-byte jump to SUBF

        SECTION .CODE

        ..
        ..                      Set up SUBF's arguments on stack
        PEA     I
        PEA     J
        JSR     JMPSUBF-.ATABS(A4)  Indirect call to SUBF

*                   (A4 contains the address of .ATAB at run-time)
        ..

I       DS.L    1
J       DS.L    1
```

8.4.4 COMMON data

Fortran variables which have been declared in COMMON can be referenced from assembler code as shown below. The use of section .ATAB is necessary in order that program initialization can relocate the common block address at run-time.

In the general case, the assembler declarations must describe the layout of the Fortran common block. Section 8.3.2 gives details of storage layout for different data types.

As an example, the following might appear in a Fortran program:

```
        INTEGER*1 HOURS,MINS,SECS
        COMMON /TIMER/ HOURS,MINS,SECS
          ..
        PRINT 100, HOURS,MINS,SECS
  100   FORMAT (' Time is: ',I2,':',I2,':',I2)
```

The method of accessing the common block TIMER is as follows

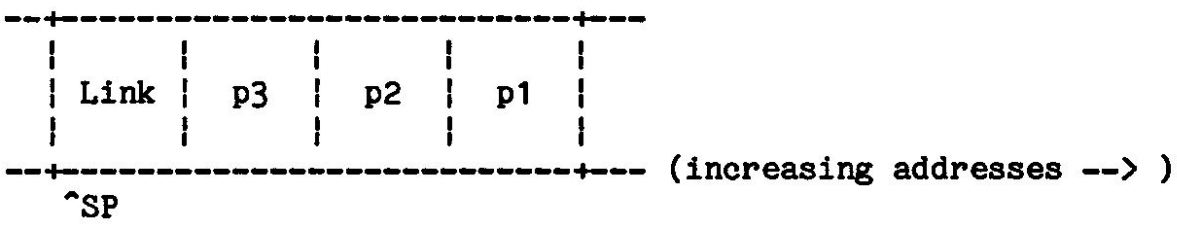
```
*.........................
* Accessing a COMMON block
*.........................

        XREF    .ATABS

*  (.ATABS is a reserved public symbol defining
*  the start of section .ATAB)

        COMMON  TIMER

*       Layout of TIMER

HOURS   DS.B    1
MINS    DS.B    1
SECS    DS.B    1

        SECTION .ATAB           .ATAB is a reserved section name

ATIMER  DC.L    TIMER           Address of /TIMER/ at run-time

        SECTION .CODE

        ..
        MOVEA.L ATIMER-.ATABS(A4),A0    Get base addrs. of /TIMER/

*               (A4 contains the address of .ATAB at run-time)

        MOVE.B  HOURS(A0),D0    Get contents of HOURS
        ..
```

8.4.5 Preservation of registers

The generated Fortran code depends upon the contents of registers A3 to A6 being unchanged on return from a subroutine. Assembler-coded subroutines or functions must conform with these requirements. On return, the link and all argument addresses must have been removed from the stack.

8.4.6 Arguments

When a subroutine or function has arguments, the actuals are pushed on the stack prior to the call. The first argument is pushed first, and so is furthest from the return link on entry to the procedure.



On return, arguments as well as link must have been removed.

In all cases, the stack contains the addresses of the actual arguments, each address occupying 4 bytes. If the actual argument is a variable, array or array element, the address is that of the item concerned (and so results can be returned), but if the actual agument is a more general expression, the address is that of a temporary workspace which is not addressable by the calling program. In the case of character variables and expressions, the address passed is that of a CVD (Character Variable Descriptor), a 6-byte datum, consisting of a 4-byte address followed by a 2-byte length, that describes the character variable or expression being passed.

8.4.7 Function results

A function call passes an additional "hidden" argument on the stack before the actual arguments are passed. The extra argument is the address of a location in the caller's data space that is to receive the function result. In the case of a character function, the extra argument is the address of a CVD.

8.4.8 Reserved section names

The section names .CODE and .ATAB and public symbol .ATABS are reserved and should only be used as shown by the examples above. The section names .INIT, .NAME, .ENTRY and .LWT are also reserved and must not be used by any assembler routine under any circumstances.

INDEX

In this index, word-symbols (IF, INTEGER, etc.) and standard names (AMAX0, SIN, etc.) are distinguished by the use of capital letters. Words from the formal syntax (program-unit, common-block, etc.) are distinguished by not having a capital letter.

PART III - PRO FORTRAN-77 OPERATION

1 INSTALLATION DETAILS

1.1 Hardware requirements

The hardware required to run the Fortran-77 compiler is a Sinclair QL computer running QDOS with at least 80K bytes of user RAM, and with the PRL ROM cartridge fitted in the QL's ROM slot.

N.B. The PRL ROM cartridge must only be fitted or removed with the QL's power supply disconnected.

The minimum requirement for user programs is a Sinclair QL computer running QDOS. Memory and peripheral requirements depend on the program.

1.2 Delivery and installation

The Fortran-77 software is delivered mainly on microdrive cartridges, containing the following files:-

F77	Fortran compiler control program
PROFOR1	Fortran compiler pass 1
PROFOR2	Fortran compiler pass 2
PROFOR_ERR	Fortran compile-time error messages
LINK	Linker
F77LIB_REL	Fortran run-time library
PLINIT_REL	Fortran library initialization module
PLEND_REL	Fortran library end module
F77_LINK	Standard Fortran linker command file
BOOT	SuperBASIC program to install PRL
PRL	Prospero Resident Library - see section 4.1
PROLIB	Librarian program
SETDDEV) NOQNS) SETSTACK)	Fortran configuration programs
TRAPREG_FOR	"TRAP" include file - see Part II, 8.2.12

Also supplied on microdrive are a few example source program (_FOR) files. If there are any special comments relating to the software, for instance descriptions of extra files, they are placed in a file called READ_ME. If this file is present, consult it before using the software.

The remaining Fortran-77 software is supplied in a ROM cartridge. This contains the Prospero Resident Library (PRL), and it must be installed in the QL's ROM slot before using the Fortran compiler and also before running a Fortran program. Disconnect the QL's power supply before installing or removing the PRL cartridge.

Before use, it is essential to create working copies of the compiler and other files, and to use these rather than the supplied files, which should be kept safely as master copies.

The files required for compilation of Fortran source programs are F77, PROFOR1, PROFOR2 and, optionally, PROFOR_ERR (if the latter is not present, error messages will identify the error type by number without the text). The disposition of these files will depend on the user machine configuration:

For users with a basic QL, it is recommended that two "compiling" cartridges be created, the first containing F77, PROFOR1 and PROFOR_ERR, and the second containing PROFOR2 and PROFOR_ERR. Then the software is immediately usable when loaded from microdrive 1.

For users with a disc system, a "compiling" disc may be set up containing all the above four files. In addition, a configuration file F77_CONFIG must be set up and program F77 configured to use this file (see section 5).

It is recommended that source files be kept on their own separate cartridges or discs. Note that source file names must end in _FOR. By default, there must also be sufficient space on a source file medium for the _REL files produced by the compiler, and also for the compiler's work file, which typically occupies about as much space as the source file. This default arrangement can, however, be changed by re-configuring (see section 5).

For microdrive users, it is recommended that a "linking" cartridge be created with the Linker and the files F77LIB_REL, PLINIT_REL, PLEND_REL, F77_LINK and PROLIB on it. For disc users, these files can probably all fit on the "compiling" disc.

In the descriptions which follow, it is assumed that the "compiling" media are in MDV1 and the user's source media in MDV2.

1.3 Simple compile, link and execute

To prove that the software is installed and functioning correctly, make working media (see above) containing the compiler and linker files and copy the sample program source PRIME_FOR to a source file medium.

1.3.1 Compile

With the first compiler cartridge in MDV1 and the source cartridge in MDV2, type

```
EXEC MDV1_F77
```

and the following output is generated in the compiler's window (the user must type the underlined information):

```
Pro Fortran-77  -  Version mmq 1.1
Copyright (C) 1985 Prospero Software

Source filename - MDV2_PRIME<ENTER>

Default options:

G - console output to LOG file ? (Y/N/.) .<ENTER>

     Unit   PRIME

Load compiler pass 2 in MDV1 and press ENTER

Name:  PRIME
                Lines:   33
                Code:   500
                Data:     8
```

Note that the prompt for the second compiler pass only occurs for microdrive configurations. At this point, the source file PRIME_FOR on MDV2 has been compiled into a "relocatable binary" file PRIME_REL also on MDV2.

In the report output at the end of pass 2, the "name" is the module name of the relocatable module produced by the compilation. This name will be displayed when the linked program is executed. The "code" and "data" figures are in bytes, and represent the total generated code (+ constants), and data (excluding COMMON data items), respectively.

1.3.2 Link

To "link edit" the file PRIME_REL and produce an executable program, place the linker cartridge in MDV1 and type

```
EXEC MDV1_LINK
```

and enter the Linker command

```
MDV2_PRIME-WITH MDV1_F77
```

The filename MDV1_F77 is converted by the Linker into the name MDV1_F77_LINK. It uses this "template" file to link PRIME_REL with selected library subroutines from the file F77LIB_REL. (Further details are in section 3.)

The result of linking is the file PRIME_BIN on MDV2.

The command DIR MDV2_ shows three PRIME files: the source _FOR, the executable program _BIN, and also the relocatable version _REL which is generated by the compiler and read by the Linker.

1.3.3 Execute

Simply enter

```
EXEC MDV2_PRIME_BIN
```

Program PRIME reads from and writes to unit *, which is by default assigned to the console window. It repeatedly asks for a number and prints its smallest factor (or else 'Prime'). For example (with input underlined):

Input a positive number less than a thousand million : 999999989

Smallest factor of 999999989 is :
4327

Input a positive number less than a thousand million : 999999937

Smallest factor of 999999937 is :
Prime

and so on.

To terminate PRIME, you can enter a non-numeric value when it asks for a number. This will cause a run-time error to be reported (see section 4.3), and you should reply N to the prompt.

The extra facilities of the compiler are explained in the next two sections. In particular, it is shown how to compile and link programs consisting of more than one source file.

2 OPERATION OF THE COMPILER

Each invocation of the QL Fortran-77 compiler processes one source file. Each source file may contain any number of Fortran program units (separated by blank lines, if desired), and the compiler converts this into a binary output file in Sinclair relocatable format. Each source file becomes a binary output file consisting of a single relocatable module.

Compilation is a 2-pass process under the overall control of program F77. Pass 1 (the program PROFOR1) reads the source file and generates a temporary work file, with a name of the form TEM$_<job id>_IL, which contains a semi-compiled "intermediate-language" representation of the source program. When processing of Pass 1 is complete, F77 gives control to Pass 2 (the program PROFOR2). This program reads the work file, and generates the relocatable _REL file. When compilation is complete, Pass 2 deletes the work file and gives control back to F77, which then terminates.

The previous section has described the simple form of operation of the compiler. In this section, the various options and messages are explained.

2.1 Forms of invocation

The Fortran compiler may always be invoked and run interactively, and, if the QL Toolkit is installed, in batch mode as well, although the latter method may not be possible in an unexpanded QL owing to memory limitations. The following text assumes that the compiler is installed on microdrive, but it will equally apply to disc-based systems as well.

2.1.1 Interactive mode

With the first compiler cartridge in MDV1, enter:

```
EXEC MDV1_F77
```

or use EXEC_W if desired. After opening its window and signing on, the Fortran compiler asks for the name of the _FOR source file to be compiled. It outputs:

```
Source filename -
```

and the user should enter, e.g. "MDV2_CALC", it being assumed that there is a file called CALC_FOR on MDV2. If this file cannot be opened, the prompt is repeated.

The compiler then displays the currently configured default compilation options as a reminder (by default, no options will be configured, but the user may configure options as described in section 5):

Default options: <defaults>

The compiler then displays a series of prompts, one for each option, each of which may be replied to with "Y" (or "y") to select an option, or "N" (or "n") to not select an option, or "." to terminate prompting and use the defaults from then on:

G - console output to LOG file ? (Y/N/.)
I - range checks on subscripts ? (Y/N/.)
A - range checks on assignments ? (Y/N/.)
N - track source names & line numbers at run time ? (Y/N/.)
M - map ? (Y/N/.)
L - source listing ? (Y/N/.)
U - report undeclared variables ? (Y/N/.)
T - INTEGER means INTEGER*2 ? (Y/N/.)
C - compact object code ? (Y/N/.)

By default, the compiler simply reads a _FOR source file and produces a _REL file, which is always on the same device as the source. If the log file option is selected, a _LOG file is produced. If the map option is selected, a _MAP file is produced. If the listing option is selected, a _PRN file is produced. The contents of these optional files, which are also produced on the same device as the source, are described in later sections.

2.1.2 Batch mode

With the QL Toolkit installed, a single command suffices to perform one compilation run. Assuming the use of floppy discs, enter:

EX FDK1_F77;'<source>/<options>'

(or use EW if desired), where

<source> might be, e.g. "FDK2_CALC", there being a source file CALC_FOR on FDK2, and

<options> gives the desired compilation option letters, e.g. "GIA", separated from <source> by "/".

Examples:

EX FDK1_F77;'FDK2_TABLE/NUM' causes the source file TABLE_FOR on FDK2 to be compiled with options N, U and M (see below).

EW FDK1_F77;'FDK2_CALC' causes the source file CALC_FOR on FDK2 to be compiled with the currently configured default options.

2.2 Compile-time options

The various compile-time options are described in the following sub-sections. The default setting for each option is "off", or N, when the software is shipped, but this can be altered by introducing a configuration file (see section 5).

2.2.1 G - console output to LOG file

When this option is specified, the messages output by the compiler to the console during compilation are written also to a file. The name of the file is the same as that of the source, with "_LOG" added. This can be a useful facility, both for inspection of compile-time errors and for recording the compilation status of each source program (code size, etc.).

2.2.2 I - range checks on subscripts

Range checks determine whether or not subscript expressions are within the correct limits. The checks are carried out just before a subscript value is to be used, and have the effect of generating more code.

Range checks can be valuable in the early stages of program testing. If code size or speed is at a premium, they may be switched off once the program has been tested.

2.2.3 A - range checks on assignments

Assignment checks determine whether or not the values assigned to INTEGER*1 and INTEGER*2 variables are within the range allowed for such quantities. The checks are carried out just before a value is to be assigned, and have the effect of generating more code.

2.2.4 N - track source names & line numbers at run time

This option instructs the compiler to insert extra code into the object program to maintain during execution a record of the source file name and line number corresponding to the code currently being obeyed. This information will be displayed in the event of any run-time error, and if the error is within a subprogram then the calling stack which is printed out (see section 4.3) will contain these file names and line numbers (for all calls which occurred in program units compiled with this option).

2.2.5 M - map

With this option, a file is generated containing information about all the names used in the source program units. The file has the name of the source with "_MAP" added, on the same device as the source file. After the compilation it may be listed or displayed as desired. Use of this facility can be an aid in the early stages of program testing, for instance, in verifying that all names have been correctly typed (in both senses of the word).

For each program unit, the information is ordered alphabetically by name, and consists of:

Kind — Whether the name refers to a variable, an array, a statement function, an intrinsic function, or an external function or subroutine

Type — 'Int' (for INTEGER), and so on, or blank for external subroutines and for data items which are never referenced in executable statements

Area — 'Data', for items which are not in COMMON, or the name of the common block, or 'Dummy', for dummy arguments to subprograms or statement functions

Offset — The relative address within the Data area or COMMON block, in hexadecimal (and decimal)

The total sizes of the Data area and COMMON blocks (if any) are also given.

2.2.6 L - source listing

A listing of the source can be generated as a by-product of compilation. Each line is preceded by its line number within the file and by the relative hexadecimal address of the start of that line within the object code. The listing is output to a file with the name of the source but ending in "_PRN" on the same device as the source. After the compilation it may be printed or displayed as desired.

2.2.7 U - report undeclared variables

This option aids good programming practice by causing the compiler to report all references to variables that have not appeared in the specifications part of a program unit (see Part II, section 2.1). Use of this option can also help in detecting mis-spelt variable names.

2.2.8 T - INTEGER means INTEGER*2

This option instructs the compiler to treat all occurrences of INTEGER as if INTEGER*2 applied, namely:

INTEGER statement

undeclared integers

IMPLICIT integers

INTEGER FUNCTION statement

All integers declared with an explicit length (e.g. INTEGER*4) are unaffected by this option.

2.2.9 C - compact object code

If the compact code option is invoked, the compiler substitutes shorter (but somewhat slower) alternatives for certain object code sequences. The amount of difference this will make depends on the nature of the program (and is anyway rather small). Use of the option would only be recommended for particularly large programs.

2.3 Compiler messages

When the compilation process proper begins, messages are output to the console to report progress and any irregularities.

2.3.1 Normal messages

In the main, these are self-explanatory. At the end of Pass 2, the sizes of the code and data areas generated, and the total number of source lines, are printed. These are all decimal values. The data sizes do not include any COMMON variables.

2.3.2 Error messages

If the specified source file does not exist, the request for a file name is repeated.

If there is insufficient memory for compiler workspace or stack, one or other of the compiler passes will fail to run, and the compiler will terminate the compilation. Some possible corrective actions are given below.

Errors in the source program may be detected during either of the passes, though the majority generally appear in pass 1. The format in each case is: source line number and error code, with an explanatory sentence if the file PROFOR_ERR is present, followed by the text of the source line in error (pass 1 only). In Appendix B is a list of the error codes, with somewhat fuller descriptions where appropriate.

The first digit of the error number indicates the "severity level" of the error and the subsequent action taken by the compiler, according to the following scheme:

First digit	Compiler action
0	Warning. Compilation continues. The program should execute normally.
1	Compilation continues. The program will execute normally except (perhaps) if control reaches the point at which such an error is located.
2	Compilation continues, but object code generation ceases.
3	Skip to end of current statement, then continue compilation but with no code generation.
4	Terminate compilation

Errors of level 2 and above (i.e. those that cause code generation to be terminated) are classified as "fatal".

The other possible problems which may arise during compilation are connected with running out of space, either in memory or on a device (e.g. insufficient room for the _REL file). Such events give rise to error messages in the normal run-time error format (see section 4.3).

If the compiler indicates that its stack space has become full, this is most likely due to an unusually complicated expression in the source being compiled, e.g. the use of many levels of nested parentheses. The normal remedy for this would be to simplify the source expression.

If run-time error H is encountered during compilation, the normal remedy would be to repeat the compilation with a larger RAM area for the compiler to run in. (Resetting the machine prior to compiling may help, in this respect.) If no more memory is available, however, the only solution is to reduce the size of the compilation input.

3 USING THE LINKER WITH FORTRAN

The Linker combines the output from one or more executions of the Fortran-77 compiler with modules from the supplied run-time library to construct an executable program file.

Linking is a 2-pass process, converting a collection of _REL files into an executable _BIN file. By default, a report is produced in a file with the same name as the _BIN file but with the extension _MAP. This report can be suppressed by including "-NOLIST" in the linker command line (see below).

In section 1, it has already been shown how, in the simple case of the compilation source being an entire program, the supplied Linker command file F77_LINK can be used directly to link the executable program.

The contents of the file F77_LINK is as follows:

```
INPUT MDV1_PLINIT
INPUT *
LIBRARY MDV1_F77LIB
INPUT MDV1_PLEND
DATA 4K
COMMON DUMMY
```

Each INPUT command directs the linker to include the specified relocatable file, the extension _REL being automatically supplied. The special form "INPUT *" causes the first filename in the linker command line to be included, again after appending _REL. A LIBRARY command, on the other hand, instructs the linker to include from the named file only those modules required, in the sense of having been referenced from module(s) already included. (So in this case, the library file MDV1_F77LIB_REL will be selectively scanned.)

If an executable program is to be constructed from more than one relocatable file - the output from two separate compilations, perhaps, or a Fortran compilation and some Assembler-coded modules - there are two ways to proceed. Either the separate relocatable files can be combined into one, by using the PROLIB utility (see section 6), or a new _LINK template file can be created, by editing extra INPUT command lines into a copy of F77_LINK.

When editing this file, it is essential to note that:

a) The file PLINIT_REL must be the first INPUT file.
b) Additional INPUT lines should be positioned before the LIBRARY line(s).
c) Additional user libraries must come before the F77LIB LIBRARY line.
d) PLEND_REL must be INPUT after all other _REL files and LIBRARY files.
e) The COMMON DUMMY option must be used.
f) If SECTION commands are introduced, the first such must be: SECTION .ENTRY.
g) The Linker's OFFSET command must not be used.

You may also wish to alter the size of the data space allocated by the DATA command. This value must be large enough to cover the following requirements:-

a) Two FCA (File Control Area) control blocks for standard i/o, the size of each being about 110 bytes.

b) Stack requirements, being 16 bytes for each procedure called, 4 bytes for each actual argument, and an extra 4 bytes for function calls. The calculation of stack space needs to allow for the worst case, i.e. the deepest nesting of subroutine calls, and should include an allowance for use by the run-time library, as well as some margin of safety.

As an example, suppose it is required to combine an Assembler-coded module ASS_REL with a Fortran-coded module FORT_REL, and that a run-time stack requirement of 6K is required. Then one method of linking such a program is to create a linker template file called FORT_LINK, say:

```
INPUT MDV1_PLINIT
INPUT *
INPUT ASS
LIBRARY MDV1_F77LIB
INPUT MDV1_PLEND
DATA 6K
COMMON DUMMY
```

The link operation can be performed by:

```
EXEC MDV1_LINK
```

and then entering, as the linker command line:

```
MDV1_FORT -WITH FORT
```

(The "-WITH" can be optionally omitted, in fact.) The linker uses the first name in this command (MDV1_FORT) to fill out the "*" place holder in the second INPUT line of FORT_LINK, and also as the "root" name for constructing the names of the executable file (by appending _BIN) and the storage-allocation report file (by appending _MAP). An executable file called MDV1_FORT_BIN is therefore produced, and can be executed by (for example):

```
EXEC MDV1_FORT_BIN
```

4 OPERATION OF OBJECT PROGRAMS

4.1 PRL

The Prospero Resident Library (PRL) is a collection of machine-code routines required by all Fortran-77 programs, and also by the compiler. Before running a Fortran program, or the compiler, PRL must be installed. PRL is supplied in the ROM cartridge issued with the compiler, and also as a separate program on a microdrive cartridge. The ROM cartridge must be used for the compiler, but compiled Fortran programs can be run using either the ROM cartridge or the "software" PRL loaded from a file. The ROM PRL is installed by plugging it in to the ROM socket at the rear of the QL before applying power. The "software" PRL is installed by means of the SuperBASIC command:

```
LRUN  MDV1_BOOT
```

(This command is, of course, invoked automatically at startup by the QL computer if there is a cartridge present in MDV1 holding the file MDV1_BOOT.

Once PRL is installed, it does not need to be re-installed except when the QL is reset or powered down with no ROM present. The SuperBASIC command

```
PRL
```

can be used to check the presence and correctness of an installed PRL.

The BOOT + PRL files can be copied along with linked Fortran object programs for use on machines other than the one used for compilation. PRL substantially reduces the size of object programs, and the time required for linking and loading programs, because the routines in PRL would otherwise be linked into every compiled program.

4.2 Execution of Fortran object programs

The normal way of executing a Fortran program that has been linked is by means of the SuperBASIC EXEC command (see below).

When executed, Fortran programs locate PRL and open a CON_ window for the display of run-time error messages and so on. This window also acts as the default standard input and output file (UNIT = *), unless otherwise specified.

4.2.1 Invocation by EXEC or EXEC_W

In order to execute a compiled and linked Fortran program, the SuperBASIC command EXEC or EXEC_W is used (or else the Toolkit command EX or EW). For example:

```
EXEC MDV1_PROG_BIN
```

After opening a window and signing on, the Fortran program prompts the user as follows:

```
Standard input file? <>
Standard output file? <>
Option string? <>
```

with <> showing where the cursor is positioned awaiting user response. In simple cases, the ENTER key can be pressed for each question.

The first response connects a data file for standard input via "UNIT *". If a file name is entered, it is opened for exclusive input. If the open fails, the prompt is repeated. If only ENTER is pressed, any use of "UNIT *" in a READ statement in the program will be directed to the standard window that is being used for this initial dialogue.

The second response connects a data file for standard output via "UNIT *". If a file name is entered, the file is opened for new overwrite. If only ENTER is pressed, any use of unit * in a WRITE statement, or use of a PRINT statement, in the program will be directed to the standard window.

The last response specifies a line of up to 80 characters to be made available as an option string to the running program. If the line is too long, the prompt is repeated. If only ENTER is pressed, the option string is of zero length. (The option string is obtained within the user program by calling the GETCOM subroutine - see part II, section 8.2.1).

The above dialogue can be suppressed in a program by means of the NOQNS program issued with the compiler - see section 5.3 below.

4.2.2 Invocation by EX or EW Toolkit commands

In this case, there is a choice between running interactively as in 4.2.1 above, and running autonomously. For example (assuming suitable default file devices):

```
        EX program_name

or      EX program_name;option_string

or      EX program_name,infile_spec,outfile_spec;option_string
```

where zero or two data files may be specified (but not just one), and "option_string" is a SuperBASIC format string of characters to be passed to the initiated program.

In the first format, the program runs just as if EXEC had been used, that is, interactively, as described in the previous section. But in the second and third examples, the program starts with no further user interaction.

In the second example, standard input and output will be assigned by the program to its own window, whereas in the latter, the specified data files will be made available to the user program for standard input and output. It is an error to give only one datafile - an extra dummy input or output file must be specified. (If more than two datafiles are specified, only the first and last files are made available to the user program as standard input and output: the others will not be accessible.)

In the second and third examples, the option_string is optional and, if omitted, a zero length string is passed to the initiated program.

Under EX control, there is no limit on the size of option_string as there is when interactive the program control is used.

4.2.3 Handling of pre-connected files

The previous two sections have shown how the user may specify one or two files to be pre-connected to a Fortran program, with the connection defaulting to the standard window. Then user program references to unit * will be routed to the specified or defaulted files.

4.2.4 Invocation using EXECPG

Programs can also be executed from within Fortran programs using the EXECPG function (see Part II, section 8.2.6). In this case, errors are reported to the initiating program by means of return codes. These return codes are listed alongside the corresponding messages for normal program initiation in section 4.4 below.

If a program uses the EXECPG facility to run a child program, its own pre-connected files are automatically made available to the child program (through any number of parent-child levels).

4.2.5 PAUSE and STOP statements

If a PAUSE statement is executed, the message

```
PAUSE  nnnnn
Continue ?
```

appears on the console. The program can be continued by pressing the key Y (or y), and aborted by pressing N (or n). (All other keys are ignored.) A similar message appears for a STOP statement, but without the option to continue.

4.3 Run-time errors

The only aspect of program operation not determined from the program itself arises if an error is detected by the run-time software.

Errors can occur during the initialisation process before the program has fully started, and which are therefore not reported via the normal run-time error reporting method. A complete list of error messages produced during this initialisation process appears in section 4.4 below.

Once program execution proper has commenced, errors may be detected in a number of situations: file handling, arithmetic operations, and so on. In some cases they may be found by the checking code incorporated by one of the compile-time options (see section 2.2). In all cases a report is made on the console, giving error type - identified by a letter - and the hexadecimal (base 16) machine address relative to the start of the code:

```
Error x at address aaaaaa
```

A list of the run-time error codes is given in Appendix C. The address aaaaaa is directly comparable with the addresses provided in the _MAP file generated by the Linker. For input/output errors, and some other cases, additional information appears with the standard message. A list of i/o error status values also appears in Appendix C.

The standard error message is followed by trace information showing how the point in error was reached. This takes the form of a list of addresses at which subroutine and function calls occurred. All addresses are relative to the start of the code. The first address given corresponds to the point where the main program called the next lower level of subroutine, and so on up to the actual subroutine or function in error.

For each source file which was compiled with the /N ("track source line numbers") option, the addresses in the error report are made more intelligible (without recourse to the linker's _MAP file) by the addition of the source file name, the subroutine name and the line number.

Finally, many classes of error allow continuation, and this choice is offered as a console option with (Y/N) response, exactly as in the handling of the PAUSE statement, described in section 4.2.5.

4.4 Miscellaneous error messages

Most of these messages appear in the initiated program's standard window, with departures from this rule noted under particular messages. Where a program was initiated using EXECPG, the error is passed back as a return code to the initiating program.

Too long: <prompt>

The reply to the previous <prompt> was too long, e.g. a filename exceeded 36 characters. The correct reply should now be given.

Embedded blanks not allowed

This can appear in the standard window during the running of a user program. The reply to the previous prompt contained an embedded space character. In particular this can occur when a program opens a workfile and no default device has been configured. The user is prompted for a device name, which may not contain spaces. A corrected reply should be given to enable the program to continue.

Execution error: <error text>

where <error text> is one of the following:

"bad command": (EXECPG return code -2)
An invalid SuperBASIC command string.

"pgm not opened": (EXECPG return code -3)
Only possible when using EXECPG. The specified user program file could not be opened (can be due to the filename exceeding 36 characters).

"not executable": (EXECPG return code -4)
The specified program file was opened successfully but the file header showed that it was not an executable program file.

"load failure": (EXECPG return code -5)
The specified program file could not be loaded into memory, due to a failure of the QDOS "load-file" operation.

"wrong version": (EXECPG return code -6)
There is an inconsistency between the version of PRL loaded and the version of the run-time software used when linking the user program. Most likely to be caused by upgrading to a new Fortran release without re-linking the user program with the new run-time software.

"out of memory": (EXECPG return code -7)
Insufficient memory is available for loading and/or running the user program.

"init. failure": (EXECPG return code -8)
The program has successfully been loaded, but then an error occurred in one of the following pre-execution steps:

(a) processing the relocatable items in section .ATAB,
(b) processing the data-initialization items in section .INIT

If the program consists purely of Fortran code, this error implies a problem with the Fortran software, and it should be reported. If user-provided assembler-language routines were included in the link of the user program, they should be checked to ensure that

(a) They do not use section .INIT.
(b) They only use section .ATAB, if at all, as described in Part II section 8.4, namely for achieving the relocation of common block addresses and of JMP instructions having 4-byte absolute operands. In particular, this error can be caused quite easily by not preceding the assembler instructions by a suitable SECTION directive, so that they become part of .ATAB instead.

It can quickly be verified whether or not assembler routines are the cause of this error, by linking the user program without them, then loading it. The program will then fail during execution, rather than during initialisation.

"no/corrupt PRL" (EXECPG return code -9)
PRL (the Prospero Resident Library) is not loaded, or alternatively has been corrupted and is no longer usable.

"window failure" (EXECPG return code -10)
An attempt to open read or write to the standard window has failed.

"only one chan" (EXECPG return code -11)
If either of standard input and standard output is specified using EX or EW commands, then they must both be specified.

"linking order" (EXECPG return code -12)
Unlikely to occur, but indicates an erroneous attempt to link with _REL files generated by other compilers or assemblers.

"no job created" (EXECPG return code -13)
QDOS failed to create a job for the program to execute under.

5 THE CONFIGURATION PROGRAMS

5.1 Configuring the compiler

The Fortran compiler consists of two main passes (PROFOR1 and PROFOR2) which execute under control of a small "parent" program F77. There is also an error message file PROFOR_ERR. During compilation, an intermediate file is produced. Together, these files are too large to fit onto a single microdrive cartridge. The default arrangements are that PROFOR1, PROFOR2 and PROFOR_ERR are on MDV1, and the work file is on MDV2. The compiler prompts for the PROFOR1 cartridge in drive 1 to be replaced with the PROFOR2 cartridge during the compilation. For users with other devices, the arrangements can be altered, as follows.

Use SETDDEV (see 5.2.1) so that the F77 compiler program has a default device specified for it. The compiler will then look on this device for a configuration file with the name F77_CONFIG. This is a five line text file, which may be prepared with a text editor, as follows:

line 1:	default compile time options (e.g. /GL)
line 2:	device holding PROFOR1 (e.g. FDK1_)
line 3:	device for PROFOR2
line 4:	device for PROFOR_ERR
line 5:	device for intermediate file

Example:

```
/GI
FDK1_
FDK1_
FDK1_
FDK1_
```

A second aspect of the compiler's operation which can be adjusted by the user is the size of its stack work space (which is only made use of to a significant extent when processing very complicated nested expressions). The compiler subprograms PROFOR1 and PROFOR2 are issued with a stack size suitable for an unexpanded machine. For some source programs, this may be insufficient, and a compile time error will indicate that it has been exceeded. In this event the compiler's capacity can be increased by running the SETSTACK program (see 5.2.3 below) on PROFOR1. The user with more than the basic 128K RAM may well choose anyway to modify his working copy of PROFOR1 to have a stack size of 8K, or even larger.

5.2 Configuring object programs

The configuration programs enable object programs to be tailored to suit the user's own requirements. Three utilities are provided on the issue cartridge. They are all machine code programs which require PRL to be installed.

The aspects which may be configured are:

a) the default device on which anonymous files are to reside;
b) whether the initial dialogue (described in part 4 above) takes place or not;
c) the stack size allocated for the running program.

Each utility reads an existing program file (which may or may not itself be configured), applies the changes specified and writes a new version of the program file. The question and answer session to specify the options required is largely self-explanatory.

5.2.1 Default device - SETDDEV

Each Fortran object program contains a field in which may be specified a "default device". This device is used by compiled programs for the placement of anonymous files (those for which no name was specified in the OPEN statement).

The initial value for this option is "no device". In this state, when a compiled program creates a work file, the user is prompted for the name of a device to use.

The default device may be altered using the program SETDDEV. It is invoked by:

```
EXEC xxxx_SETDDEV
```

(where xxxx is the name of the device holding the SETDDEV program), and the user is then prompted for the name of the program file to be modified, and the name of the device to be installed as the default for that program. A 4-character device name should be specified, or 4 blanks for "no device".

5.2.2 Initial dialogue - NOQNS

A dialogue normally takes place with the user when a Fortran object program is executed (as described in section 4 above). In many cases, this dialogue is not required and execution of the program can proceed immediately with default values for the initial responses. The initial dialogue (and associated messages) can be suppressed with the NOQNS program.

NOQNS is initiated by the SuperBASIC command:

```
EXEC xxxx_NOQNS
```

(where xxxx is the name of the device holding the NOQNS program). The user is prompted for the name of the program file to be modified, and the appropriate change is made.

5.2.3 Stack size - SETSTACK

The instructions to the linker contained in F77_link or any individual link job derived from it include a specification of the "DATA" size. In Prospero object programs the space so defined is used mainly for the stack, which in Fortran seldom becomes very large, and the 4K bytes in F77_LINK is normally ample.

The stack size for a program is modified using the SETSTACK program. SETSTACK is run by the SuperBASIC command:

```
EXEC xxxx_SETSTACK
```

(where xxxx is the name of the device holding the SETSTACK program). SETSTACK then gives instructions for modifying the stack size.

6 OPERATION OF THE LIBRARIAN

The purpose of the PROLIB librarian utility program is to administer files which are in Sinclair relocatable object format - such as those produced by Prospero's Pro Fortran-77 or Pro Pascal compilers, or by GST's macro Assembler. Individual modules may be extracted, and/or files may be merged together into libraries. A number of report options are also available.

A file created by PROLIB will be in Sinclair relocatable format, and so suitable for processing by linkers capable of handling this format, such as GST's LINK.

6.1 Forms of invocation

There are three ways of operating the librarian: the "one-line", the "conversational" and the "indirect" mode. All the options are available in each mode.

6.1.1 The one-line command

With the QL Toolkit installed, a one-line execution of PROLIB is possible, as in:

```
EX FDK1_PROLIB;'MDV1_PRIME/MX'
```

(or use EW if desired).

The option string (supplied in quotes after the ';' character) must be constructed as follows. First must come the name of the "library" file. This may optionally be followed by the character "/" together with one or more letters from the set M, X, U, N, D.

Each letter stands for a particular option regulating the report(s) that are produced by the librarian (see 6.2). The letters may be run together, as in this example, or may be separated by spaces or further / characters; they may be in upper or lower case.

A one-line command of the above form indicates a "read-only" operation on the library file: the file must already exist, and the purpose of the PROLIB execution is solely to list certain information about this relocatable file.

Alternatively, the library filename (and any option letters) may be followed by an "=" sign and one or more input filenames, separated by commas, as in:

```
EX FDK1_PROLIB;'MDV2_NEWLIB/M = MDV1_MOD1, MDV1_MOD2'
```

A one-line command of this form indicates a "create" mode of operation: if the library file already exists it will be overwritten, and the purpose of the PROLIB execution is to combine the input filenames into a new library with the given name. Any of the input filenames may be immediately followed by a "module selector" (see 6.3).

No filename extension may be given (whether for the library or the component input file names): the extension _REL is supplied automatically by the librarian.

6.1.2 Conversational mode

By entering the command (assuming a floppy-disk system):

```
EXEC FDK1_PROLIB
```

the conversational mode of operation is entered.

The first request is for the library filename. There is then a series of questions relating to the report options (cf. 6.2). Reply Y (or y) to select the option, otherwise N (or n). The final question is whether or not to create a new library with the given filename. If the answer is affirmative, the librarian repeatedly issues an invitation to input a line containing filename(s). The filenames are entered just as for the one-line mode of operation, that is, they must be separated by commas and each may be followed by a "module selector". To terminate this process, respond to the prompt

```
Input filename(s) -
```

with just <ENTER> on its own.

Again, no filename extension must be given: PROLIB automatically appends _REL to all filenames.

6.1.3 Indirect mode

The QL Toolkit must be installed for this mode to be used.

The indirect mode of operating the librarian combines the features of the first two modes: a one-line command is given containing the name of a "command file" (preceded by the character @), this command file containing the answers to the questions which would be asked in the "conversational" mode.

For example, typing

```
EX FDK1_PROLIB;'@ FDK2_MLIB'
```

where the text file MLIB contains the lines

```
MDV2_MLIB
N
N
N
Y
MDV1_M1LIB, MDV1_M2LIB, MDV1_M3LIB
```

causes PROLIB to combine the modules from the 3 files M1LIB_REL, M2LIB_REL and M3LIB_REL into the composite library file MLIB_REL. Note that if the command file name has no extension, none is supplied by PROLIB.

6.2 Report options

Whether or not in the "create" mode of execution, the librarian can be requested to produce a report describing the library file. (If in the create mode, the report will reflect the contents of the library file on completion of processing.)

The various report options are described in the following sub-sections. Each sub-heading contains (in brackets) the associated letter which must be written after the library filename in the one-line form of execution in order to invoke the option.

6.2.1 Module listing (M)

A report is produced which gives, for each module in the library file (in order of occurrence within the file), the name of the module, the Sections it contains, and all Public symbols defined and External symbols referenced within it. The "Sections" are pieces of the code or data which go to make up an executable program; their sizes (in decimal) are printed.

6.2.2 Cross-reference listing (X)

The report consists of two parts. The first part gives, for each Public/External name in the library file (in alphabetical order), the name of the module in which it is defined (i.e. is a Public name) plus the names of all modules in which it is referenced (i.e. is an External name).

The second part is a listing of all Sections (in alphabetical order) together with the names of the modules which reference them.

6.2.3 Unsatisfied references listing (U)

This report is concerned with the requirement imposed by many linkers that, for a library which is to be "selectively" searched (cf. the /S option described in section 6.3), the component modules must be ordered in such a way that, if module A contains an external reference to an entry point in module B, then module B must follow module A in the library file. The report lists all External names (in alphabetical order) which do not obey this rule, either because they are defined in an earlier module or because they are not defined at all.

6.2.4 Suppress .names (N)

only requested if M, X or U

(This option is only meaningful if at least one of M, U or X has been selected.)

In order to avoid conflict with user-defined names, most Public and Section names in the Fortran library begin with '.'. Since these are rather numerous, it can on occasion be desirable to suppress them. By specifying this option, no name beginning with '.' will appear in the report(s). The default is that all names, including those beginning with '.', are listed.

6.2.5 Listings to disc (D)

only requested if M, X or U

(This option is only meaningful if at least one of M, U or X has been selected.)

The default destination for reports is the console. If this option is chosen, the reports are written instead to a disc or microdrive file. The file is given the same name as the library file, but with the extension _PRN.

6.3 Module selection

In the "create" mode of operation (only), the user may specify that only some of the modules in an input file are selected. (The default is to select all modules from each file.) For this purpose, two kinds of "selector" are provided.

The first kind is the "selective scan" of an input file, and is specified by following the filename with the two characters "/S". Only those modules that have been referenced by previously selected modules will be incorporated into the output library file (and so into any reports).

Example: FDK1_FNAME/S

The second kind is by "module enumeration", and is specified by following the filename with the character "[", then a collection of module names, and finally the character "]". This "collection" of module names is to be written as a list of names, separated by commas; optionally, in place of a module name, the list can contain, at any point, two names separated by "-" (i.e. name1 - name2), signifying "all modules from name1 to name2 inclusive".

Example: FDK1_FNAME1 [MOD1, MOD4 - MOD8, MOD16]

A particular filename can be followed by at most one of these two kinds of selector.

An example of an input line containing all the above features is:

MDV1_FN1, MDV1_FN2 [M6], MDV1_FN3 [MOD3-MOD9], MDV2_LIBN/S

6.4 Librarian messages

6.4.1 Normal messages

If in the "create" mode, when it starts to process each input file the librarian writes the full filename to the console.

6.4.2 Error messages

6.4.2.1 Non-fatal errors

If an input file is empty, this is reported and the next file is processed.

If an input file cannot be found (perhaps because its name has been misspelt), the librarian reports this and invites more filename(s).

If a character other than 'S' is supplied after '/' following an input filename (i.e. where a "selective scan" directive is anticipated), the librarian reports this error and ignores the incorrect character.

6.4.2.2 Fatal errors

If any other error situation occurs, execution is aborted immediately, after outputting a message to the console.

The first group of such messages are caused by driving the librarian incorrectly. There are 4 such.

Command line improperly terminated

> In the one-line command mode, the library filename and switches have been read, followed by a character other than "=".

Command file not found

> In the indirect mode, the filename after the @ character is illegal or the file does not exist.

No library filename supplied

> In the indirect mode, the first line in the command file should contain a valid QDOS filename.

Illegal module-selection syntax

> The rules given in 6.3 have been broken. In particular, "-" must have a module name on either side of it, and "[" must have a matching "]" on the same line.

The other group of errors should never occur. The most probable cause is that an input file is not in Sinclair relocatable format at all. The error messages are:

Cannot find subsection
End of input file encountered
Illegal directive encountered
Illegal id encountered
Inconsistent section id
Input file relocatable format incorrect
Name in input file exceeds 32 characters
SECTION/COMMON inconsistency

A LANGUAGE SUMMARY

A.1 NOTATION

The notation used throughout this manual for the Fortran-77 syntax is summarised in the following table:

Notation	Meaning
=	is defined to be
¦	alternatively
[x]	zero or one instances of x
{x}	zero or more instances of x
<x¦y¦...¦z>	grouping: any one of x, y, ..., z
lower-case-name	a non-terminal symbol

(For increased readability, the non-terminal symbols are often hyphenated.) Any string of characters not covered by the above list is a terminal symbol: it stands for itself; for example:

```
ASSIGN
.TRUE.
(
**
```

In this appendix, the nature of the source file which is input to the compiler (the 'compilation-input') is viewed from two complementary aspects: the lexical (or bottom-up) and syntactic (or top-down). These views merge at about the level of the 'token'. The division of the remainder of this appendix into two subsections is designed to mirror these two viewpoints.

The definitions in each of the following subsections are grouped and ordered according to their 'level'. At the first level comes, in each case, the definition of the 'compilation-input': the only concept given two (complementary) definitions. The definition of any other concept is to be found on the next level to that in which the concept first appears. The definition level is printed at the left margin.

Taken together, subsections A.2 and A.3 contain one, and only one, definition for every non-terminal symbol. The only exception - apart from 'compilation-input' - is 'string-character', which stands for any 8-bit-code character. (The characters in the 7-bit-code ASCII set are printed in Appendix D.)

Except within a character-constant, there is no distinction in meaning between the upper- and lower-case versions of any letter.

A.2 LEXICAL ASPECTS

1 compilation-input = {token}

2 token = special-symbol | name | constant | statement-label

3 special-symbol = = | , | (|) | : |
+ | - | * | / | ** |
.LT. | .LE. | .EQ. | .NE. | .GE. | .GT. |
.NOT. | .AND. | .OR. | .EQV. | .NEQV. |
word-symbol

name = letter { letter | digit }

constant = arithmetic-constant | logical-constant |
character-constant

statement-label = digit-string

4 word-symbol = ACCESS | ASSIGN | BACKSPACE | BLANK | BLOCKDATA |
CALL | CHARACTER | CLOSE | COMMON | COMPLEX |
CONTINUE | DATA | DIMENSION | DIRECT | DO |
DOUBLEPRECISION | ELSE | END | ENDFILE | ENTRY |
EQUIVALENCE | ERR | EXIST | EXTERNAL | FILE |
FMT | FORM | FORMAT | FORMATTED | FUNCTION |
GOTO | IF | IMPLICIT | INCLUDE | INQUIRE |
INTEGER | INTRINSIC | IOSTAT | LOGICAL | NAME |
NAMED | NEXTREC | NUMBER | OPEN | OPENED |
PARAMETER | PAUSE | PRINT | PROGRAM | READ |
REAL | REC | RECL | RETURN | REWIND | SAVE |
SEQUENTIAL | STATUS | STOP | SUBROUTINE | TO |
UNFORMATTED | UNIT | WRITE

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

arithmetic-constant = integer-constant | real-constant |
double-precision-constant | complex-constant

logical-constant = .TRUE. | .FALSE.

character-constant = ' string-element {string-element} '

digit-string = digit {digit}

```
5  integer-constant = [sign] unsigned-integer
   real-constant = [sign] unsigned-real
   double-precision-constant = [sign] unsigned-double
   complex-constant = [sign] ( real-part , imaginary-part )
   string-element = string-character ¦ apostrophe-image

6  sign = + ¦ -
   unsigned-integer = decimal-integer ¦ hexadecimal-integer
   unsigned-real = < basic-real [E exponent] ¦ digit-string E exponent
   unsigned-double = <basic-real ¦ digit-string> D exponent
   real-part = real-constant ¦ integer-constant
   imaginary-part = real-constant ¦ integer-constant
   apostrophe-image = ''

7  decimal-integer = digit-string
   hexadecimal-integer = $ hexdigit {hexdigit}
   basic-real = < digit-string . [digit-string] ¦ . digit-string >
   exponent = [sign] digit-string

8  hexdigit = digit ¦ A ¦ B ¦ C ¦ D ¦ E ¦ F
```

A.3 SYNTACTIC ASPECTS

1 compilation-input = program-unit {program-unit}

2 program-unit = main-program ¦ subroutine-subprogram ¦
function-subprogram ¦ block-data-subprogram

3 main-program = [program-statement] program-body

subroutine-subprogram = subroutine-statement program-body

function-subprogram = function-statement program-body

block-data-subprogram = block-data-statement block-data-body

4 program-statement = PROGRAM name

subroutine-statement = SUBROUTINE subroutine-name
[([dummy-argument-list])]

function-statement = [type-specifier] FUNCTION function-name
([dummy-argument-list])

block-data-statement = BLOCK DATA [name]

program-body = specifications
definitions
executable-part
end-statement

block-data-body = block-data-specifications
block-data-definitions
end-statement

5 subroutine-name = name

function-name = name

dummy-argument-list = dummy-argument { , dummy-argument }

type-specifier = < arithmetic-type ¦ logical-type ¦
CHARACTER [* len] >

specifications = { specification-statement ¦
format-statement ¦ entry-statement }

definitions = { statement-function-definition ¦ data-statement ¦
format-statement ¦ entry-statement }

```
executable-part = { executable-statement ¦ data-statement ¦
                    format-statement ¦ entry-statement }

block-data-specifications = { specification-statement }

block-data-definitions = { data-statement }

end-statement = END
```

6

```
dummy-argument = variable-name ¦ array-name ¦ procedure-name ¦ *

arithmetic-type = integer-type ¦ real-type ¦
                  double-precision-type ¦ complex-type

logical-type = LOGICAL [ * <1¦2¦4> ]

len = decimal-integer ¦ ( integer-constant-expression ) ¦ ( * )

specification-statement = type-statement ¦ implicit-statement ¦
                          dimension-statement ¦ common-statement ¦
                          equivalence-statement ¦ save-statement ¦
                          external-statement ¦ intrinsic-statement ¦
                          parameter-statement

format-statement = FORMAT format-specification

entry-statement = ENTRY name [ ( [dummy-argument-list] ) ]

statement-function-definition = function-name ( [stf-argument-list] )
                                = expression

data-statement = DATA data-initialisation
                 { [ , ] data-initialisation }

executable-statement = assignment-statement ¦ control-statement ¦
                       input-output-statement
```

7

```
variable-name = name

array-name = name

procedure-name = subroutine-name ¦ function-name

integer-type = INTEGER [ * <1¦2¦4> ]

real-type = REAL [ *4 ]

double-precision-type = < DOUBLEPRECISION ¦ REAL*8 >

complex-type = COMPLEX [ *8 ]
```

```
integer-constant-expression = constant-expression

type-statement = < non-character-type-statement ¦
                   CHARACTER [ * len [,] ] character-item-list >

implicit-statement = IMPLICIT implicit-declaration
                              {, implicit-declaration}

dimension-statement = DIMENSION array-declarator
                                {, array-declarator}

common-statement = COMMON [common-block]  common-item-list
                          { [,] common-block  common-item-list }

equivalence-statement = EQUIVALENCE equivalence-group
                                    {, equivalence-group}

save-statement = SAVE [ save-item { , save-item } ]

external-statement = EXTERNAL procedure-name {, procedure-name }

intrinsic-statement = INTRINSIC function-name { , function-name }

parameter-statement = PARAMETER ( param-item { , param-item } )

format-specification = ( [format-list] )

stf-argument-list = variable-name { , variable-name }

expression = arithmetic-expression ¦ logical-expression ¦
             character-expression

data-initialisation = variable-list / constant-list /

assignment-statement = arithmetic-assignment ¦ logical-assignment
                       character-assignment ¦ label-assignment

control-statement = goto-statement ¦ arithmetic-if-statement ¦
                    logical-if-statement ¦ block-if-statement ¦
                    else-if-statement ¦ else-statement ¦
                    end-if-statement ¦ end-statement ¦
                    call-statement ¦ return-statement ¦
                    pause-statement ¦ stop-statement ¦
                    continue-statement ¦ do-statement

input-output-statement = read-statement ¦ write-statement ¦
                         print-statement ¦ backspace-statement ¦
                         endfile-statement ¦ rewind-statement ¦
                         open-statement ¦ close-statement ¦
                         inquire-statement
```

8 constant-expression = expression

```
non-character-type-statement = < arithmetic-type ¦ logical-type >
                                typed-item { , typed-item }

character-item-list = character-item { , character-item }

implicit-declaration = type-specifier
                        ( implicit-item {, implicit-item} )

array-declarator =
        array-name ( subscript-bounds {, subscript-bounds} )

common-block = / [block-name] /

common-item-list = common-item {, common-item }

equivalence-group = ( equiv-item , equiv-item {, equiv-item} )

save-item = /block-name/ ¦ variable-name ¦ array-name

param-item = constant-name = constant-expression

format-list = format-item { , format-item }

arithmetic-expression = [sign] arithmetic-term
                        { <+¦-> arithmetic-term }

logical-expression = logical-disjunct
                     { <.EQV.¦.NEQV.> logical-disjunct }

character-expression = character-primary { // character-primary }

variable-list = initialised-item {, initialised-item }

constant-list = initial-setting {, initial-setting}

arithmetic-assignment = variable-element = arithmetic-expression

logical-assignment = variable-element = logical-expression

character-assignment = character-field = character-expression

label-assignment = ASSIGN statement-label TO integer-variable

goto-statement =
        unconditional-goto ¦ assigned-goto ¦ computed-goto
```

```
arithmetic-if-statement =
     IF ( arithmetic-expression ) statement-label ,
                                  statement-label , statement-label

logical-if-statement =
             IF ( logical-expression ) executable-statement

block-if-statement = IF ( logical-expression) THEN

else-if-statement = ELSE IF ( logical-expression ) THEN

else-statement = ELSE

end-if-statement = END IF

call-statement = CALL subroutine-name [ ( [actual-argument-list] ) ]

return-statement = RETURN [ integer-expression ]

pause-statement = PAUSE [ digit-string ¦ character-constant ]

stop-statement = STOP [ digit-string ¦ character-constant ]

continue-statement = CONTINUE

do-statement = DO statement-label [ , ] do-control

read-statement = < READ ( read-write-control ) [io-list] ¦
                   READ format-identifier [, io-list] >

write-statement = WRITE ( read-write-control ) [io-list]

print-statement = PRINT format-identifier [, io-list]

backspace-statement = < BACKSPACE unit-identifier ¦
                        BACKSPACE ( pos-control ) >

endfile-statement = < ENDFILE unit-identifier ¦
                      ENDFILE ( pos-control ) >

rewind-statement = < REWIND unit-identifier ¦
                     REWIND ( pos-control ) >

open-statement = OPEN ( open-control )

close-statement = CLOSE ( close-control )

inquire-statement = INQUIRE ( inquire-control )
```

```
9  typed-item = variable-name ¦ array-name ¦ array-declarator ¦
                constant-name ¦ function-name

   character-item = typed-item [ * len ]

   implicit-item = letter [- letter]

   subscript-bounds = [ lower-bound : ] upper-bound

   block-name = name

   common-item = variable-name ¦ array-name ¦ array-declarator

   equiv-item = variable-element ¦ substring ¦ array-name

   constant-name = name

   format-item = [repeat-count] repeatable-descriptor ¦
                 nonrepeatable-descriptor ¦
                 [repeat-count] ( format-list )

   arithmetic-term = arithmetic-factor { < */ > arithmetic-factor

   logical-disjunct = logical-term { .OR. logical-term }

   character-primary = variable-element ¦ substring ¦
                       function-reference ¦ character-constant ¦
                       constant-name ¦ ( character-expression )

   initialised-item = variable-element ¦ substring ¦
                      array-name ¦ data-implied-do

   initial-setting = [ <unsigned-integer ¦ constant-name> * ]
                     < constant ¦ constant-name >

   variable-element = variable-name ¦ array-element

   character-field = variable-element ¦ substring

   integer-variable = variable-name

   unconditional-goto = GOTO statement-label

   assigned-goto = GOTO integer-variable [ [ , ] label-list ]

   computed-goto = GOTO label-list [ , ] integer-expression

   actual-argument-list = actual-argument {, actual-argument }

   integer-expression = expression

   do-control = control-variable = initial-value ,
                               terminal-value [ , increment-value ]
```

```
io-list = io-item {, io-item}

read-write-control = unit-specifier
                     [, format-specifier]
                     [, REC = integer-expression]
                     [, IOSTAT = integer-variable-element]
                     [, END = statement-label]
                     [, ERR = statement-label]

format-identifier = statement-label ¦ integer-variable ¦
                    array-name ¦ character-expression ¦ *

unit-identifier = integer-expression ¦ * ¦
                  array-name ¦ character-field

pos-control = unit-specifier
              [, IOSTAT = integer-variable-element]
              [, ERR = statement-label]

open-control = unit-specifier
               [, IOSTAT = integer-variable-element]
               [, RECL = integer-expression]
               [, FILE = character-expression]
               [, STATUS = character-expression]
               [, ACCESS = character-expression]
               [, FORM = character-expression]
               [, BLANK = character-expression]
               [, ERR = statement-label]

close-control = unit-specifier
                [, IOSTAT = integer-variable-element]
                [, STATUS = character-expression]
                [, ERR = statement-label]

inquire-control = < unit-specifier ¦ FILE = character-expression
                  [, IOSTAT = integer-variable-element]
                  [, RECL = integer-variable-element]
                  [, NEXTREC = integer-variable-element]
                  [, NUMBER = integer-variable-element]
                  [, EXIST = logical-variable-element]
                  [, OPENED = logical-variable-element]
                  [, NAMED = logical-variable-element]
                  [, NAME = character-variable-element]
                  [, ACCESS = character-variable-element]
                  [, SEQUENTIAL = character-variable-element]
                  [, DIRECT = character-variable-element]
                  [, FORM = character-variable-element]
                  [, FORMATTED = character-variable-element]
                  [, UNFORMATTED = character-variable-element]
                  [, BLANK = character-variable-element]
                  [, ERR = statement-label]
```

```
10  lower-bound = integer-expression

    upper-bound = integer-expression | *

    substring = variable-element ( [substring-expression] :
                                   [substring-expression]  )

    repeat-count = decimal-integer

    repeatable-descriptor = D-descriptor | E-descriptor |
                            F-descriptor | G-descriptor |
                            I-descriptor | L-descriptor |
                            A-descriptor

    nonrepeatable-descriptor = apostrophe-descriptor | H-descriptor |
                               T-descriptor | X-descriptor |
                               slash-descriptor | colon-descriptor |
                               S-descriptor | P-descriptor |
                               B-descriptor

    arithmetic-factor = arithmetic-primary [ ** arithmetic-factor ]

    logical-term = logical-factor { .AND. logical-factor }

    function-reference = function-name ( [actual-argument-list] )

    data-implied-do = ( implied-do-item {, implied-do-item }
                        , do-control )

    array-element = array-name ( subscript {, subscript} )

    label-list = ( statement-label {, statement-label} )

    actual-argument = expression | variable-element | array-name |
                      procedure-name | alternate-return-specifier

    control-variable = variable-name

    initial-value = arithmetic-expression

    terminal-value = arithmetic-expression

    increment-value = arithmetic-expression

    io-item = io-element | io-implied-do

    unit-specifier = [ UNIT = ] unit-identifier

    format-specifier = [ FMT = ] format-identifier

    integer-variable-element = variable-element

    logical-variable-element = variable-element

    character-variable-element = variable-element
```

11 substring-expression = integer-expression

D-descriptor = D w . d

E-descriptor = E w . d [E e]

F-descriptor = F w . d

G-descriptor = G w . d [E e]

I-descriptor = I w [. m]

L-descriptor = L w

A-descriptor = A [w]

apostrophe-descriptor = character-constant

H-descriptor = n H {string-character}

T-descriptor = < T | TL | TR > c

X-descriptor = n X

slash-descriptor = /

colon-descriptor = :

S-descriptor = S | SP | SS

P-descriptor = k P

B-descriptor = BN | BZ

arithmetic-primary = variable-element | function-reference |
arithmetic-constant | constant-name |
(arithmetic-expression)

logical-factor = [.NOT.] logical-primary

implied-do-item = array-element | data-implied-do

subscript = integer-expression

alternate-return-specifier = * statement-label

io-element = variable-element | array-name | substring | expression

io-implied-do = (io-list , do-control)

```
12  c = decimal-integer

    d = decimal-integer

    e = decimal-integer

    k = [sign] decimal-integer

    m = decimal-integer

    n = decimal-integer

    w = decimal-integer

    logical-primary = variable-element ¦ function-reference ¦
                      logical-constant ¦ constant-name ¦
                      relational-expression ¦ ( logical-expression )


13  relational-expression = arithmetic-relational-expression ¦
                            character-relational-expression


14  arithmetic-relational-expression = arithmetic-expression
                                       rel-op arithmetic-expression

    character-relational-expression = character-expression
                                      rel-op character-expression


15  rel-op = .LT. ¦ .LE. ¦ .EQ. ¦ .NE. ¦ .GE. ¦ .GT.
```

B COMPILE-TIME ERRORS

For each error number, the text which is printed at compile time (provided the file PROFOR_ERR is present) is given, plus extra explanation where necessary.

Number	Meaning
002	No path: statement can never be executed Is preceded by an unconditional jump
003	Illegal STOP/PAUSE string
005	Procedure call: varying number of arguments
006	Type already specified
007	DO loop has zero iteration count
008	COMMON: character and non-character items mixed
009	EQUIVALENCE: character and non-character items mixed
010	DATA: non-character item with character constant
011	Array declared with more than 7 dimensions
012	IMPLICIT: same letter specified again
013	Continuation line: first 5 columns not blank
098	Undeclared variable See Part III, section 2.2.7
100	Non-Fortran source character encountered
101	Statement type recognized but mis-spelt
102	Invalid character in label field
103	Logical constant mis-spelt .TRUE. or .FALSE. expected
104	ASSIGN statement: 'TO' mis-spelt
105	Unlabelled FORMAT statement
106	Missing digit(s) in format descriptor e.g. E.7

107 Improper zero in format descriptor
e.g. E0.7

108 Unsigned number required

109 Illegal descriptor after scale factor
e.g. 2PI5

110 Repeat specification not allowed

111 FORMAT: parenthesis nesting depth exceeds 7

112 Attempted EQUIVALENCE of items in COMMON

114 Item recurs in EQUIVALENCE group
e.g. EQUIVALENCE (I, J, I,...)

115 Continuation line illegally positioned

119 Illegal "*" type
e.g. REAL*3

122 IMPLICIT: range of letters not sensible

125 Range error

130 More than one main program encountered

201 Item declared in COMMON more than once

202 Common-block/procedure name clash

203 DATA: item is in COMMON
Initialisation must be done in BLOCK DATA subprogram

204 DATA: item is not in named COMMON
Initialisation of blank common is not allowed

205 Illegal program or ENTRY name

206 Nested DOs with same control variable

207 Illegal type of DO loop control variable
Must be integer-type or real or double precision

208 Illegal type of DO expression

211 DO increment must be non-zero

212 Bracketed i/o list has no implied DO

213 Statement not permitted as DO terminator
Terminal statement may not be arithmetic IF, etc.

214 DO-loop unterminated at END statement

215 DATA/EQUIVALENCE: subscript count wrong

217 EQUIVALENCE group only has one element
(must have at least two)

218 Integer constant outside INTEGER*1 range

219 Integer constant outside INTEGER*2 range

220 Integer constant outside INTEGER*4 range

221 Floating-point constant out of range

222 Illegal type with // operator
e.g. character // integer

223 Illegal type mixture with COMPLEX
e.g. integer * complex

224 COMPLEX expression in arithmetic IF

225 COMPLEX operand in relational expression

226 Statement label referenced but undefined

227 Statement label already defined

228 Statement label was not on FORMAT statement

229 Statement label was not on executable statement

230 Statement label was on specification statement

231 Statement label and statement type are inconsistent

232 Type of variable must be INTEGER*4

233 Type of variable must be LOGICAL*4

234 Variable or array-element expected

235 Illegal use of logical type
e.g. arithmetic expression assigned to logical variable

236 Illegal use of arithmetic type

237 Adjustable dimension is not integer type

238 Statement function: duplicate dummy argument name

239 Statement function call: wrong number of arguments

240 Control-list item illegal in this context
e.g. END = in an OPEN statement

241 OPEN, READ etc: control-list item specified twice

242 OPEN etc: invalid character constant

243 Character expression expected

244 External procedure reference invalid

245 Illegal argument type for intrinsic function

246 Illegal occurrence of unsubscripted array name

249 Character constant has zero length

250 Character constant expression longer than 1327 characters

251 Character expression longer than 32767 characters

253 Block IF: missing ENDIF

255 Arithmetic expression expected

256 Constant expression expected

257 Error in specification of intrinsic function

258 Integer-type expression expected here

259 Illegal constant substring expression

260 Substring not permitted here

261 Illegal use of CHARACTER*(*)

262 ENTRY: dummy argument has already been referenced

263 ENTRY has wrong type

264 RETURN not allowed in main program

265 CALL: name previously used as function

266 Function name previously used as subroutine

267 Alternate return specifier only allowed for subroutines

268 Attempt to access assumed-size dimension

269 ENTRY name has already been referenced

270 Illegally positioned ENTRY statement

272 Too many COMMON blocks in program unit
More than 128

273 Alignment error
Items bigger than 1 byte in size must be word-aligned

300 Character expected but not found
The expected character is printed on the next line

301 Invalid operator beginning with '.'
e.g. .NP.

302 Operator not preceded by an operand
e.g. I = * K

303 DATA: implied DO variable expected

304 DATA: invalid repeat count

305 DATA: error in constants list

306 DATA: too few constants in list

307 DATA: too many constants in list

308 Logical IF: illegal dependent statement
e.g. another logical IF

309 Block IF: ENDIF/ELSE has no matching IF

310 Block IF: nesting depth exceeds 10

311 DO statement: label already defined

312 DO-type statement does not start with DO
e.g. D 10 I = 1, 20

313 Illegal expression type with relational operator

314 Arithmetic expression expected

316 Logical expression expected

318 Exponent expected, beginning with D or E

319 Bad exponent in floating-point constant
e.g. 1.2ED6

320 Error in COMPLEX constant

321 Character constant longer than 255 characters

322 Character constant not terminated
(at end of statement)

323 Array declarator: invalid array name

324 Adjustable/assumed-size declarator: array is not dummy

325 Array element: too few subscripts

326 Array element: too many subscripts

327 Array element not allowed here

328 Function call not allowed here

329 Function call: missing '('

331 Improper dummy argument name

332 Variable name missing

333 Variable/array/procedure name missing

334 Non-dummy variable/array name required
May not be a dummy argument

335 Improper name for statement function

336 Unidentifiable statement type
e.g. CUMMON ...

337 Statement type not allowed in BLOCK DATA
e.g. FORMAT, EXTERNAL, RETURN, etc.

338 Statement incorrectly terminated

339 Statement type out of sequence

340 Type-statement begins incorrectly

341 Expression incorrectly terminated

342 Statement label expected

343 Statement label has more than 5 digits

344 CALL: invalid subroutine name

345 Illegal actual argument

346 Illegal character in format specifier

347 Missing '(' in format specifier

348 Missing ')' in format specifier

349 In nH (Hollerith) descriptor, n not in range 1 thru 255

351 Unit specifier missing or incorrect

352 Format specifier missing or incorrect

353 OPEN, READ etc: control-list keyword expected

354 READ etc: incorrect DO-implied list

355 READ: illegal item in i/o list

358 IMPLICIT: INTEGER etc. expected

359 IMPLICIT: letter expected but not found

361 INCLUDE: filename invalid or not found

362 INCLUDE not allowed in an INCLUDEd file

369 Duplicate SUBROUTINE/FUNCTION/ENTRY name

370 Code for program unit exceeds 32K bytes

380 INTEGER*4 range exceeded
in compile-time computation of subscripts, etc.

402 End-of-file encountered on source input
May be due to missing END statement

403 Compiler stack size insufficient
May be due to a particularly complex source expression, when using a compiler configured for minimum memory use

404 Compiler workfile contents invalid
Should not normally occur, and may indicate a compiler malfunction

405 Compiler workspace insufficient
Insufficient memory available to the compiler.
Memory may have been 'lost' to the system by programs run before the compiler. Re-booting the machine may recover such memory. Otherwise, smaller source programs or more memory are the only solutions.

C RUN-TIME ERROR CODES

The format of the messages produced for run-time errors is given in Part III under "Operation of object programs". This appendix lists the error codes, with significance and possible causes.

Code Meaning

A Angle argument error.

From SIN, DSIN, COS or DCOS when the argument is so large that range reduction would lead to serious loss of accuracy.
REAL argument: abs(value) > 32768.0
DOUBLE PRECISION argument: abs(value) > 4.295D9

B Bounds exceeded.

A subscript bound has been exceeded (with /I compile-time option selected).

D Disc or Device error.

Disc or directory space insufficient for output. Attempt to read from an output device, or write to an input device. Attempt to read a fixed-length file with wrong "RECL=" parameter.

E Stack error on exit.

The stack pointer is not correct on exit from a subprogram. Possibly because an incorrect Assembler-coded routine has been called.

F File programming error.

Error in file usage. Message displays status value and unit number - see separate list of status codes below.

J Divide error (integers).

In I/J or MOD(I,J), J is zero. Continuation possible, but results not predictable.

K Overflow on type conversion.

Conversion of a real or double-precision value to integer gives a value outside integer range.

L Log argument error.

Argument to ALOG, ALOG10, DLOG or DLOG10 is zero or negative.

I/O status values

Following are the values which will be given in the reply when an error occurs and "IOSTAT=" is specified, or in the error message with error F.

If an error message is produced (i.e. no ERR= or IOSTAT=) the status value is shown with the statement type (e.g. "READ") plus either the unit number, or "*" for standard i/o, or "Internal" for an internal file. Also, if a message is produced, status values less than 10 are treated as recoverable (though the result of a READ operation may not be predictable), while values of 10 or more cause termination of the program.

Value	Meaning
01	Missing or illegal numeric, illegal zero, field width > 255, nesting of () exceeds 7, w < d in w.d type descriptor, repeat count exceeds 32767, tab out of range, m > w in Iw.m descriptor.
02	Separator missing in a format.
03	Format does not begin with "(", or unbalanced parentheses, or format is all blanks, or format is terminated improperly.
04	Illegal character in a format.
06	Illegal character in a numeric input field, or sign but no digits to follow.
07	Record overflows line buffer (or array if internal file)
08	Incorrect format of real or double precision input.
09	In list-directed input, either a character literal or a complex value is not separated from the next item.
10	Formatted and unformatted operations on same unit.
11	Input and output operations on same unit.
12	End-of-file encountered, no END= or IOSTAT= specified.
13	End-of-file previously encountered on this unit.
14	Unformatted transfer attempted to/from device.
15	Incorrectly formed character expression in OPEN statement.

16 Attempting direct-access READ or WRITE, but OPEN has not specified RECL; or attempting sequential-access READ or WRITE, or an ENDFILE operation, but OPEN has specified RECL.

17 Wrong direction of transfer for device.

18 Negative unit number given.

19 Input file cannot be opened.

20 In list-directed input, a repeat count is zero.

21 In list-directed input involving a non-null item with a repeat count, not all the corresponding list items have the correct data type for the item.

22 In list-directed input, a complex datum is not in the correct format.

23 In list-directed input, a character datum is not in the correct format (doesn't start with quote, or zero-length).

24 In list-directed input, a character datum exceeds 255 characters.

25 In formatted READ, format contains 'xxx' literal or nH descriptor.

26 In formatted READ, record is too short for list.

27 In formatted or list-directed READ with integer list item, the input value exceeds the maximum value associated with the given variable.

28 In formatted or list-directed READ with logical list item, datum is invalid.

29 In formatted or list-directed input, a real or double-precision value exceeds the maximum possible (could be part of a complex value).

30 In formatted I/O, no descriptor for list item.

31 In formatted I/O with integer list item, descriptor is not I.

32 In formatted I/O with real or complex list item, descriptor is not one of D, E, F or G.

33 In formatted I/O with double precision list item, descriptor is not D, E, F or G.

34 In formatted I/O with logical list item, descriptor is not L.

35 In formatted I/O with character list item, descriptor is not A.

36 In formatted WRITE to an internal file, more records were created than the file contained.

40 In unformatted READ, record is too short for input list.

41 In unformatted WRITE, a (fixed-length) record is too short for output list.

44 In an OPEN statement, the RECL value is too large.

45 In an OPEN statement, contradictory options have been given.

47 In a READ or WRITE statement, REC= specifies an invalid record number.

50 Unit does not exist.

51 Unit is not connected.

52 The program has too many open files.

53 The size of an existing file opened for direct access is not an exact multiple of the specified RECL value.

54 An attempt has been made to READ a file opened with STATUS='NEW' or STATUS='SCRATCH'.

55 An attempt has been made to CLOSE a file with STATUS='KEEP' when the OPEN specified STATUS='SCRATCH'. The workfile will, however, be kept.

56 An OPEN statement with STATUS='NEW' names a file that already exists.

57 An attempt has been made to execute a BACKSPACE or REWIND statement on a file not connected for sequential access.

58 BACKSPACE on a file that does not exist.

90 In CLOSE, the chain of FCA's is corrupt (internal library error).

1000+n QDOS error -n has occurred. The corresponding QDOS error message is also given.

D ASCII CHARACTER SET

Hex	Character	Hex	Character	Hex	Character	Hex	Character
00	NUL	20	space	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(	48	H	68	h
09	HT	29	)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[	7B	{
1C	FS	3C	<	5C	\	7C	|
1D	GS	3D	=	5D	]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL

E MIXED LANGUAGE PROGRAMMING

Program construction

Mixed language programs may be constructed by amalgamating Pro Fortran-77 and Pro Pascal components. Pascal segments can be included in a Fortran program, or Fortran subprograms can be included in a Pascal program.

Input/output may be performed in both languages independently. Only one special rule applies here: in the case of a Fortran main program with Pascal procedures, the standard files "input" and "output" will not have been implicitly assigned to the console, and hence they must be assigned explicitly and a reset or rewrite given before they are used.

When a mixed language program terminates, either normally, or through a run-time error, all open Pascal and Fortran files are closed.

After compilation by the appropriate compiler, the components are link-edited together (refer to Part III). In the linker command, the _REL files are listed in sequence (with the main program module typically coming first), followed by the names of the two libraries to be selectively scanned, with the library for the main program language coming first.

Correspondence of data types

The table below shows the correspondence between Fortran and Pascal data types.

Fortran	Pascal
INTEGER	integer
INTEGER*2	-32768..32767
INTEGER*1	-128..127
REAL	real
DOUBLE PRECISION	longreal
COMPLEX	RECORD realpart,imagpart: real; END
LOGICAL*1	boolean
CHARACTER * n	PACKED ARRAY [1..n] OF char

Single variables of equivalent type may be associated (for example by the COMMON facility) and referenced from either language.

Arrays may also be referenced from either language, but if an array has more than one dimension it is important to notice that Fortran stores in "column major" order (as given by the "subscript value" function) and Pascal in "row major" order. The sequence of subscripts/indices must therefore be reversed when changing from one language to the other.

COMMON

The COMMON facility can be used within mixed language programs with the understanding that when a Pascal variable is declared in COMMON the variable name becomes a common block name. The significance of this may be seen in the example below.

```
Fortran                  Pascal

COMMON /CB/ A,B,C        COMMON cb: RECORD
                                      a,b,c: real;
                                 END

A = 5.5                  cb.a := 5.5;
```

Here the two declarations each describe a common block containing three real variables. The suggested form of declaration of cb as a record provides the equivalent separation of the common block name from the names of the variables within it. However, if the block contains just one component another method is to give the block and the contents the same name in Fortran:

```
Fortran                  Pascal

COMMON /X/ X(120)        COMMON  x: ARRAY [1..120] OF real;
```

Parameters

The name of a Pascal procedure at the outer level can be quoted in a Fortran CALL statement, or a Fortran subroutine can be given an EXTERNAL declaration within Pascal and then referenced. Pascal and Fortran functions are similarly equivalent.

The Pascal parameters and the Fortran arguments must match in number, order, and type (see above). All Pascal parameters must be VAR parameters.

Note that in the case of a CHARACTER argument, Fortran does not pass the address of the start of the data (as would be done in Pascal for a PACKED ARRAY [ ] OF char), but the address of a 6-byte Character Variable Descriptor (CVD). This is structured like a Pascal record consisting of a 4-byte address field followed by a 2-byte length field.

A Fortran subroutine or function which has a subroutine or function as a dummy argument can be called from Pascal, but because Pascal passes two addresses in such cases (the entry address and the static link), the Fortran must include an extra dummy argument to match the latter. For example

```
Pascal      FUNCTION area (PROCEDURE calc): real; EXTERNAL;

Fortran     REAL FUNCTION AREA (DUMMY,CALC)
```

The Pascal procedure passed as an actual corresponding to calc must be declared at the outer level.

Interchange of files

A Pro Fortran-77 file of variable-length formatted records has the same layout as a Pro Pascal file of type text. (Both, in fact, are normal QDOS text files.)

A Pro Fortran-77 file of fixed-length records has the same layout as a non-text file in Pro Pascal. To exchange binary files, it is necessary to know the file element size implied by the Pascal file declaration, since this must be quoted explicitly in the RECL= parameter of the Fortran OPEN statement. Refer to the user manual for sizes of the various data types. Correspondence between data layouts within the file elements is similar to that for COMMON variables (see above), bearing in mind that a Fortran unformatted READ or WRITE simply copies the data items in the iolist between memory and file. Thus for example a file written by a Pascal program as

```
FILE OF RECORD
            item: integer;
            vmax,vmin: real;
        END;
```

would be read in Fortran by

```
OPEN (5, RECL=12, ...)
...
READ (5) ITEM,VMAX,VMIN
```

There is no equivalent in Pro Pascal to a Pro Fortran-77 file of variable-length unformatted records.

Prospero Software

LANGUAGES FOR MICROCOMPUTER PROFESSIONALS

Pro Fortran-77 Version mmq 1.1.7 - extra remarks.

Multi-tasking on the QL

Users new to the QL should be aware that, when more than one task is active in the QL, as for example when the Fortran-77 compiler or a Fortran-77 object program are activated from the SuperBASIC command line, then more than one window is in general open on the screen, and more than one can be awaiting user input from the keyboard. When this is the case, the user should use <CTRL-C>, if necessary, to switch among active tasks until the cursor flashes (awaiting input) in the required window.

Use of the configuration programs

The User Manual (Part III, section 5) describes three configuration programs: SETDDEV, NOQNS and SETSTACK. It is important to use them only as directed in the Manual; in particular, they must not be used on the Linker (LINK).

Altering the default program window

All the supplied Prospero executable programs, except the GST Linker, and all the user executable programs created using the supplied softwawre use a default program window. The user may, if desired, patch such an executable program so as to reconfigure the program window. The window definition is stored at a fixed offset in all the programs, as follows:

Hex. offset	Bytes	Contents	Default Values
2C	1	Border colour	Green (04 hex)
2D	1	Border width	1 (01 Hex)
2E	1	Paper colour	Black (00 hex)
2F	1	Ink colour	Green (04 hex)
30	2	Window width	486 (01E6 hex)
32	2	Window height	150 (0096 hex)
34	2	X-origin	12 (000C hex)
36	2	Y-origin	16 (0010 hex)

N.B. The LINK program must not be patched, as it does not conform to the above conventions.

Additional sample programs

This issue includes some source (_FOR) files containing sample Fortran programs not listed in the manual.

1. MAZES_FOR

This program will print out a different maze every time it is run (and guarantees only one path through). It illustrates use of several Fortran features: logical and character variables and arrays, statement functions, DATA statements with character constants, Block IFs, computed GOTOs, and the random number generator (RANDOM). It is not a very "typical" example of Fortran coding style, but indicates that Fortran can be used to solve logical as well as number-crunching problems.

2. SQUARES_FOR

This is a simple illustration of the use of the graphics library routines, and displays squares of randomly varying sizes and colours on the screen.

3. TRAPDEMO_FOR

This is a program to show how QDOS traps can be called from within a Pro Fortran-77 program. It issues the MT.MODE manager trap to dtermine the display mode.

Integrity of issued files

The Pro Fortran-77 software is normally supplied on write-protected microdrive cartridges. Do not write-permit the master copy or execute the compiler from it. Copy the files from the supplied cartridges onto your own cartridges. You may then verify that the working copy is correct (see below), and go on the compile, link, and run your own programs as described in Part III of the User Manual.

Checking validity of files.

The program FCHECK is supplied to guard against copying errors in the issued software. To run the program, ensure that the PRL ROM is in place, then type:

```
EXEC xxxx_FCHECK
```

where xxxx is the device containing the program FCHECK. The program asks for a device name, such as MDV1 or FLP2. By just pressing the <ENTER> key, the default device MDV1 will be selected. The program then sumchecks all the files on that device whose names it recognises. These sumchecks are compared against information recorded within itself (the correct sumchecks) and either "OK" or an error message is given, for each file.