DIGITAL C

BY

G. JACKSON

Manual by

Dr. Helmut Aigner & Dr. Andy Aigner



Copyright 1988 Digital Precision Ltd.
Published by F. A. Vachha for DP, 222 The Avenue, London E4 9SE

C O N T E N T S

1. INTRODUCTION TO DIGITAL C
1.1 What You Got for Your Money
1.2 Getting Started
1.3 Copyright Notice

2. COMPILING AND RUNNING DIGITAL C PROGRAMS
2.1 Source Programs
2.2 Parser
2.3 Code Generator/Linker
2.4 Library Generator
2.5 Program Execution
2.5.1 General rules
2.5.2 Parser command line
2.5.3 Code Generator/Linker command line
2.5.4 Library Generator command line
2.5.5 Input/output redirection and pipes
2.5.6 The sample program sort
2.6 The Configurator Program

3. A THUMBNAIL DIGITAL C TUTORIAL
3.1 A Short Program
3.2 Getting Bolder
3.3 Flexing Digital C's Muscles
3.4 Striking Out on Your Own

4. A THUMBNAIL DIGITAL C REFERENCE
4.1 Limitations
4.2 Program Line Organisation
4.3 Comments
4.4 Constants
4.5 Variable Types
4.6 Initialisation
4.7 Scope of Variables
4.8 Operators
4.8.1 Unary operators
4.8.2 Binary operators
4.9 Expressions
4.10 Statements
4.11 Keywords
4.11.1 The if ... else statement
4.11.2 The while statement
4.11.3 The do ... while statement
4.11.4 The for statement
4.11.5 The switch statement
4.11.6 The break statement
4.11.7 The continue statement
4.11.8 The goto statement
4.11.9 The return statement
4.12 Functions
4.12.1 Function definition
4.12.2 Function calls
4.12.3 Pointers to functions
4.12.4 Function argument counts
4.13 The Preprocessor
4.14 Standard Definitions
4.15 Standard DIGITAL C Channels
4.16 Libraries
4.16.1 Standard C library functions
4.16.2 QDOS functions
4.16.3 Floating-point library
4.17 Errors
4.17.1 How to interpret error messages
4.17.2 Parser error messages
4.17.3 Code Generator/Linker error messages
4.17.4 Library Generator error messages

1. INTRODUCTION TO DIGITAL C

1.1 What You Got for Your Money

DIGITAL C is an implementation of the Small-C compiler originally published in Dr. Dobbs' Journal. This compiler, which was originally written to run on Z80 and 8080-based systems has been heavily modified to run on the QL or Thor. Included with the compiler proper, which will henceforth be referred to as the Parser to avoid ambiguity, are two other programs specially written for this implementation, a Code Generator/Linker and a Library Generator, as well as some library programs: these two programs are all that we are charging you for.

Chapter 2 of this manual will tell you how to run programs written in DIGITAL C. It is indispensable for all users, including those who can program in C. For those new to the language, Chapters 3 and 4 contain an introduction to programming in C, but this can only skim the surface. For serious study - or if you do not know SuperBASIC reasonably well - you should get a text on programming in C, such as

Boris Allan
Introducing C
William Collins Sons & Co Ltd
£ 9.95
ISBN 0-00-383105-1

Boris Allan
C Programming: Principles and Practice
Paradigm
£ 12.50
ISBN 0-948825-15-4

Kevin Sullivan
The Big Red Book of C
Sigma Press
£ 7.50
ISBN 0-905104-68-4

or the C 'bible' by the creators of the language:

Brian W. Kernighan, Dennis M. Ritchie
The C Programming Language
Prentice-Hall Inc.
ISBN 0-13-110163-3

The DIGITAL C Parser is an integer-only parser handling a subset of the complete C language. A library is provided to handle floating-point arithmetic and QL input/output.

The following files are included in this package:

cc	Parser
cg	Code Generator/Linker
lg	Library Generator
mc_obj	Machine-code module
std_lib	Standard library
stdio_h	Standard definitions
updates_doc	Updates to the manual, to be read with Quill - only present if needed
config	Change the default device, alter the screen characteristics
sieve_c	Sample program
sort_c	Sample program
clone	Backup facility

1.2 Getting Started

Do curb your impatience to get DIGITAL C up and running right away. Media have been known to become faulty, and files to be erased by mistake. Murphy's Law decidedly states that it will happen to YOU if you fail to make a backup copy of your DIGITAL C medium.

To make a backup copy, put the master disk or cartridge in drive 1 and examine your blank disk or cartridge, checking whether it is write-protected; if necessary, unprotect it. All kinds of trouble can result when an attempt is made to output to a write-protected microdrive (this is an unfixable QL bug), whereas the same blunder with a disk will merely result in a 'read only' error message. So do use that piece of tape if the write-protect tab is missing. Now insert your blank medium in drive 2 and type

LRUN flp1_clone or LRUN mdv1_clone

You will be prompted for the name of the source device and that of the destination device. When you have answered these questions, you will be asked whether the destination medium needs to be formatted. Reply y or n. When the drives have stopped whirring and the lights gone out, remove your master medium and stow it in a safe place. Only the cloned medium should be used henceforth (except for making new clones in case something happens to the first one). The working copy you have just made is intended to work from drive 1; so put it there now before continuing. If you prefer another default device, you can change this by running the program config (see section 2.6).

Before continuing with DIGITAL C, we strongly recommend that you load the file updates_doc into QUILL and peruse it for stop-press information on improvements not contained in the manual.

1.3 Copyright Notice

DIGITAL C, like the rest of DP's programs, is not protected against unauthorised copying and theft. This has obvious advantages for you, in that you can make backups freely and transfer the program suite freely from one medium to another. But the absence of copy protection has a big disadvantage for us: it means that every time you sell or give away a copy of DIGITAL C to someone, we lose a sale that would have helped us to fund more development of QL software.

There is no QL software publisher so large and committed as DIGITAL PRECISION. But the flow of new QL software from DP, and the availability of older titles, depends upon the honesty of a circle of buyers.

DIGITAL PRECISION will remain active in the QL market for the foreseeable future, but future products can only appear if people are willing to pay for them.

In effect, against the trend - especially on top-quality products - we have taken a gamble on your honesty. Please don't let us down.

We offer rewards for information which enables action (civil and criminal) to be taken against software pirates, big and small. Of course, we hope to spend our time more productively!

2. COMPILING AND RUNNING DIGITAL C PROGRAMS

2.1 Source Programs

The DIGITAL C microdrive or disk contains two sample programs named sieve_c (a sieve-of-Eratosthenes prime-number algorithm) and sort_c (a multi-purpose sorting program). If you wish to add programs copied from a magazine or written by yourself, they must be written in ASCII format. This can be done in QUILL, if you take certain precautions to prevent non-printing characters from going into the file: set left margin to 1, indent margin to 1, right margin to suit your screen display, right justify off and page length to zero (Design command). Use the Print rather than the Save command, and when you are offered the default 'to printer', type in the name of the file (be sure to specify a _c extension, so that the Parser can deal with it). After writing the filename and before pressing Enter, remove the Quill disk or microdrive to make the printer_dat file unavailable. Ignore the 'bad or changed medium' message (for once). Alternatively, if you have a version of QUILL which supports the File Export command, use this.

It would be vastly preferable, though, to use an editor program to write your files, such as Digital Precision's The Editor. This not only makes for faster writing and editing but also greatly simplifies the saving process. Again you must save your programs with a _c extension so that the Parser can deal with them.

2.2 Parser

This section may safely be skipped if you are not interested in the theory but only want to run programs.

The original Small-C compiler was written by J. E. Hendrix and placed in the public domain. A description and source code of the compiler (and many other useful programs) are contained in "Dr. Dobbs' Toolbook of C" published in 1986 by Brady (Prentice Hall), ISBN 0-89303-599-8. This is a source of many useful programs. The DIGITAL C Parser takes C source code and produces an intermediate-code file. The original compiler produced output suitable for a conventional assembler and linker.

The DIGITAL C Parser produces a numeric intermediate-code file which is used by the Code Generator/Linker. This approach was taken because it is much faster: e.g., with the original system it took the compiler about 12 minutes to compile itself and a further 13 minutes for an assembler to generate an executable program. With the code-generator approach the Parser takes 6 minutes and the Code Generator 50 seconds. Large programs can be compiled in separate modules and the object modules linked together into an executable program by the Code Generator/Linker. This makes recompilation a much quicker process when programs are modified, since only those modules which have been changed need to be recompiled.

2.3 Code Generator/Linker

The DIGITAL C Code Generator/Linker has been specially written for the QL, using DIGITAL C itself. It takes object-code modules generated by the Parser and library modules produced by the Library Generator, generates 68008 machine code for each module, and links all modules together to produce an executable program which may be EXEC'd in the usual way. Executable programs are limited to 64K (including runtime dataspace) in size; however, later releases (to be produced if justified by demand) may have no size limitations. The Code Generator will tell you if an executable program is too large.

2.4 Library Generator

The Library Generator is another program written in DIGITAL C for the QL. It takes a series of compiled object modules, generated by the Parser, and generates a single library module for use by the Code Generator. This approach is typically used for C functions which may be used by several application programs, because the Code Generator only selects those library functions which are actually used by the object modules and excludes irrelevant (unused) functions from the final executable-code file. The standard library module supplied in the DIGITAL C package was generated by this Library Generator.

2.5 Program Execution

2.5.1 General rules

All three programs (the Parser, the Code Generator/Linker, and the Library Generator) are run by using EXEC or a similar command provided by the various toolkits available. Each program will then request a command line to inform the program about the action to be taken.

The command line will typically contain a mixture of filenames (NOT enclosed in quotes) and command-line switches (recognised by the first character being a -) separated by spaces. A filename is specified either by using the full QDOS name preceded by the device name, e.g. mdv1_xyz_c or by relying on a default device and extension provided by the program, e.g. xyz , or a mixture of the two, e.g. mdv1_xyz or xyz_c . Details of the command lines for the three programs are given in Subsections 2.5.2 to 2.5.4. Programs may optionally use the command-line facility in the Code Generator/Linker (see command-line switch -nc in Subsection 2.5.3).

The default device is set to mdv1_ for all three programs. This may be changed by using:

(a) the supplied program config (see section 2.6)

(b) an editor, such as Digital Precision's The Editor (use unformatted input mode with overstrike).

The string mdv1_ only occurs once in each program.

If you have Digital Precision's TURBO TOOLKIT, you may use one of the commands EXECUTE, EXECUTE_A, EXECUTE_W instead of EXEC. This has the advantage that the option string in these commands may be used to pass the command line to the program. A sequence like the following in a SuperBASIC program can save a lot of time on repeated compilations while a program is being tested:

```
230 EXECUTE_A mdv1_cc;'prog -p -m'
240 EXECUTE_A mdv1_cg;'prog_new prog'
```

A similar facility is offered by the EX commands available on the Thor and in Tony Tebby's Toolkit II.

A further command-line facility, not used by the compiler suite but usable in DIGITAL C programs (e.g. in the example program sort), is input/output redirection, which will be dealt with in Subsection 2.5.5.

2.5.2 Parser command line

The command line for the Parser contains one or more C source-file names with optional command-line switches. The source file must be stored in a file on one of the standard QL devices, e.g. mdv1_, flp2_, ram1_, etc.; and the filename must end with _c or _C , e.g. the supplied program mdv1_sieve_c . Possible command-line switches are:

-p Pause on error: if an error occurs, the Parser waits until ENTER is pressed; if ESC is pressed, the program aborts.

-m Monitor progress: this prints the first line of every C function as it is compiled.

-d/dev Set the default device for C source files: this overrides mdv1_, the default in the compiler suite as supplied, or whatever default device has been configured with the config program.

The first item ('argument') on the command line should be a filename. This is taken for both the input filename (with a _c extension) and the output filename (with an _obj extension). Other switches and filenames may occur in any order. If other filenames are included, they are compiled into the same object module.

For example, to compile sieve_c, any of the following may be typed in as the command line:

```
mdv1_sieve_c -p -m
sieve -m -d/ram1_ -p
sieve
```

All of these will produce an object module called sieve_obj .

The command line

```
fred -p mary_c flp1_joe -d/ram1_
```

will attempt to compile ram1_fred_c , ram1_mary_c and flp1_joe_c, pausing after every error. A single object module ram1_fred_obj will be produced for input to the Code Generator.

With the TURBO Toolkit, it is possible to compile a program that will not have to stop for a command line (say, from RAM disk to RAM disk, with the pause and monitor options) by entering

```
EXECUTE ram1_cc;'ram1_test -p -m'
```

2.5.3 Code Generator/Linker command line

The command line for the Code Generator/Linker contains a mixture of an output filename, object module names, library module names and command-line switches. The first name on the line is taken as the name of the executable task; the others may be in any order, except that, if you have several library modules, earlier ones may call functions in the later ones but not vice versa. If this causes problems, combine library modules.

Object module names must have the _obj extension, which may be omitted on the command line. Library modules must have a _lib extension, which must be included in the command line (the Code Generator uses this to distinguish between object and library modules).

Note that the Code Generator expects to see two of the supplied files on the default device, mc_obj and std_lib. Though these must not be specified on the command line (we want to spare you the drudgery of typing them), the Code Generator automatically searches for them on the default device and will generate an error if they are not found.

Possible command line switches are:

-p	Pause on error: The program waits for ENTER to be pressed
-m	Monitor progress: outputs original source filenames
-d/dev	Set the default device
-nc	No command-line request to be included in the executable program. Omitting this will result in a command request
-c/name	Reads a file for the command line, e.g. -c/ram1_fred will read file ram1_fred_cmd for a command line. The _cmd extension in the command-file name is mandatory. This does not affect any previous switches, but does override any following ones and all filenames.
-sn	(where n is a decimal number in the range 1000 to 65535): Sets the dataspace of the executable program to n; e.g. -s2000 sets the program's run-time dataspace to 2000 bytes. The default is 7000 bytes.

To generate an executable sieve program called sieve_obj, any of the following command lines will do:

```
mdv1_sieve mdv1_sieve_obj
flp1_sieve sieve -p -m -s1000 -nc
sieve sieve
```

If the TURBO Toolkit is available, the same effect, but without the Code Generator/Linker stopping for the command line, can be obtained by the following command:

```
EXECUTE mdv1_cg;'sieve sieve'
```

2.5.4 Library Generator command line

The command line for the Library Generator contains only object-module names. There are no command-line switches, nor should library module names be included. Object-module names must have an _obj extension, which may be omitted on the command line. The output library module will take the first of the names on the command line, but with a _lib rather than an _obj extension. For example, the supplied library module std_lib was produced with this command line:

```
std fp qdos
```

with files std_obj, fp_obj, and qdos_obj on the default device. Note that use of the Library Generator is compulsory even if library generation involves only a single file, because the object code has to be converted into a special format.

2.5.5 Input/Output redirection and pipes

The standard channels stdin and stdout may be changed from the console window by redirection in the command line. This is not used in the compiler suite itself but may be used in DIGITAL C programs (e.g. in the example program sort). Redirection is achieved by preceding the desired filename with < or > or >> , e.g.

<mdv2_fred in a command line will take standard input from a file mdv2_fred. Of course this file must already exist.

>mdv2_joe will send standard output to mdv2_joe after deleting any existing file with that name.

>>mdv2_mary will append standard output to the end of an existing file mdv2_mary .

Compiled programs may also make use of pipes if you have a toolkit (e.g. Digital Precision's TURBO Toolkit) which supports pipes. E.g.,

```
EXECUTE raml_input TO raml_test;'option string' TO raml_output
```

will run task raml_test, set its input stream (stdin), to raml_input, and set its output stream (stdout) to raml_output. The error stream will remain set to the console. Of course the output stream can be piped on to another task.

If input/output redirection is included in the option string (a bit silly but possible), this will take priority over what i/o may have been specified in the EXECUTE command; e.g.,

```
EXECUTE raml_i1 TO raml_test;'<raml_i2 >raml_o2' TO raml_o1
```

causes files raml_i1 and raml_o1 to be opened but ignored by raml_test. They will be closed on task termination, of course, since they are owned by the task.

Note that

```
EXECUTE raml_input TO raml_cc
```

will cause the Parser to read a command line from file raml_input. But

```
EXECUTE raml_input TO raml_cc;'-p -m'
```

will cause the Parser to compile file raml_input. On the other hand,

```
EXECUTE raml_i1 TO raml_cc;'raml_i2 -p -m'
```

will compile raml_i2 and ignore raml_i1. All perfectly logical, we assure you.

Output pipes from the Parser are ignored, since output is always sent to an _obj file.

2.5.6 The sample program sort

As a further example of what can be done with this version of C, an example text-file-sorting program is included. You will have to compile this first with, say, the following series of commands and command lines:

```
                        exec mdv1_cc
with command line:      mdv1_sort -m

                        exec mdv1_cg
with command line:      mdv1_sort mdv1_sort -s20000
```

This will create a task file named mdv1_sort with a dataspace of 20000 bytes, which is used for a sorting buffer. If the file being sorted is larger than this, temporary files are created with the name and on the device specified in the source file mdv1_sort_c. This is set to ram1_sort00.$$$ but can of course be changed as desired if you do not have a RAM disk.

To activate sort, issue the command:

```
                        exec mdv1_sort
with command line:      <mdv1_sort_c -u
```

which takes input from file mdv1_sort_c, sorts the lines, discards duplicate lines and displays the rest on the screen. A command line:

```
                        <mdv1_sort_c >mdv1_ordered -u
```

will do the same but save the results in file mdv1_ordered.

Note that tabs in the source file are significant.

Possible command line switches for the sort program are:

-u Unique: Discard duplicate lines

-d Sort in descending order

-q Use quicksort instead of the default Shell sort

-c# Sort from column number: e.g. -c10 sorts from column 10

-f#? Sort from field number, with character ? delimiting fields: e.g., -f2* means sort on the second field, where asterisks are used to delimit fields. If no delimiter is specified, 'white space' (i.e. space or tab characters) is used as the field delimiter.

2.6 The Configurator Program

The DIGITAL C suite includes a program named config, which will permit you to

(a) change the default device (originally mdv1_ or flp1_), and
(b) alter the characteristics of the screen (window parameters, character size, colours) for the programs cc, cg, lg and config (yes, you can configure the configurator!).

To make any of these changes, place your DIGITAL C medium in drive 1 and type

```
exec mdv1_config
```

The program will ask you a number of questions, supplying in each case (except the filename, which must be cc, cg, lg or config) both a range and a bracketed default value (which you can select by simply pressing Enter). When all questions have been answered, the program displays the requested screen and prompts you either to accept it or to restart.

If you accept the changes, you will be asked for the name of another program to configure. To terminate the program, simply press Enter at this stage.

PROFESSIONAL ASTROLOGER

★ Amazing Astrology system ★ Supplied with 140 A4 page Manual which assumes no knowledge of astrology ★ Gives 10 A4 pages of personality/character delineation ★ Gives 6 A4 pages of day-to-day and year-to-year personalised text predictions ★ Gives 3 A4 pages of text comparisons between two people ★ Massive 300K of user adjustable text files (450K for disk users) supplied together with machine code editor ★ Incredibly fast - 0.5 seconds/computation ★ One minute accuracy this century ★ Exact hardcopy on all options - 16 print modes, user definable astrological glyphs and printer driver ★ Transits ★ Progressions ★ Synastry ★ Choice of 7 house systems ★ Individually adjustable orbs ★ Batch processing mode ★ Full Quill compatability ★ 158 birth data files supplied ★ File compression ★ Output to any device, including files ★ Advanced command language using ANDORNOT and a P0123456789 multiple criteria facility ★ Eclipses ★ Closing aspect indicator ★ Comprehensive user tunable defaults ★ Rectification ★ Extremely user friendly ★ Real-time interpretation trim ★

"The most powerful & complete astrology package on any micro .... provides everything that a present day Nostradamus will need .... the ultimate astrology package .... a 5-Star (☆☆☆☆☆) program – a Sinclair User Classic (the highest award given to any program)" SINCLAIR USER

FEATURES	SUPER ASTROLOGER DE LUXE	PROFESSIONAL ASTROLOGER
AUTOMATIC HOUSE CALCULATION	★	★
TEXT FILES FOR HOUSE INTERPRETATION	31K	70K+
AUTOMATIC SIGN CALCULATION	★	★
TEXT FILES FOR SIGN INTERPRETATION	37K	80K+
AUTOMATIC ASPECTS CALCULATION	★	★
TEXT FILES FOR ASPECTS INTERPRETATION	30K	80K+
GRAPHIC PRINT OF NATAL CHART	Not proportional	Proportional
INTERPRETATION USES ASC + M.C.	–	★
CALCULATION ACCURACY THIS CENTURY	Within 5 mins.	Within 1 min.
USER DEFINABLE ASTROLOGICAL CHARACTER SET	★	★
USER-MODIFIED INTERPRETATION FILES WITH AND/OR LOGIC - FULL SCREEN EDITOR	★	★
USER DEFINABLE PRINTER DRIVER	★	★
OUTPUT TO SCREEN/PRINTER/MICRODRIVE/DISK	★	★
AUTOMATIC PROGRESSION CALCULATION	★	★
TEXT OUTPUT FOR PROGRESSION INTERPRETATION ALLOWING YEAR TO YEAR HOROSCOPES	–	★
AUTOMATIC COMPATIBILITY CALCULATION	★	★
TEXT OUTPUT FOR COMPATIBILITY TESTING ALLOWING AUTOMATIC COMPARISONS BETWEEN INDIVIDUALS	–	★
AUTOMATIC TRANSITS CALCULATION	–	★
TEXT OUTPUT FOR TRANSITS INTERPRETATION ALLOWING DAY TO DAY HOROSCOPES	–	★
USER-SWITCHABLE HOUSE SYSTEM	–	★ (7 defaults)
USER-DEFINABLE ASPECT ORBS	–	★
TEXT & DEFAULTS EDITOR PROGRAMS	SuperBASIC	High speed M/code
"NOT" FACILITY IN INTERPRETATIONS	–	★
SELECTIVE "ANYTHING" FACILITY (EG: ANYTHING IN THIRD HOUSE BIQUINTILE ANYTHING IN ARIES) IN INTERPRETATIONS	–	★
BATCH PRINTING/PROCESSING	–	★
NARROWING/WIDENING/STATIONARY ASPECT INDICATOR	–	★
SAMPLE BIRTH DATA FILES	20	150+
BIRTH DATA FILE COMPRESSION	–	★
FULL QUILL COMPATIBILITY (TO/FROM)	–	★
AUTOMATIC ESC/RETRY/ABORT FILE FACILITY	–	★
PRINTER CONTROL CODE DEFAULTS	–	2 (long) sets
PRINT MODES	2	4
SEPARATE ORB DEFINE FOR NATAL/TRANSITS PROGRESSIONS & COMPARISONS	–	★

"Digital Precision have achieved the impossible with Professional Astrologer. Descriptions such as superb, ultimate, excellent are descriptions which are only barely adequate" ... QL WORLD/QL USER

£59.95 COMPLETE WITH HUGE MANUAL, OR £69.95 WITH ASTRONOMER TOO

PROFESSIONAL ASTRONOMER is £10 extra for users of Professional Astrologer. ASTRONOMER has a full planetarium display, a choice of 5 coordinates systems, Moon/Mercury/Venus/Mars actual face display with real shadows and eclipses, automatic parallax correction (only NASA has this!), solar system display in parallel projection with zoom, tilt, freeze, autoincrement etc. If not bought at the same time as Astrologer, the price is £79.95 ☆☆☆☆☆

3. A THUMBNAIL DIGITAL C TUTORIAL

3.1 A Short Program

The following mini-tutorial would need far too much space if it were to illustrate every feature of DIGITAL C, let alone if it had to start from scratch concerning programming principles. We do assume that you are acquainted with SuperBASIC and that you will consult Chapter 4 for the finer points.

Let's consider the notorious minimal program in any language, i.e. the one that produces the output

```
Hello, world!
```

As we know, this can be accomplished in SuperBASIC by the program

```
100 PRINT "Hello, world!"
```

The corresponding program written in DIGITAL C is somewhat longer:

```
main()
{
puts("Hello, world!");
}
```

Every C program consists of one or more functions, which are rather similar in concept to both procedures and functions in SuperBASIC. A DIGITAL C function is defined by writing its name followed by the parameters in parentheses (parameters are optional, the parentheses are not) and by the 'body' of the function (i.e. what in SuperBASIC would come between the DEFine and END DEFine statements) enclosed in braces (also called curly brackets), i.e. { and }.

We can see that the function main(), which constitutes the whole program shown above, conforms to this function syntax. The name main() was not selected at random: every DIGITAL C program must contain a function called main(), or else the Code Generator will complain.

Since DIGITAL C syntax does not distinguish between functions and procedures, a function may be called by assigning its value to a variable, but it can also be called like a SuperBASIC procedure. In the example, the latter method is applied to the function puts(arg) which prints arg, followed by a linefeed, to the default output channel.

You will have noticed that the only executable statement in the above program is terminated by a semicolon. This is required statement syntax in C.

Every DIGITAL C program automatically opens 3 default channels at the start of execution. They are for input (channel 0), output (channel 1) and error messages (channel 2). More about channels can be found in Section 4.15.

By the way, the puts() function, like all i/o functions and many others, is not 'really' part of the DIGITAL C language. It was the aim of the creators of C to define a small and straightforward compiler, so they restricted the keywords of the original language to variable type declarations and program-control statements. DIGITAL C, being a subset of the original C language, sticks to the same philosophy.

But what about the many other things the user needs? These elements are either contained in a library or must be programmed by the user. Every C system features a standard library; in DIGITAL C it can be found on your disk/microdrive under the name std_lib. It is accessed automatically whenever necessary, without the user having to compile, link or otherwise connect it with his/her programs. But be sure it is available on the default device.

Of course the program, if first put down on paper, needs to be transferred to a file and subsequently compiled before it can be run. The routine for this has been given in Section 2.1 and Subsections 2.5.1 to 2.5.3.

3.2 Getting Bolder

Our next program is somewhat longer. It requests two words of input and compares the numbers of vowels they contain.

```
main()
        {
        char word1[255], word2[255];
        fputs("\nFirst word: ",1);
        wordinput(word1);
        fputs("Second word: ",1);
        wordinput(word2);
        fputs("\nThe two words have ", 1);
        if(vowelcount(word1) != vowelcount(word2))
                fputs("different numbers",1);
        else
                fputs("the same number",1);
        fputs(" of vowels\n",1);
        }

wordinput(word)
char word[255];
        {
        int i;
        i=0;
        while((word[i] = getchar()) != 10)
                {
                putchar(word[i]);
                i++;
                }
        putchar(10);
        }

vowelcount(word)
char word[255];
        {
        int i, nvowels;
        nvowels=0;
        for(i=0; word[i] != 0 && word[i] != 10;i++)
                switch(word[i])
                        {
                        case 'a':
                        case 'A':
                        case 'e':
                        case 'E':
                        case 'i':
                        case 'I':
                        case 'o':
                        case 'O':
                        case 'u':
                        case 'U':nvowels=nvowels+1;
                        break;
                        }
        return nvowels;
```

A short look at the program shows that it consists of three functions: main(), wordinput() and vowelcount(). wordinput(word) requests a word (terminated with ENTER) from the user, places it in the variable word and returns no value. vowelcount(word) needs a character array as its parameter and returns the number of vowels contained in it as the function value. We'll look at the internal structure of these functions later.

The first line of main() illustrates one of the major differences between SuperBASIC and DIGITAL C. In DIGITAL C every variable used in a program has to be declared beforehand. Among the variable types supported are integer, single character, integer array and character array (the last two are both one-dimensional). The array index is surrounded by brackets, not parentheses. More than one variable of the same type can be declared on the same line.

fputs() is another standard library function. It is similar to puts(), except that it does not add a linefeed character at the end of the string and that it needs a channel number as a second argument. In this case, the standard output channel 1 is used.

Non-printing characters can be printed by means of an 'escape sequence' starting with a backslash. E.g. \n (newline) means a linefeed, \t a tab, and \r a return character. If the backslash is followed by a number, this is interpreted as the octal representation of the ASCII code of a character to be printed.

The if structure has the syntax

```
if(condition)          or          if(condition)
  statement                            statement1
                                     else
                                       statement2
```

statement can be either a single statement followed by a semicolon or a compound statement, i.e. a number of statements enclosed by braces. Thus no endif is needed or allowed. The same principle holds for other constructs, e.g. the while loop in function wordinput().

The relational operator != is equivalent to SuperBASIC's <> .

Now for a description of wordinput(). A function parameter must be declared immediately after the function name, i.e. before the { character. The variables declared within a function (in our example only i) are local to that function.

while, like if, needs its condition enclosed by parentheses. The rather cryptic program line

```
while((word[i] = getchar()) != 10)
```

is only a short way of writing

```
word[i] = getchar();
while(word[i] != 10)
```

This often-used feature of DIGITAL C is called implicit assignment. Note the parentheses it requires.

In DIGITAL C integer and character variables can often be used interchangeably, as the two data types are essentially the same. Thus, the above program line checks whether word[i] is a linefeed character (ASCII code 10).

getchar() is a library function which returns a character from stdin. Similarly, putchar() writes its character argument to stdout.

The statement i++ illustrates one of the many possible abbreviations in DIGITAL C. It means nothing else but i=i+1. Other possible abbreviations are i-- for i=i-1, i+=j for i=i+j and many more.

The vowelc unt() function, finally, introduces three more DIGITAL C statements: for, switch and return.

The for construct in DIGITAL C differs somewhat from SuperBASIC. The pair of parentheses after the keyword encloses three expressions separated by semicolons. The first is an initialisation statement, the second a condition which has to be true for the body of the loop to be executed, and the third an action to be performed at the end of each repetition of the loop. Thus, SuperBASIC's

```
FOR a = b TO c STEP d
```

would come out in DIGITAL C as

```
for(a=b;a<=c;a+=d)
```

There is no restriction on the type of the statements within the parentheses; any or all of them may even be empty (though the semi-colons are compulsory). Thus the first statement does not necessarily assign a value to a loop variable, nor does the second one necessarily check its value. Some possible for headers are

```
for(a=1;b<27;c--)      or      for(;getchar()='x';)
```

The && in the for statement in function vowelcount() is equivalent to SuperBASIC's AND; || is equivalent to OR, and ! to NOT. (DIGITAL C has & and ^ corresponding to SuperBASIC's bitwise && (and) and || (or) respectively, in case you were wondering).

Remembering the syntax for if and while, we are not surprised to note that there is no endfor statement. Again, the body of the loop consists of just one (single or compound) statement.

The switch construct corresponds to SuperBASIC's SELect group, with the case prefix replacing the ON statements and default the REMAINDER keyword; only single cases are allowed. Once the program has found the first match, it performs the action prescribed not only for that case but also for all following cases. This 'fall-through' is desirable in our case, but can be avoided if necessary by means of the break keyword, which causes the program to leave a switch (or for or while) construct, much like SuperBASIC's EXIT.

The return statement is an old acquaintance from SuperBASIC. Note that it is not necessary if no value needs to be returned by the function (see wordinput()).

3.3 Flexing Digital C's Muscles

Our final program takes care of the few important points that have not been covered yet. It reads nine characters and a number n, and then reverses the order of the second to nth characters.

```
#define    MAXNUM   8

int start = 2;
char c[MAXNUM];

main()
{
int end;
char help[1];
read_values(c,help);
end=atoi(help);
reverse_part(c,start-1,end-1);
write_string(c);
}

read_values(c,end)
char *end,c[];
{
char h;
fputs("Enter 9 characters: ",1);
fgets(c,10,0);
fputs("\nEnter number of last character to be changed: ",1);
fgets(end,1,0);
}

reverse_part(c,x,y)
char c[];
int x,y;
{
int h;
for(;x<=(x+y)/2;x++,y--)
  {
  h=c[x];
  c[x]=c[y];
  c[y]=h;
  }
}
write_string(c)
char c[];
{
fputs("\n",1);
fputs(c,1);
}
```

The first line introduces us to the 'preprocessor' (something like the EQU directives in assembler packages), which is also part of the DIGITAL C package. Any statements prefixed by a hash sign # are executed before the start of compilation. The most important two are #include and #define.

```
#define MAXNUM 8
```

scans the whole program for appearances of the so-called symbolic constant MAXNUM and replaces each with the integer constant 8, whereas

```
#include filename
```

merges the contents of filename into the program. Note that preprocessor statements, not being part of DIGITAL C proper, do not need semicolons after them. For further information about the preprocessor and its possible uses see Sections 4.13 and 4.14.

The next two lines introduce us to the concept of 'global variables'. As we have said earlier, most variables in DIGITAL C are local to their functions. There is just one exception: variables which are defined outside any function are global, i.e. they can be accessed anywhere in the program module. Another feature of global variables is that they can be initialised when they are defined.

If within a function a variable is called which is defined not only outside that function but also outside the whole module, it is clearly necessary to tell the compiler that it must search elsewhere for such an 'external' variable. This can be accomplished with an extern statement, such as

```
extern int x, char c[], d;
```

As the extern statement refers to global variables, it must occur before main() is defined. Note also that the array is not explicitly dimensioned in the external declaration, this being the privilege of a global declaration in the called module.

Back to our sample program: at first glance the function main() does not introduce any new statements except the function atoi(), which converts its string argument to an integer and returns the integer value. But there is one important concept introduced here, the concept of pointers.

In all standard programming languages, arrays as function parameters are passed by reference, using the address of the first array element (the other elements have consecutive addresses in memory). The method is made transparent in DIGITAL C: the address of an array (as of any variable) is called a pointer to the array and accessed by the name of the array without brackets (see the function calls in main()). To refer to the pointer to a scalar variable var, use &var; the inverse process (accessing the contents of a given address addr) is accomplished with *addr for both scalar and array variables. Thus the first and third elements in array c[] (i.e. the ones with subscripts zero and two) can be accessed by *c and *(c+2) instead of c[0] and c[2].

It is also possible to declare pointers to variables or arrays; the syntax for this is shown in the first line of read_values(). Before looking at how this is implemented, observe (comparing the declaration of help in main() with that of end in read_values()) that arrays of size one and scalar variables can be used interchangeably.

Since end has not been declared global, it needs to have its value passed from read_values() back to main(). This can be accomplished by two methods. We already know one of them: putting the value in a return statement at the end of read_values() and assigning that value to help[0] in main(). This works all right, but only for a single variable. This is why we have chosen to illustrate the other method: passing a pointer to read_values() rather than the variable itself. In the declaration, *end does not, of course, mean 'contents of the address end', but is simply the syntax for the declaration of a pointer to a variable.

An array, on the other hand, is declared as such (and not by pointer) in the function to which it is passed.

From the lofty to the mundane: the standard function fgets(str, nr, fd), which is used in read_values(), reads up to nr-1 characters from channel fd into a character array str[]; the str argument must be the pointer to the array.

The for statement in reverse_part() illustrates a convenient feature of DIGITAL C: multiple statements (comma-separated) may be used in the first and third parameters of a for statement; in the second parameter the && and || operators render this feature unnecessary.

3.4 Striking Out on Your Own

This concludes our stroll through the most important features of DIGITAL C. You will find some supplementary information in Chapter 4. In particular, the explanations of error messages given in Section 4.17 may be helpful in case of unexpected trouble.

Alas, it is in the nature of the C language that various infringements of its rules, e.g. referring to a non-existent array element, do not lead to error messages but to what are euphemistically called 'unexpected results'. So be prepared to experiment.

The information in this manual should enable you to analyse the sieve_c, and perhaps also the sort_c file, on your own. After that, will be ready for just about anything!

May we also remind you again of the literature available for further study (Section 1.1).

4. A THUMBNAIL DIGITAL C REFERENCE

4.1 Limitations

This chapter gives a rapid overview of the features of DIGITAL C for the beginner. It presupposes a reasonable acquaintance with SuperBASIC.

4.2 Program Line Organisation

DIGITAL C is fairly permissive about where you put your linefeed characters; for example, a curly bracket may come at the start or end of a line or even on a line by itself. Don't, however, try to split a string (including the quotes round it) or a name. For your own convenience, though, sticking to one statement per line is strongly recommended.

4.3 Comments

A comment may be placed wherever a space or linefeed is legal, as long as it is preceded by /* and followed by */ .

Comments can span many lines, and it is a common error to forget the closing */, thus inadvertently commenting out whole functions.

4.4 Constants

Type:	Example:	
Integer	123	(-32768 to 32767)
Character	'x'	
String	"abc"	

Note the difference between single and double quotes.

In the internal representation of a string constant, the computer marks its end with a null character /0.

Several characters are represented as 'escape sequences', i.e. preceded by a backslash:

\n	line feed (newline)
\f	form feed
\t	tab
\b	backspace
\\	(single) backslash
\'	single quote
\###	3 octal digits representing the ASCII value of a character

In all other character combinations the backslash is disregarded.

4.5 Variable Types

Every variable in a DIGITAL C program must be declared before its first use. ('Variable' is a SuperBASIC term corresponding roughly to what C calls an lvalue - because you find it at the Left of the '=' in an assignment.) The 'scope' of such declarations will be treated in section 4.7. The following table shows the different variable types and the syntax for their declaration.

Syntax (example)	Variable type
int a;	sixteen-bit integer
char b;	character (almost identical with int, e.g. b=' '; can be replaced by b=32;). DIGITAL C treats all character variables like unsigned integers, i.e., the most significant byte is cleared to zero on assignment.
int c[const];	(one-dimensional) integer array of const elements starting with c[0]
char d[const];	(one-dimensional) character array of const elements starting with d[0]
int *e;	pointer to an integer, i.e. an address where an integer is stored. In an expression, e means the address and *e the contents of the address, viz. the integer value. All pointers are themselves of type integer
char *f;	pointer to a character. For explanation see above
int (*g)();	pointer to a function returning an integer. Note that the parentheses around *g are compulsory: int *g() defines a function returning a pointer to an integer
char (*h)();	pointer to a function returning a character; otherwise see above

More than one variable can be declared in a single statement, e.g.:

```
int a, b, c[32], *d;
```

declares two integer variables, one array and one pointer.

It is not necessary to declare a pointer to an array, because the name of the array itself (without the brackets) serves this function. E.g.

```
int a[12];          and          int a[12];
*(a+3)=5;                        a[3]=5;
```

both have the same meaning.

4.6 Initialisation

All global variables and array elements are initialised to zero by default. It is, however, possible to initialise global variables explicitly when defining them. The examples

```
int a = 5;
char c = 'x';
int a[] = {3,7,5,4,8,6,2,4};
char c[10] = "DIGITAL C\n";
```

show the syntax rules for initialisation. Note that character arrays can - but need not - be initialised by strings and that the number of array elements need not be specified if the variable is initialised explicitly. If the array size has been specified and the number of initialisers is less than the number of array elements, the remaining elements are initialised to zero. The number of initialisers must not be greater than the array size.

The initial value of local variables is undefined. They cannot be initialised with their declaration and must therefore be assigned a value before they are first used.

4.7 Scope of Variables

Variables can be declared in two places: at the beginning of any compound statement (this includes the 'body' of a function definition), and outside of any function.

If a variable is declared inside a compound statement, it is defined inside this compound statement and nowhere else; it is local to the statement or function. Function parameters must always be declared between the parameter list and the body of the function (see Section 3.2 for an example).

A variable defined outside any function is called a global variable. It can be accessed by every function in the program module. Of course global variables can only be initialised when they are defined, i.e. outside any function.

If two modules are compiled together, module A may use variables declared in module B (external variables). Such variables must be declared global in module B and external in module A. The external declaration differs from the global declaration in that it is preceded by the keyword extern and that any dimensioning brackets are empty. The above principle can be extended to more than two modules.

4.8 Operators

4.8.1 Unary operators

-expr	arithmetic negative
!expr	logical negation
~expr	bitwise complement
*expr	'indirection': the result is the value stored at the address expr
&var	the inverse operator to *: yields the address where the variable var is stored
++var var++	both increment var by 1
--var var--	both decrement var by 1

The difference between prefix and postfix notation is best described by an example:

```
if(++x>5) . . .
```

increments x by 1 and compares the result with 5, whereas

```
while(x++>5) . . .
```

compares x with 5 and increments x afterwards.

4.8.2 Binary operators

expr1+expr2	arithmetic addition
expr1-expr2	arithmetic subtraction
expr1*expr2	arithmetic multiplication
expr1/expr2	arithmetic division
expr1%expr2	modulo operator
expr1<<expr2	left shift of expr1 by expr2 bits
expr1>>expr2	right shift of expr1 by expr2 bits
expr1&expr2	bitwise and
expr1^expr2	bitwise exclusive or
expr1|expr2	bitwise not
expr1>expr2	expr1 is greater than expr2
expr1<expr2	expr1 is less than expr2
expr1>=expr2	expr1 is greater than or equal to expr2
expr1<=expr2	expr1 is less than or equal to expr2
expr1==expr2	expr1 is equal to expr2
expr1!=expr2	expr1 is not equal to expr2
var=expr;	assignment
var+=expr;	increments var by expr

The operators -=, *=, /=, %=, <<=, >>=, &=, ^= and |= work much like += . Any of these assignment operators can be used in an expression, e.g. if((x=a+b)==5) . . . (implicit assignment).

Any operator may be preceded and/or followed by 'white space' (space or tab characters), mainly for clarity's sake: Compare i+++++j and the equivalent i++ + ++j .

4.9 Expressions

An expression is basically a combination of variables, constants, function calls, operators and/or parentheses. Names can be up to 16 characters long, with upper and lower case distinguished, and can consist of letters, digits and the underscore _ . The first character must be a letter. As in SuperBASIC, 'logical expressions' are assigned integer values: 0 for false, and 1 (or any non-zero value on assignment) for true.

4.10 Statements

There are two types of statements: single statements, which are followed by a semicolon; and compound statements (each of which is an optional list of variable definitions followed by a list of single and/or compound statements), surrounded by braces. A single statement may be an assignment, a function call, or one of the following keywords or keyword combinations.

4.11 Keywords

4.11.1 The if ... else statement

```
if(expr)
statement1
else
statement2
```

is similar to the SuperBASIC IF ... ELSE ... ENDIF construct; DIGITAL C also supports the if ... else if ... else variation.

4.11.2 The while statement

```
while(expr)
statement
```

repeats statement until the expression is false (equals zero) or the loop is left with a break statement.

4.11.3 The do ... while statement

```
do
statement
while(expr)
```

is similar to the while loop, except that the condition is checked after the loop, so that statement is executed at least once.

4.11.4 The for statement

```
for(expr1;expr2;expr3)
statement
```

works exactly like

```
expr1;
while(expr2)
statement
expr3;
```

All three expressions are optional, the semicolons are not.

4.11.5 The switch statement

```
switch(expr)
{
case const1: statement1
.
.
case consti: statementi
default:     statement0
}
```

is equivalent to the SuperBASIC CASE statement. The expression is evaluated and compared with the 'case' constants. If a match is found, the relevant case prefix marks the place where program execution continues. If no match is found and there is a default prefix, the statements following it are executed. Contrary to SuperBASIC, the case prefix has no influence on the flow of program control, which continues until either a break statement is executed or the end of the switch statement is encountered.

4.11.6 The break statement

```
break
```

is similar to the SuperBASIC EXIT statement: it exits a while, do ... while, for, or switch statement, but in case of nested loops only the innermost one. Program execution continues with the first statement after the exited statement. Break statements are necessary to avoid 'fall-through' in the switch statement (see above).

4.11.7 The continue statement

```
continue
```

is equivalent to the SuperBASIC NEXT statement: it jumps to the end of an enclosing while, do ... while, or for loop, causing the next repetition of the loop to start.

4.11.8 The goto statement

```
goto label
```

jumps to a label of the form

```
label:
```

within the same function and continues the program from there.

4.11.9 The return statement

```
return
```

or

```
return expr
```

terminates a function and returns control to the calling function. If there is an expression after the keyword return, it is evaluated and returned to the calling function.

4.12 Functions

4.12.1 Function definition

DIGITAL C only allows external function definition, i.e., no function may be defined inside another function.

The syntax of a function definition is

```
name(parameter list)
parameter declaration
compound statement
```

where the parameter list (and with it the declaration) is optional.

Example:

```
charcount(line)
char line[160];
{
int num;
for(num=1; line[num]!='\n' && line[num]!=0; num++)
  putchar('*');
return num;
}
```

is a function that (a) counts the number of characters in a character array line until a linefeed character or \0 (the end-of-string marker) is reached, and (b) prints an asterisk for every character.

4.12.2 Function calls

A function can be called either by using it in an expression (in which case the function must contain a return statement returning a value, much like a SuperBASIC function) or simply by stating its name (like a SuperBASIC procedure).

Function parameters are basically passed by value with the exception of arrays, which are always passed by reference. If you want to pass a scalar (i.e. non-array) variable by reference, you can do so by passing a pointer to its address.

The final executable program must contain a function called main() or main(argc, argv) which will be the first function executed. If a command line is requested by the application program, the arguments argc and argv will be passed to main(). argc is the number of arguments in the command line; argv[] is an array of pointers to strings:

```
argv[1] points to the first argument on the command line
        (other than the name of the called program);
argv[2] points to the second argument in the command line;
.......
argv[0] in DIGITAL C points to a '*'string.
```

If your program contains a function called _banner, this will be executed before a command line is requested. The use of function and variable names beginning with _ should be avoided otherwise, since the system uses several such names.

If your program contains a function called _console, this will be executed before any console window is opened by the entry code. This is useful for opening your own console window and overriding the default windows. Your _console function must return the file descriptor of your console window.

4.12.3 Pointers to functions

It is possible to pass a function as a parameter to another function via a pointer. In the following example a sorting function is to be passed to a function called fprint:

```
fprint(sort)
int (*sort)()
{
...
(*sort)(a,b);
...
}
```

An example for the function call would be fprint(shell) or fprint(quick), depending on the name of the sorting algorithm to be used. Of course, either sorting function needs to refer to the same number of parameters (two in this example). Observe that the name of a function without parentheses denotes a pointer to the function (similar to arrays; see Section 4.5).

4.12.4 Function argument counts

Few functions expect a variable number of arguments. Only two are supplied in std_lib, printf() and fprintf(). If these are not used, the compiled code may be made more concise and faster by defining NOCCARGC at the start of a source file, e.g. with

```
#define NOCCARGC
```

The upper case is essential. If printf or fprintf are used, NOCCARGC must not be defined. Of course you can restrict the functions using printf and fprintf to a separate module.

4.13 The Preprocessor

The DIGITAL C Parser includes a 'preprocessor' to facilitate inclusion of other sources files, conditional compilation and definition of constants.

The keywords which are understood by the preprocessor and executed before the start of compilation are listed below.

```
#include filename
```

Merges the contents of filename into the source file at the place of the #include keyword.

```
#define name constant
```

Gives a name to a constant used in a program, e.g.

```
#define MAXLENGTH 92
#define SPACE     ' '
```

Note that the preprocessor statements, not being part of DIGITAL C proper, do not need the semicolon at the end.

For advanced programmers, DIGITAL C also supports conditional compilation with the keywords

```
#ifdef name
#ifndef name
#else
#endif
```

#ifdef checks whether name has been defined in the preprocessor. If so, the following lines are compiled (until a #else or #endif is reached). #ifndef checks for the inverse condition.

4.14 Standard Definitions

Some standard definitions are contained in a file called stdio_h . To make programs portable to other compilers, these definitions should be used in place of the constants they represent, e.g. stdout instead of 1 as the number of the standard output channel. However, some authors do not conform to the conventions for standard definitions, e.g. Dr. Dobbs uses NULL for the null string rather than for a non-existent address, opening the door to all sorts of problems. Remember that strings are terminated with a null character, no matter what name you use for it, and that the NULL address in DIGITAL C is -32768 (Hex 8000) and not 0; address 0 actually exists.

4.15 Standard DIGITAL C Channels

There are three standard channels when a compiled DIGITAL C program is started up. They have the file descriptors 0 (stdin), 1 (stdout) and 2 (stderr) for input, output and error reports respectively (The three names in parentheses are conventionally established as equivalent to the numbers in stdio_h). They all use the same QDOS channel, which is a console channel. If you close one, you will close all three! You are of course free to open more channels up to a maximum of 16 (including the three already used).

4.16 Libraries

Libraries are supplied in the file std_lib, which, as explained in section 2.2, must not be included in the command line of the Parser. There are three categories of functions in the library:

(a) standard C functions
(b) QDOS functions
(c) the floating-point library

These are explained below, giving function name and parameters as well as a brief explanation of the function's operation. Note that all these library functions are contained in std_lib. Where a file descriptor fd is mentioned, this is the value of the integer returned by fopen when the file is opened.

4.16.1 Standard C library functions

The following standard library functions are common to nearly all C systems:

```
abort(n) int n;     Aborts a program, error message if n != 0.

abs(n) int n;       Returns absolute value of n.

atoi(s) char *s;    Returns string s converted to an integer.

atoib(s, b) char *s; int b;
                    Returns string s converted to an integer using base
                    b.
```

avail(code) int code;
Returns the number of bytes of free memory in the dataspace of the task. This number does not include holes left by previous use of free(). If there is no free memory, the action taken depends on the value of code: if code = 0, avail() returns 0; if code != 0, the task is aborted.

calloc(nbr, sz) int nbr, sz;
Allocates nbr*sz bytes of zeroed memory from the task's dataspace. Returns the address if successful, otherwise NULL.

ccargc()
Returns the number of arguments in the function call.

cfree(addr)
Frees allocated memory, addr being the address previously returned by calloc() or malloc().

delete(str) char *str;
Deletes the file whose name is at str.

fclose(fd) int fd; Closes file fd, returns 0 for success, else -1.

fgetc(fd) int fd; Returns the next character from file fd .

fgets(str, sz, fd) char *str; int sz, fd;
Reads up to sz-1 characters from file fd into memory at address str.

fopen(name, mode) char *name, *mode;
Opens a file by name. Mode points to a string which is "r" for read, "w" for write, "a" for append. Returns the file descriptor fd, which must be used by other i/o functions. Returns NULL if the open fails.

fprintf(str, var1, var2, ...)
Formatted print to file. str is a string (enclosed in double quotes) specifying how the arguments are to be converted for printout. With the exception of conversion specifications, every character in str is printed literally. A conversion specification is introduced by a percentage sign and ended with a conversion character. This can be d, o, x, u, s, b or c, causing conversion of the argument to decimal, octal, hexadecimal, unsigned decimal, string, binary or character notation, respectively. Between the percentage sign and the conversion character the following optional characters may be written:

- a minus sign, specifying left instead of the default right justification
- a number specifying the minimum width of the printed field. If necessary, the field will be extended. If the length of the converted argument is less than the field width, the field is padded with spaces as a default. If the first character of the field width is a 0, the field is padded with zeroes
- a second number preceded by a full stop, specifying the number of characters of a string to be printed

If the character after the percentage sign is not one of the above, it is printed literally. This is a way to print the percentage sign itself.
Each conversion specification corresponds to an argument of the fprintf function, i.e., the number of conversion specifications and arguments should be the same.

fputc(c, fd) char c; int fd;
Writes character c to file fd.

fputs(str,fd) char *str; int fd;
Writes string str to file fd.

fread(ptr, sz, n, fd) char *ptr; int sz, n, fd;
Reads n items, each of size sz, from file fd into memory at address ptr.

free(addr) Frees allocated memory, addr being the address previously returned by calloc() or malloc().

freopen(fname, mode, fd) char *fname, *mode; int fd;
Closes the file indicated by fd, and opens a new file whose name is fname. Mode is as for fopen(). If successful, fd is returned, otherwise NULL.

fwrite(ptr, sz, n, fd) char *ptr; int sz, n, fd;
Writes n items, each of size sz, from memory starting at address ptr into file fd.

getc(fd) int fd; Returns the next character from file fd.

getchar() Returns the next character from stdin.

gets(str) char *str;
Reads a string into str from stdin.

getarg(n,s,size,argc,argv) char *s; int n, size, argc, argv;
Locates argument n from the command line, moves it to string s, and returns the length of the string. argc and argv must be the values provided to function main() when the program is started.

isalnum(c) char c; Returns 1 if c is an alphanumeric character.

isalpha(c) char c; Returns 1 if c is an alphabetic character.

isascii(c) char c; Returns 1 if c is an ASCII character.

iscntrl(c) char c; Returns 1 if c is a control character.

isdigit(c) char c; Returns 1 if c is a digit.

isgraph(c) char c; Returns 1 if c is a graphic character (ASCII codes 33..126).

islower(c) char c; Returns 1 if c is lower case.

isprint(c) char c; Returns 1 if c is a printable character (ASCII codes 32..126).

ispunct(c) char c; Returns 1 if c is a punctuation character.

isspace(c) char c; Returns 1 if c is a white-space character (SP, HT, VT, CR, LF or FF).

isupper(c) char c; Returns 1 if c is an upper-case character.

isxdigit(c) char c;
Returns 1 if c is a hex digit.

itoa(n, s) int n; char *s;
Converts integer n to a decimal string.

itoab(n, s, b) int n; char *s; int b;
Converts integer n to a string according to base b.

itod(nbr, str, sz) int nbr; char str[]; int sz;
Converts nbr to a signed character string of size sz. Result is right-justified and blank-filled in str.

itou(n, str, sz) int n; char *str; int sz;
Converts unsigned integer n to string str of size sz. Result is right-justified and blank-filled in str.

itox(n, str, sz) int n; char *str; int sz;
Converts integer n to hex string str of size sz. Result is right-justified and blank-filled in str.

left(str) char *str;
Removes leading spaces from string str.

lexcmp(str1, str2) char *str, *str2;
Similar to strcmp(), except that a lexicographical comparison is used, i.e. upper- and lower-case characters are considered equal.

lexorder(c1, c2) char c1, c2;
Returns an integer less than, equal to or greater than zero depending on whether c1 is less than, equal to or greater than c2 lexicographically.

malloc(n) int n; Allocates n bytes of uninitialised memory from the task's dataspace. Returns the address if successful, otherwise NULL.

printf(str, var1, var2, ...)
Formatted print to stdout. For a list of format options see fprintf.

putc(c, fd) char c; int fd;
Sends character to file fd.

putchar(c) char c; Equivalent to fputc(c, stdout).

puts(str) char *str;
Equivalent to fputs(char,stdout) followed by line-feed.

read(fd, ptr, n) int fd; char *ptr; int n;
Reads n bytes from file fd into memory at address ptr.

reverse(s) char *s;
Reverses the order of the characters in string s.

strcat(s, t) char *s, *t;
Appends the string at t to the string at s. No check is made on the size.

strchr(str, c) char *str, c;
Returns a pointer to the first occurrence of character c in a string, returns NULL if not found.

strcmp(s, t) char *s, *t;
Returns an integer less than, equal to or greater than 0 if string s is less than, equal to or greater than string t.

strcpy(s, t) char *s, *t;
Copies string t to s.

strlen(s) char *s; Returns the length of string s.

strncat(s, t, n) char *s, *t; int n;
Like strcat(), except that a maximum of n characters is transferred.

strncmp(s, t, n) char *s, *t; int n;
Like strcmp(), except that a maximum of n characters is compared.

strncpy(dest, sour, n) char *dest, *sour; int n;
Like strcpy(), except that n characters are copied.

strrchr(s, c) char *s, c;
Like strchr, except that the rightmost occurrence of c is found.

tolower(c) char c; Returns lower-case equivalent of c.

toupper(c) char c; Returns upper-case equivalent of c.

utoi(str, nbr) char *str; int *nbr;
Converts the unsigned decimal number represented by the string at str to an integer at nbr and returns the length of the numeric field found, or ERR if the number is larger than 65535.

write(fd, ptr, n) int fd, char *ptr; int n;
Writes n bytes from memory starting at address ptr into file fd.

4.16.2 QDOS functions

File std_lib includes some QL-specific functions which provide the same operations as SuperBASIC keywords, usually with the same name. There is, however, less flexibility than in SuperBASIC, since all parameters must be included, in particular the file descriptor fd, which is equivalent to the SuperBASIC channel number.

For the format of floating-point arguments see Subsection 4.16.3.

Functions for which no return value is specified in the following description return the QDOS error code, i.e.

0 for no error
-n for an error where n is as described in the QL User Guide.

adate(n) int n; Adjusts the clock by n seconds. Return is undefined.

arc(fd, x1, y1, x2, y2, angle) int fd; 'float' x1,y1,x2,y2,angle;
Plots an arc from (x1,y1) to (x2,y2) subtending angle.

at(fd, line, col) int fd, line, col;
Positions the cursor using character co-ordinates.

baud(rate) int rate;
Sets the baud rate of the serial interfaces. Return is undefined.

```
beep(dur, pitch)    Return is undefined.

beeping()           Returns 1 if a sound is being made.

block(fd, width, height, x, y, colour) int ...;
                    Draws a solid rectangular block in colour.

border(fd, size, col) int fd, size, col;
                    Draws a border round a window.

circle(fd, x, y, radius) int fd; float x, y, radius;
                    Draws a circle of given radius at (x,y).

cls(fd) int fd;     Clears a window.

clsp(fd, switch) int fd, switch;
                    Clears part of a window or line.

csize(fd, width, height) int fd, width, height;
                    Sets the character size in a given window.

cursor(fd, x, y) int fd; float x, y;
                    Positions the graphic cursor at (x,y).

curson(fd) int fd; Enables a cursor.

cursoff(fd) int fd;
                    Suppresses a cursor.

dates(str) char *str;
                    Reads the date into a string.  Return is undefined.

date()              Returns the  date as  a long integer.  Used  in the
                    form  *d = date(), where  d has been declared  as a
                    2-element integer array.

day(str) char *str;
                    Reads the day into a string.

ellipse(fd, x, y, radius, ecc, angle) int fd; float ...
                    Draws an ellipse.

fill(fd, switch) int fd, switch;
                    Turns area flood on or off.

flash(fd, switch) int fd, switch;
                    Turns flash mode on or off.

ink(fd, col) int fd, col;
                    Sets the ink colour to col.

keyrow(n)           Returns the value of keyrow n.

line(fd, x1, y1, x2, y2) int fd; float ...;
                    Plots a line.

mode(n) int n;      Sets 4 or 8 colour mode.  Return is undefined.

pan(fd, n) int fd, n;
                    Pans a window n pixels right or left.

panp(fd, n, part) int fd, n, part;
                    Pans part of a window left or right.

over(fd, switch) int fd, switch;
                    Sets character-plotting mode.
```

```
paper(fd, col) int fd, col;
                    Sets the paper colour to col.

point(fd, x, y) int fd; float x, y;
                    Plots a point.

quiet()             Turns off any sound.

scale(fd, scale, x, y) int fd; float scale, x, y;
                    Sets the graphic scale.

scroll(fd, n) int fd, n;
                    Scrolls a window up or down.

scrollp(fd, n, part) int fd, n, part;
                    Scrolls part of a window up or down.

sound(d,p1,p2,x,y,wr,fz,ra) int ...;
                    Complete sound specification,  like SuperBASIC BEEP
                    with all parameters.

strip(fd, col) int fd, col;
                    Sets the strip colour to col.

under(fd, switch) int fd, switch;
                    Turns underline mode on or off.

window(fd, width, height, x, y) int ...;
                    Adjusts the size and position of a window.
```

4.16.3 Floating-point library

The original Small-C did not include floating-point (real) numbers at all; but these are necessary on the QL if you wish to use some of the graphics routines. Therefore floating-point library functions (all the usual mathematical functions plus some conversion functions) have been provided.

A floating-point number must be declared as a 3-element integer array; e.g.

```
int a[3], b[3], c[3];
```

declares three floating-point variables a, b and c. For the equivalent of a = b + c you must write one of the following in your program:

```
*a = fadd(b,c);
a[0] = fadd(b,c);
```

or

Similarly for all the other floating-point operations.

To do several operations, they must be programmed separately, e.g., if a, b, c, d are all floating-point variables, for a = b + c + d you must write

```
*a = fadd(b,c); *a = fadd(a,d);
```

or similar. A line such as

```
*a = fadd(fadd(b,c),d)
```

will not work.

The list of operations is: (x, y are floating-point variables)

acos(x)

acot(x)

asin(x)

atan(x)

atof(str) char *str;
Convert string to float.

cos(x)

cot(x)

exp(x)

fabs(x) Absolute value.

fadd(x,y)

fcmp(x,y) Compare: returns <0, 0, >0 for x<y, x=y, x>y.

fdiv(x,y)

float(n) Convert integer n to floating point number.

fmove(x,y) Move x to y.

fmult(x,y)

fneg(x)

fsub(x,y) x-y.

ftoa(x, str) char *str
Convert float to string.

int(x) Convert float to int.

log(x) Natural logarithm.

log10(x) Log to base 10.

pow(x,y) x to the power y.

sin(x)

sqrt(x)

tan(x)

4.17 Errors

4.17.1 How to interpret error messages

Most error messages in DIGITAL C are self-explanatory. The rather cryptic "must be lvalue" usually means that you have not declared a variable. The error marker appears just after the actual error, so that a missing semicolon should often be located at the end of the previous line. An error in an expression turns off error detection until the next semicolon, so that there may be further unreported errors on a line.

As DIGITAL C assumes undeclared identifiers to be function names in another module, undeclared variables may not be detected until the Code Generator is run. Variables defined in other modules must be declared using the extern declaration.

Error reports in the Code Generator refer to the source module and line number where the error was detected and, where relevant, give the offending identifier. If, say, the required definition of a function is missing, only the first reference to it is reported, further occurrences being suppressed.

4.17.2 Parser error messages

"Already defined" Identifier has been declared before and cannot legally be redefined, e.g. a global variable with the same name.

"Bad label" The label following a goto is illegal.

"Cannot assign to pointer"
A pointer variable cannot be initialised in a declaration.

"Can't subscript" Only arrays or pointer variables may be followed by a subscript, i.e. a '[' character.

"Error writing to file"
When saving intermediate code to an _obj file, a QDOS error has occurred, e.g. if the drive is full.

"Global symbol table overflow"
The internal space allocated for storing global variable names has been filled up (a fatal error). Use fewer global variables.

"Illegal address" following the & address operator.

"Illegal argument name"
An invalid identifier has occurred in a list of arguments in a function call or declaration.

"Illegal function or declaration"
An invalid identifier has occurred where a function or other declaration was expected.

"Illegal symbol" An invalid identifier has occurred.

"Invalid expression"
You have made some syntax error in an expression.

"Line too long" After pre-processing, i.e. replacing macros with their text, the resulting line is too long for the internal buffer.

"Literal queue overflow"
The space allocated for temporary storage of string literals has been filled up. Use shorter strings, or split the function in which this error occurs into two or more smaller functions.

"Local symbol table overflow"
The internal space allocated for local variables has been filled up. Use fewer local variables in that function.

"Macro name table full"
The internal space allocated for macro names has been filled. Use fewer macros.

"Macro string queue full"
The internal space allocated for macro replacement strings has been filled. Use fewer macros or shorter strings.

"Missing token" One of the following characters or keywords has been omitted: '}',']','(',')',':' , while .

"Multiple defaults"
More than one default keyword has been used inside a switch statement.

"Must assign to char pointer or array"
You have tried to initialise a variable which is not a char pointer or array to a literal string.

"Must be constant expression"
A constant expression was expected, e.g. after the case keyword or when initialising a variable.

"Must be lvalue" You have an invalid identifier on the left-hand side of an assignment expression, e.g. a function name or undeclared identifier.

"Must declare first in block"
Local variables must be declared at the start of a block.

"Negative size illegal"
The index to an array must not be negative, e.g. a[-2] is illegal.

"Not allowed in switch"
Local variables cannot be declared in the middle of a switch statement.

"Not allowed with block-locals"
"Not allowed with goto"
Local variables cannot be declared in the middle of a block containing a goto statement; they must be declared at the start of the function.

"Not an argument" An illegal argument has occurred in the argument list of a function call.

"Not a label" An invalid label, e.g. a local variable, has been used where a label should be used.

"Not in switch" A case or default has occurred outside a switch statement.

"No apostrophe" A ' character has been omitted in an input line.

"No closing bracket"
Tnere is a missing } somewhere in the program. This message only occurs at the end of a program.

"No comma" Commas must be used to separate arguments in the argument list of a function call or declaration.

"No final }" A } is missing at the end of a compound statement.

"No matching #if..."
#else or #endif has occurred in the source file without a matching preceding #if.

"No open paren" The (character must follow the function name in a function declaration.

"No pointer arrays"
You cannot declare an array of pointers.

"No quote" The " character is missing from a line of input.

"No semicolon" This message can occur wherever a ; is expected.

"Open failure on include file"
Tne file whose name followed a #include directive cannot be opened.

"Out of context" One of the keywords break or continue has occurred out of place, e.g. outside a while loop.

"Staging buffer overflow"
The temporary buffer holding code for the _obj file has been filled. Use simpler expressions.

"Too many active whiles"
An internal table holding details of while and similar statements has been filled. Have fewer nested while statements.

"Too many cases" An internal table holding details of case statements inside a switch statement has been filled. Use smaller switch statements.

"Wrong number of arguments"
In a function declaration the number of argument declarations does not match the number of arguments in the argument list.

4.17.3 Code Generator/Linker error messages

In the Code/Generator/Linker, there are three causes of an error message:

(A) A user-generated error, e.g. undefined variable.
(B) A QDOS system error, e.g. 'drive full', 'bad medium', etc. All you can do if you get such a message is try again.
(C) A Parser or Code Generator/Linker error. Errors of this type should never occur. If one does, and is not due to a corrupted file, please report the occurrence, accompanied with the source or object code which generates the message, to Digital Precision. Please try to isolate and minimise the source code which causes the error.

The error messages can take one of four forms, which are given in the list below.

Form 1: <module_name> at line nnn: <error_message>
Form 2: <module_name> at line nnn: <name> - <error_message>
Form 3: <error_message>
Form 4: <name> <error_message>

where <module_name> is the source filename which was compiled into the code wnere the error was detected.

nnn is a line number in the source file.

<error_message> is one of the error messages listed below.

<name> is a variable or function name.

"ambiguous reference" (cause A, form 2)
An external variable in one module has been declared as a global variable in two or more other modules, instead of the one that is permitted. The reference cannot be resolved.

"Cannot allocate negative byte count" (cause C, form 1)

"Cannot copy negative byte count" (cause C, form 1)

"Cannot make program EXECable" (cause B, form 3)
An attempt to generate an executable task has failed.

"Cannot skip negative byte count" (cause C, form 1)

" - Can't open" (cause B, form 4)
An attempt to open a file has failed.

"Close error" (cause B, form 3)
An attempt to close a file has failed.

"Dataspace must be at least 1000" (cause A, form 3)
The dataspace specified in the command line was less than 1000 or greater than 32767.

" - device name too long" (cause A, form 4)
The specified device name is too long. Use a shorter name.

"Fatal error reading file" (cause B, form 3)
An attempt to read a file has failed.

"Fatal error writing to file" (cause B, form 3)
An attempt to write to a file has failed.

"function multiply defined" (cause A, form 2)
The same function name has been used in two or more modules.

"function not found" (cause A, form 2)
A function reference has been made to a non-existent function.

"Instruction out of range" (cause C, form 1)

"Invalid instruction" (cause C, form 1)

"Invalid variable type" (cause C, form 1)

"Label too big" (cause C, form 3)

"Name already exists" (cause A, form 1)
A function or variable name has been used twice.

"name not defined" (cause A, form 2)
A reference to a non-existent function or variable has been made.

"Phase error" (cause C, form 1)

"Premature end of file" (cause C, form 3)

"Program too large" (cause A, form 3)
The compiled program plus requested dataspace is greater than 64K.

"Symbol table full" (cause A, form 3)
The internal symbol table of the Code Generator/ Linker has been filled. Use fewer and/or smaller variables.

" - Too long" (cause A, form 4)
A user-specified filename is too long. Use a shorter filename.

"Usage: outfile objfile ... libfile ... [-p] [-m] [-d/dev] [-c/file] [-nc] [-s#]" (cause A, form 3)
An error has been made in the command line; the permitted format is indicated. [] around a command-line switch indicate it is optional.

"variable not defined" (cause A, form 2)
A reference to a non-existent variable has been made.

" - Variable or external symbol table error" (cause C, form 4)

4.17.4 Library Generator error messages

The three causes and the four forms of error messages in the Library Generator are the same as for the Code Generator/Linker.

"Cannot copy negative byte count" (cause C, form 1)

" - Can't open" (cause B, form 4)
An attempt to open a file has failed.

"Close error" (cause B, form 3)
A failure has occurred when closing a file.

"Fatal error writing to file" (cause B, form 3)
A failure has occurred wnen writing to a file.

"Fatal error reading file" (cause B, form 3)
A failure has occurred wnen reading from a file.

"Funcsize space exhausted" (cause C, form 3)

"Instruction out of range" (cause C, form 1)

"Invalid instruction" (cause C, form 1)

"Invalid variable type" (cause C, form 1)

"is duplicated" (cause A, form 2)
A function name has been duplicated.

"Premature end of file" (cause C, form 3)

"Symbol space exhausted" (cause A, form 3)
An internal table has been filled. Don't try to generate such a large library file.

" - Too long" (cause A, form 4)
A user-specified device or filename is too long. Use a shorter name.

FEATURES	STANDARD EDITOR	Metacomco ed	QUILL v2.35
Handle Quill DOC files	Yes	-	Yes
Accept all characters 0-255	Yes	-	-
Display all characters 0-255	Yes	-	-
Printer driving	Yes	-	Yes
Simultaneous printing/editing	Yes	Yes	-
All printer fns allowed	Yes	-	-
Restriction-free cursor move	Yes	Yes	-
Change delimiters for words	Yes	-	-
Cursor word left/right	Yes	Yes	Yes
Cursor start/end of line	Yes	Yes	-
Cursor to next truncated line	Yes	-	-
Cursor top/end of screen/file	Yes	Yes	-
Cursor to marker	Yes	-	-
Cursor to specified line	Yes	Yes	-
Cursor to next/prior para	Yes	-	Yes
Cursor to start/end of block	Yes	-	-
Cursor to last command point	Yes	-	-
Non-immediate cursor commands	Yes	-	-
Page screen forward/back	Yes	Yes	-
Scroll screen up/down	Yes	-	-
Delete char/word left/right	Yes	-	Yes
Speed independent of filesize	Yes	-	-
Delete to start/end of line	Yes	-	Yes
Delete/Un-Delete line	Yes	-	-
Define block in any sequence	Yes	-	-
Delete/Move/Copy block	Yes	-	Yes
Set marker	Yes	-	-
Find from current position	Yes	Yes	-
Replace from current position	Yes	Yes	-
Set right/left/indent margin	Yes	-	Yes
Switchable case sensitivity	Yes	Yes	-
Set tabs - regular+asymmetric	Yes	-	Yes
Remove tabs	Yes	-	-
Expand tabs	Yes	-	-
Compress tabs	Yes	-	-
Justify left/right	Yes	-	Yes
Justify centre/middle	Yes	-	-
Paragraph reform (selective)	Yes	-	-
Overstrike/insert mode	Yes	-	Yes
Case translate - to UPPER	Yes	-	-
Case translate - to lower	Yes	-	-
Case translate - to Mixed	Yes	-	-
Word wrap	Yes	Yes	Yes
Move block	Yes	-	-
Retain definition of block	Yes	-	-
Sequence file on cols a to b	Yes	-	-
Renumber program lines	Yes	-	-
Undo current line editing	Yes	Yes	-
Issue multiple commands	Yes	Yes	-
Recall last commands	Yes	-	-
Issue repeat commands	Yes	Yes	-
Repeat last commands	Yes	Yes	-
Dynamic memory management	Yes	-	Yes
Memory status display	Yes	-	-
On-line help	Yes	-	Yes
Dynamic speed adjustment	Yes	-	-
Multitasking	Yes	Yes	-
Range of available colours	Yes	-	-
Adjustable window size/posn	Yes	Yes	-
Range of character sizes	Yes	-	-
PROCESS COMMAND FILES	Yes	-	-
FULLY CONFIGURABLE BY USER	Yes	-	-
SPECIAL FONTS	Yes	-	-
HANDLE ANY FILE(ASCII OR NOT)	Yes	-	-
INSTANT RESPONSE TO KEYBOARD	Yes	-	-

INDEX

DIGITAL C function names are alphabetised separately in subsections 4.16.1, 4.16.2, 4.16.3; DIGITAL C error messages in subsections 4.17.2, 4.17.3, 4.17.4

ABBREVIATIONS 3.2
ADDRESS 3.3 4.12.2
ARGC 4.12.4
ARGUMENT COUNT 4.12.4
ARGV 4.12.4
ARRAYS 3.2 3.3
ASCII 2.1
ASSIGNMENT 4.10
ASSIGNMENT, IMPLICIT 3.2
ATOI(). 3.3

BACKSLASH 3.2 4.4
BACKSPACE. 4.4
BACKUP. 1.2
BINARY OPERATORS 4.8.2
BITWISE OPERATORS 3.2 4.8.1 4.8.2
BRACES 3.1 3.2 4.10
BRACKETS 3.2
BREAK 3.2 4.11.5 4.11.6

CASE 3.2 4.11.5
CHANNELS 3.1 3.2 4.15
CHARACTERS 3.2 4.4 4.5
CHARACTER ARRAYS 3.2
CLONE 1.2
CODE GENERATOR/LINKER. 1.1 2.2 2.3 2.5.3 4.17.1 4.17.3
CODE SIZE LIMIT. 2.3
COMMA 3.3
COMMAND LINE 2.5.1 2.5.2 2.5.3 2.5.4 2.5.6 4.12.2
COMMENT 4.3
COMPILER 1.1
COMPOUND STATEMENT 3.2 4.7 4.10
CONFIG 1.2 1.5.6 2.5.1
CONSOLE WINDOW 4.12.2
CONSTANTS 4.4
CONTINUE 4.1.7
COPYING 1.2 1.3
CURLY BRACKETS 3.1 4.2

DATASPACE 2.5.3 2.5.6
DECLARATION 3.2 4.5 4.7 4.17.1
DEFAULT 3.2 4.11.5
DEFAULT CHANNELS 3.1
DEFAULT DEVICE 2.5.1
DEFAULT INITIALISATION 4.6
DEFINITION 4.14
DO 4.11.3
DR. DOBBS 1.1 2.1

EDITOR 2.1 2.5.1
ELSE 3.2 4.11.1
ERROR MESSAGES 4.17.1 4.17.2 4.17.3
ESCAPE SEQUENCES 3.2 4.4
EX 2.5.1
EXEC 2.5.1
EXECUTE 2.5.1 2.5.2 2.5.3 2.5.5
EXPRESSION 4.9
EXTERN 3.3 4.7
EXTERNAL VARIABLES 3.3 4.7

FALL-THROUGH 3.2 4.11.6
FGETS() 3.3
FILENAMES 2.5.1 2.5.2 2.5.3
FLOATING-POINT ARITHMETIC 1.1 4.16.3
FOR 3.2 4.11.6
FORM FEED. 4.4
FPRINTF 4.12.4
FPUTS() 3.2
FUNCTION 2.4 2.5 3 3.1 4.12 4.16 4.17.1
FUNCTION CALL 4.10 4.12.2
FUNCTION DEFINITION 4.12.1

GETCHAR() 3.2
GLOBAL VARIABLES 3.3 4.7
GOTO 4.11.8

HENDRIX, J.E. 2.1

IF 3.2 4.11.1
IMPLICIT ASSIGNMENT 3.2
INTEGER 3.2 4.4 4.5
INTEGER ARRAY 3.2
INITIALISATION 3.2 3.3 4.6
INTERMEDIATE CODE 2.2
I/O REDIRECTION 2.5.1 2.5.5

KEYWORDS 4.11

LABELS. 4.11.8
LIBRARIES. 4.16
LIBRARY GENERATOR 1.1 2.4 2.5.4 4.17.4
LIBRARY PROGRAMS 1.1 2.5.3
LINE FEED 3.2 4.2 4.4
LINKER - see Code Generator/Linker . . . . . .
LOCAL VARIABLES 3.2 3.3 4.6 4.7
LOGICAL EXPESSIONS 4.9
LVALUE 4.5 4.17.1

MAIN(). 3.1 3.3 4.12.2
MC_OBJ 2.5.3
MULTIPLE STATEMENTS 3.3

NEW LINE 4.4

OBJECT FILE 2.5.2 2.5.3 2.5.4
OCTAL NUMBERS 3.2 4.4
OPERATORS 3.2 4.8
OPTION STRING 2.5.1

PARAMETERS 3.1 3.3 4.12.1 4.12.2 4.12.4
PARENTHESES 3.1
PASSING BY REFERENCE 3.3 4.12.2
PARSER 1.1 2.2 2.5.2 4.17.2
PASSING BY VALUE 4.12.2
PIPES 2.5.5
PIRATING 1.3
POINTERS 3.3 4.5 4.12.2 4.12.3
PREPROCESSOR 3.3 4.13
PRINTF. 4.12.4
PROGRAM EXECUTION 2.5
PROGRAM LINE. 4.2
PUTCHAR() 3.2
PUTS() 3.1

QDOS FUNCTIONS 4.16 4.16.2
QUILL 2.1
QUOTES 4.2 4.4

RAM DISK 2.5.6
REFERENCE BOOKS 1.1
REFERENCE CHAPTER 4.2
RETURN 3.2 4.4 4.11.9 4.12.1 4.12.2

SAMPLE PROGRAMS 2.1 2.5.6
SCALAR VARIABLES 3.3
SCOPE OF DECLARATIONS 4.5 4.7
SEMICOLON 3.1 3.2 4.17.1
SIEVE-C 2.1 2.5.2 3.4
SIEVE_OBJ 2.5.2
SMALL-C 1.1
SORT_C 2.1 2.5.5 3.4
SOURCE PROGRAMS 2.1 2.5.2
STANDARD CHANNELS 4.15
STANDARD DEFINITIONS 4.14
STANDARD FUNCTIONS 4.16 4.16.1
STATEMENTS 3.2 4.10
STDERR 4.15
STDIN 2.5.5 4.15
STDIO_H 4.15
STD_LIB 2.5.3 3.1 4.16
STDOUT 2.5.5 4.15
STRING 4.4
SUPERBASIC 3.1 4.2
SWITCH 3.2 4.11.5

TAB 3.2 4.4
TEMPORARY FILES 2.5.6
TOOLKIT II 2.5.1
TURBO TOOLKIT 2.5.1 2.5.2 2.5.3 2.5.5
TUTORIAL CHAPTER 3

UNARY OPERATORS 4.8.1
UPDATES_DOC 1.1 1.2
UPPER CASE 4.12.4

VARIABLE ARGUMENT COUNT 4.12.4
VARIABLES 3.2 3.3 4.5

WHILE 3.2 4.11.2 4.11.3

Z80 . . . . . 1.1

8080 . . . . . 1.1

& 3.3
* 3.3
#DEFINE 3.3 4.13
#ELSE 4.13
#ENDIF 4.13
#IFDEF 4.13

TURBO V2.0

DP Classic

THOR COMPATIBLE

"Dramatic . . . bug free . . . an excellent program . . . a great advance on SuperCharge . . . more capable than the latest version of QL thereafter) . . . the fastest SuperBASIC compiler so far . . . solved branding . . . takes SuperBASIC and the QL into a new era . . . With TURBO one should start to see programs which are unthinkable on any other small micro"

Upto 100X speedup on BASIC! QL WORLD April 1987

☆ Supremely fast – even QL World's independent benchmarks, which do not use TURBO's speed optimisation options (which give a further 100% – 150% speed increase), show a 2181% (yes – TWO HUNDRED AND THIRTY!) speed increase over the latest version of QLiberator. We can demonstrate speed improvements that are far greater, and the speed of compilation itself is typically 1000% + more than on QLiberator

☆ Supremely concise – even QL World's size benchmarks, which do not use TURBO's size optimisation or diagnostics off options (which yield a further 40% saving in code space), show that TURBO generates more concise jobs than QLiberator!

☆ Supremely compatible – TURBO compiles virtually anything you care to throw at it Full auto-corrector built in, takes interpreter bugs in its stride, warns you if you've been too naughty

☆ Supremely easy to use – a beautiful front end, on-screen help, context sheet, our LENS.OBJ, window adjustable during compilation, multitasks while compiling, adjustable buffer size/window copying/DATASPACE from front end/etc. etc. even configurable to avoid losing keyb'ashing, adjustable report file (to screen/printer/file etc.) in full or condensed format, etc.

☆ Supremely flexible task communication – use PRINT and INPUT to communicate between any number of multitasking jobs – even arrays can be passed, and channels can be shared! Option strings can be passed as parameters of EXECUTE

☆ Supremely free from restrictions – you can use procedures or functions in any task from any other task! Generate tasks as large as the QL's RAM, HAM . . . or upgrade the machine route (we supply a program to help you) and instantly link any number of modules

TURBO AND TURBO TOOLKIT V2.0 – FASTER AND MORE POWERFUL THAN EVER BEFORE!!

V2.0 is the ultimate version of the TURBO SYSTEM, incorporating all the features you have been waiting for. Never before has the system been more flexible and easy to use, with a totally redesigned user interface giving you even greater control over the compiled program. You can now fine-tune throughput and output in the following ways:
* Code address size (16bit or 32bit) * Object size (SHRT, CODE) * Optimisation (BRIEF, FAST or BOTH for locally switchable optimisation on a statement by statement basis) * Line numbers (ignore, display only or include) * Diagnostics on/off * Source format (freeform or structured – the latter for those who write structured BASIC programs). TURBO V2.0 providing them with the big benefits of both faster compiling and greatly enhanced execution speed, never before available to compiler users; the former where TURBO is much more relaxed about the number of programming atrocities that it permits users to get away with!) * Local strings (ignore, report or create) * GRAD & Pointer environment (yes or no) * Number of windows to copy * Dataspace * Buffer size * Output filename * Medium * Report filename or destination * Listing on/off * Pause on/off * Sound on/off * Listing character width & height * Number of listing lines (dynamically adjustable with user-definable keys) * Listing colour * Compiler resident * Output task name (as opposed to filename) * Default settings for most of the above

TURBO V2.0 is absolutely the last word in refinement and ease of use. It is now totally compatible with the TURBO and GRAD. The toolkit has been revised and expanded and now has 200 add-on commands, procedures & functions (most of which are listed on the right) to extend the power of SuperBASIC – all but half a dozen are available both under interpreter and TURBO! Now you realise why so many organisations have benefited from TURBO technology! QL WORLD • CST • CIDERSOFT • ERIK • PSION • PSION • DABSOFT • GARNER • IN WASDF • MB • MIRACLE • NICHOLE • SHANNON • TRANSTORM • CLWYD • ARC ELECTRONIC • RESSEARCH • SPELLBOUND • QUANTUM SERIES • QS ENTERPRISES • SMILING SOFTWARE •

Below we present the PCW benchmarks for TURBO V2.0, with source format left on default (FREEFORM): compiling this to STRUCTURED makes it go even faster!

STANDARD PERSONAL COMPUTER WORLD SPEEDUP BENCHMARKS	1	2	3	4	5	6	7	INT MATH	FPP MATH
TURBO vs SuperBASIC	34x	60x	40x	51x	40x	32x	41x	69x	22x

☆ Supremely SuperQuick (!) – TURBO introduces brave new technology/terminology (all = shorter code!), and between setting optimisations to global speed/size/short speed (see a statement block basis) & step & run with localised speed & error, a total of 15 options. You are in the driving seat. If you slip up, we give you a report that pinpoints your mistake, exact to the character!

☆ Supremely behaved Virtual arrays, sidebar arrays, implicit datatypes (accelerated, fully re-entrant integer FOR loops and integer/STRING SELECT are all 12...4), full WHEN – ERROR mapping of any error on any QL version to a single point (error handler) – you can have any number of them, hierarchically laid out – have full access to line and error numbers, and you can RETRY or CONTINUE RESTART as you choose even when in another task, parameters passing by reference as well as by value, 9 digits of displayed accuracy, etc.

☆ Supremely powerful TOOLKIT included, with TWO HUNDRED (!!!!) really useful commands, utilities and functions for the BASIC programmer. There is no space to even begin to list them – suffice it to say that TURBO TOOLKIT's most 'comprehensive' rival has little more than half as many features. The TOOLKIT (which is RAM based) is designed to complement existing toolkits (e.g. the Supertoolkit) as far as possible.

☆ Supremely documented – a THREE HUNDRED AND FIFTY A4 PAGE, lucid, illustrated user guide/reference

☆ Supremely generous – there are no royalty restrictions on TURBO'd programs

The TURBO System costs £99.95 COMPLETE

TURBO IS BRILLIANT. PRESS COMMENT FROM EVERY SOURCE IDENTIFIES IT AS THE MOST OUTSTANDING BASIC COMPILER FOR ANY MICRODCOMPUTER.
The TURBO TOOLKIT is useful on a stand-alone basis, and costs just £29.95. This amount will be credited if you subsequently purchase THE TURBO SYSTEM.
"You are getting great value, and there is a lot more . . . TURBO TOOLKIT is a comprehensive product which will cost many pay considering where your money went – it offers a wide range of facilities to everyone (whether or not they own a compiler)!" . . QL World, December 1986

ALLOCATION	CURSOR XY	G_SCALE	RELEASE BASIC	WINDOW ON
ARROW$()	DATASPACE	HEX$ (DEC)	RENUM BASIC	WINDOW$
BASIC BOOT	DATA SPACE	INSTR (SUB)	RUN BASIC	WINDOW$
BASIC DIR	DEC$ (HEX)	INSTR (SUB)	SCALE	WINDOW$
BASIC FLASH	DEFAULT DEVICE	INK COLOUR$	SCREEN$ (,)	WINDOW$
BASIC FUNCTS	DEVICE$ SPACE	INVERT MODE$	SCREEN$ (,)	WINDOW$
BASIC$(,)	DEVICE$ STATUS	INVERSE$	SD PROFILE	WINDOW$
BASIC NAME$	DEVICE$ STATUS	JOYSTICK$	SELECT ACCESS	WINDOW$
BASIC POINTER	DEVICE$ TIMEOUT	LINE CHANGE	SC CHANNEL	WINDOW$
BASIC RENUM	DEVICE$	LINE$ CHANGE	SEARCH MEMORY	WINDOW$
BASIC SEARCH	DIR$	LINE$ LOAD A	SHOW MUSIC	WINDOW$
BASIC SIZE	DIR$	LINE$ LOAD B	SIO CHANNEL	WINDOW$
BASIC TYPE$	DO CMD	LIST$ (STRING)	SKIP$ (N)	WINDOW$
BASIC VAR	DOS NAME	LIST$ (VAR)	SKIP$ (STRING)	WINDOW$
BASIC VARS	DOS$	LIST$ PROCS	SKIP$ (VAR)	WINDOW$
BOOT$ (FILE)	ENVIRON$	LIST$ VARS	SKIP$ (STRING)	WINDOW$
BORDER$ (,)	ERR$ (NR)	LIST$ VARS	SKIP$ (VAR)	WINDOW$
BOTH$ (,)	EXECUTE A	MACHINE CODE	SPACE$ (N)	WINDOW$
BOTH$ (,)	EXECUTE$ A	MOVE MEMORY	SPACE$ (N)	WINDOW$
BOTH$ (,)	EXECUTE$ B	MOVE RAM	SPACE$ (N)	WINDOW$
BOW POSITION	EXTERNAL (ADR)	NET NUMBER	SPACE$ (N)	WINDOW$
BUN MODE	EXTERNAL (ADR)	NET$ (NUMBER)	SPACE$ (N)	WINDOW$
CAPS LOCK	FADE (OFF)	OPEN CHANNEL	SPACE$ (N)	WINDOW$
CHR$ (N)	FILE SIZE	OPTION (ON/OFF)	SPACE$ (N)	WINDOW$
CHANNEL (CHAN)	FILL	PAPER (N)	SPACE$ (N)	WINDOW$
CHANNEL$ (N)	FILL	PAPER (N)	SPACE$ (N)	WINDOW$
CHANNEL	FLIP	PEEK$ (N)	SPACE$ (N)	WINDOW$
CLEAR ALARM	FLOAT	PEEK$ (N)	SPACE$ (N)	WINDOW$
CLOCK	FLOAT	PIPED MUSIC	SPACE$ (N)	WINDOW$
CLOSE (CHAN)	FONT$ (N)	PLAIN FUNCTION	SPACE$ (N)	WINDOW$
CONTROL (CHAN)	FONT$ (N)	POKE$ (N)	SPACE$ (N)	WINDOW$
CONTROL (CHAN)	FORMAT (N)	POKE$ (N)	SPACE$ (N)	WINDOW$
CONTROL (CHAN)	FREEZE SCREEN	POKE$ (N)	SPACE$ (N)	WINDOW$
CONTROL (CHAN)	FREE (CHAN)	POKE$ (N)	SPACE$ (N)	WINDOW$
CONTROL (CHAN)	GRAB