DTL-BASIC

64

Tapper

DISK VERSION

Dataview

WORDCRAFT

Copyright

License

Drive Technology Limited grants the registered user the right to distribute the compiled programs produced by DTL-BASIC 64 without payment of royalties provided that the following copyright notice is clearly included in the distribution media

" Parts of this product are copyrighted by Drive Technology Ltd., 1983"

A further condition of license is that DTL-BASIC 64 may only be used in conjunction with the security key supplied with the product or with additional keys supplied by the Distributors.

Notice

Drive Technology Ltd. shall not be liable for any loss or damage resulting from the use of DTL-BASIC 64 or for incidental or consequential damages in connection with the furnishing, performance, or use of this product.

Drive Technology Ltd., reserves the right to alter this product without notice and without the obligation to notify any person of such alterations.

Unless the Distributor is contacted in writing within 10 days from the shipment of this program, it shall be assumed by the Distributor that the registered user has read and fully accepted the above conditions.

Published by:
Dataview Wordcraft Limited,
Radix House,
East Street, Colchester
Essex, C01 2XB, England
Tel: Colchester (0206) 869414
Telex: 987562

Issue 4: June 25th 1984.

Contents

1. Introduction.

DTL-BASIC 64 is a Basic compiler for the CBM 64. This manual is for use with DTL-BASIC 64.4 (ie. release 4), or a later release.

The function of a compiler is to convert a program from its source form (ie. the form in which it is written) into a more efficient form that can run much faster than the original.

DTL-BASIC 64 has been specially optimised for the CBM 64 and it not only makes every Basic program a lot faster but it will also make each program significantly smaller; except for programs of only a few lines.

DTL-BASIC 64 is 100% compatible with CBM 64 Basic. This means that an existing Basic program can be compiled, without any alteration, to produce a program that performs exactly the same, and yet is very much faster and requires less memory and disk space.

DTL-BASIC 64 is designed so that it can be used by people with no programming knowledge to compile existing programs; and yet for more experienced users a range of facilities is provided to enable the full potential of the CBM 64 to be realised.

DTL-BASIC 64 can be supplied either on tape or on disk. The disk versions will work either on a disk unit attached to the serial port or with one attached via an IEEE-488 interface.

Note : a more detailed description of the differences between a compiler and an interpreter is given in Appendix A.

1.1 Versions available.

Versions of DTL-BASIC are available for the whole range of CBM machines with the exception of the VIC 20.

For the CBM and SX64 there are three versions -

DTL-BASIC 64 JETPACK (Tape)
DTL-BASIC 64 JETPACK (Disk)
DTL-BASIC 64 PROTECTOR

The tape version is for use on CBM 64 machines without a disk drive, ie. with only a cassette tape unit for program storage.

The other two versions are for machines with a disk drive and are identical except that the PROTECTOR version is for use by software houses and provides protection for compiled programs against illegal copying.

This manual applies to all versions and except for where it explicitly states otherwise the three versions are identical.

1.2 Purpose of this manual.

This manual describes how to use and operate all the versions of DTL-BASIC 64.

No attempt is made to teach Basic programming or to define the Basic language. This is not necessary due to the high level of compatibility with the Basic interpreter in the CBM and SX64.

1.3 How to use this manual.

This manual is intended for use both by programmers and by non-programmers.

Before using the compiler all users should read chapters 2,3 and 4 (covering installation, main features and operation).

The operation of DTL-BASIC 64 JETPACK (Tape) is slightly different from the disk based versions and there are separate sections within chapters 2,3 and 4 specially for the tape based version. This version also has some limitations when compared to the disk versions and these are identified in section 3.6.

It is especially important that the user notes the potential problem of disk corruption that can occur when the replace option is used with the SAVE command (see section 4.10). This is a problem in the DOS in the disk drive and is nothing to do with the compiler. However, if the compiler is used on a corrupt disk then it may appear that the compiler is not working correctly. Therefore, avoid using the replace option with SAVE.

If the program to be compiled is a single program consisting solely of standard Basic, ie. not involving any machine code and does not chain in any other program then the remaining chapters can be left until they are required.

If two or more programs are chained together then chapter 7 should be read before compiling; similarly, if extensions to Basic, machine code or cartridge based products are involved then refer to chapter 8.

Chapter 5 explains how to get the greatest benefit from DTL-BASIC 64.

It is important to realise that, whatever speed improvement is achieved by compiling a program without any alteration, it is almost certain that very significant additional improvements can be gained by making slight changes. Often, this only involves the addition of a single compiler directive at the front of the program. The reasons for this are given in chapter 5 and chapter 6 describes the directives that are

available. A compiler directive is an instruction to the compiler stored within the source program.

If any errors occur while compiling or running compiled programs then refer to chapter 9.

1.4 Major benefits of DTL-BASIC 64.

* compiled programs can run up to 25 times faster in ideal situations - typical improvements are normally in the range of 5 to 15 times faster;

* compiled programs are between 50% and 80% of the size of uncompiled program which means not only that there is more space for variables and arrays but that programs will load faster and will need less disk space;

* compiled programs cannot be listed or altered;

* the compiler is totally compatible with all the features of CBM 64 Basic meaning that a Basic program can be compiled without alteration;

* the compiler provides true integer arithmetic as well as floating point arithmetic;

* the compiler is compatible with existing machine code routines, ie. where a Basic program uses separate machine code then that code should work with the compiled Basic program without alteration;

* the compiler can compile programs incorporating extensions to Basic; ie. additional Basic statements implemented by machine code in ROM or RAM;

* the compiler is especially effective for compiling games programs involving graphics. This is because special attention has been paid to making the statements used to control sprites as fast as possible. A special directive is available to facilitate fast sprite movement (see sections 5.1 and 5.2);

* great flexibility is offered to the programmer who produces sophisticated suites of programs. For example, the start address of the variable list can be defined by the programmer and when chaining several programs then the variables may, or may not be, shared between the separate programs as required by the programmer. By sharing variables time can often be saved by not having to re-load information from disk;

* DTL-BASIC 64 has its own garbage collection routine which takes less than a second (on the interpreter programs involving a lot of string processing can experience long delays due to the slow garbage collection routine);

* a facility is provided to ease the transportation of programs from other machines to the CBM and SX64. This enables PEEK/POKE addresses to be automatically adjusted without changing every statement. This can be very useful in many situations, eg. when a program POKEs data directly to the screen.

2. Installation of DTL-BASIC 64.

2.1 Contents.

Your copy of DTL-BASIC 64 Jetpack should consist of :

- a DTL-BASIC 64 Jetpack compiler disk
- this manual
- a security key (not necessary for tape)

2.2 The security key.

The security key is a device designed to protect Drive Technology's copyright. It is only by protecting the product well that such a sophisticated product can be made available at such a low price.

There are two keys available, one can be installed on the cassete port with the lettered side facing upwards,this is for use with the CBM 64; and another which locates in joystick port 2, this would be used on either the CBM or SX 64 which does not have a cassette port.
If the key is not present, or is not fitted correctly, then when the compiler is run the machine will be reset, ie. the screen will revert to the state after power up.

2.3 Making backup copies.

Before using the compiler for the first time it is essential to make at least one copy. This protects you in case your disk becomes corrupt for any reason.

Always use the copy disk and store the original in a safe place.

2.3.1 Disk versions.

The procedure for making a copy of the compiler disk on a single drive disk unit is

- format a spare disk by means of a NEW command, eg.

```
OPEN 15,8,15
PRINT 15, "NØ:DTL-BASIC,C1"
```

Note : make sure that the identifier ("C1" in this example) is not the same as the compiler disk identifier, ie. is not "64".

- transfer the six files from the compiler disk to the copy disk; the six files are -

DTL-BASIC
DTL-BASIC-E (for use with DAM's IEEE cartridge)
RTL-64
DTL-BASIC-MC

DTL-BASIC-MCE (used with a DAM's IEEE cartridge)
ERROR LOCATE

The procedure for copying a file from one disk to another is -

- insert the disk containing the file in the disk unit

- load the file to be copied with a LOAD command, eg.

```
LOAD "DTL-BASIC",8
```

- remove the disk and replace with the second disk

- save the file with a SAVE command, eg.

```
SAVE "DTL-BASIC",8
```

The procedure for making a copy on a dual drive disk unit is

```
OPEN 15,8,15
PRINT  15, "D1=Ø"
```

ie. Duplicate drive Ø onto drive 1.

3. Main features of DTL-BASIC 64 Jetpack.

3.1 Requirements.

The tape version requires a standard CBM 64 plus a cassette tape unit. The other versions require a standard CBM 64 plus a disk drive (either a single or a dual drive), or a standard SX 64.

The compiler can make use of a printer but one is not essential.

Neither the compiler nor compiled programs use a ROM cartridge so that the user is free to use cartridge software together with compiled programs, although in some cases the cartridge programs may relocate and conflict with the compiler.

3.2 The compilation process.

DTL-BASIC 64 is complementary to the CBM Basic interpreter due to its total compatabilty. This means that programs can for convenience be developed and debugged on the interpreter and when working can be compiled for maximum speed and to reduce program size.

It is usual before compiling to make sure that the program works on the interpreter. DTL-BASIC 64 does make thorough checks for errors both at compile time (ie. whilst the program is being compiled) and at run-time (ie. when the compiled program is being run) but it is more convenient to detect and correct errors on the interpreter, rather than the compiler.

The compilation process involves :

- loading and running the compiler;

- telling the compiler the name of the source file; ie. the file containing the program to be compiled;

- telling the compiler the name for the object file; ie. the file to be created by the compiler to hold the compiled program;

- compilation is a two pass process; on the first pass the source file is read a line at a time and a semi-compiled version of the program is written to a work-file. On the second pass the work-file is read back, additional information is added and the object file is created. Note that for DTL-BASIC 64 T the work-file is held in memory in the area unused by Basic.

After the compilation is complete the work-file is deleted, and the object file may then be loaded and run, or another program may be compiled.

Note that, for the disk based compilers, because the

program is never totally held within the compiler there is no limit to the size of program that can be compiled. Any program that will run on the interpreter should be able to be compiled. For the tape based version there is a limit on the size of program that can be compiled - see section 3.6.

If required, the compiler can produce a listing of the program and/or a report of any errors found.

In addition to the compiled program the compiler produces a file LN-name (where 'name' is the name of the compiled program). This file is not involved in running the compiled program but is only needed if the program has run-time errors (eg. DIVIDE BY ZERO ERROR) that were not found before compilation - see chapter 9.

3.3 The run-time library.

The run-time library (file RTL-64) is a set of machine code routines that must be in memory whilst a compiled program is run.

It is not necessary for the user to load this file as, every time a compiled program runs, the program first checks to see whether the run-time library is in memory and, if it is not, then it will load the file automatically from either disk or tape, ie. from wherever the program was loaded. This means that the disk or tape from which the first compiled program program is loaded after power up should contain a copy of RTL-64.

The run-time library is just less than 8K bytes in size but in order to avoid using up the valuable space within the Basic area, ie. the 38K available to Basic, the run-time library is stored outside this area in some of the RAM that would otherwise be unused.

For the benefit of machine code programmers, the run-time library is stored in the 8K from $A000 to $BEFF which is an area of RAM that cannot be directly accessed from Basic. This leaves the 4K of accessible RAM at $C000 (ie. RAM that can be accessed via SYS,PEEK and POKE) free for machine code and/or data (see sections 8.2 and 8.3).

3.4 Combining Basic and machine code.

Many Basic programs utilise machine code subroutines to perform tasks that are not possible or are difficult in Basic. With the greatly improved performance provided by DTL-BASIC 64 it is possible to replace many machine code routines with Basic code.

However, there will always be situations where some machine code is desirable. DTL-BASIC 64 has been

especially designed to ensure that in the vast majority of cases machine code that works with a Basic program on the interpreter will also work without alteration with the compiled program. This is possible because the compiler preserves precisely the same format for page zero, the variable list, the array list and string storage etc.

This means that machine code that, for example, searches the variable list for a particular variable or sorts a string array, will still work with a program compiled by DTL-BASIC 64.

Further details are given in chapter 8.

3.5 Extensions to Basic.

One very useful feature of CBM machines is the way that it is possible for additional features to be added to the Basic by means of machine code routines either in RAM or ROM (eg. ROM in a plug-in cartridge).

DTL-BASIC 64 has features that enable programs using such extensions to be compiled and run successfully even though the compiler does not know the details of the extensions. This means that programmers are free to use extensions to Basic and are still able to obtain all the benefits of compilation.

This is possible because when the compiler checks the syntax of a statement then if it cannot recognise the first character of the statement (ie. if the character does not start with either a legal keyword or an alphabetic character) the compiler assumes that the statement is valid but is an extension to standard Basic.

The compiler embeds the text of the extension statement in the compiled program exactly as it occurs in the source program, precedes it by a special code and follows it by a SYS call to the run-time library. When the program is run and the run-time library detects the special code, it sets up the page zero pointers to the extension statement and calls the interpreter to process it. The interpreter processes the statement as though it was in a normal program and invokes the additional machine code to implement the statement. When the machine code routine returns control the interpreter obeys the SYS call and re-enters the run-time library.

The whole process can work because the machine code finds the variable and array lists etc. exactly as it expects.

See chapter 8 for further information as to how extensions (and SYS calls with parameters) are handled.

3.6 Tape version - facilities available.

The tape based version of the compiler provides a subset of the facilities of the disk based compilers.

The main restriction is the size of program that can be compiled. This is because the whole program is held in memory during compilation which means that the largest program that can be compiled by the tape based version is 12K bytes (ie. 12288 bytes). If an attempt is made to compile a program that is too large then the compiler will stop and give an error message.

The only other restrictions are that the directives VL,RO and VN cannot be used with the tape based compiler (see chapter 6) and the control file facility cannot be used (see section 4.9).

4. Operation of DTL-BASIC 64.

4.1 How to run the compiler.

4.1.1 Disk versions.

The compiler disk actually contains two separate compilers; one for single drive disk units (eg. the 1540 or 1541 units) and one for dual drive disk units (eg. the 4040). The dual drive compiler will work with drives attached to the serial port or with drives attached via an IEEE-488 cartridge.

When the dual drive compiler is used then the compiler disk must be in drive 0 and the program to be compiled must be on the disk in drive one.

There are two ways of using the single drive compiler. The first is to copy the program to be compiled onto the same disk as the compiler (ie. by use of LOAD and SAVE commands). The second is to load and run the compiler and then remove the compiler disk and replace it with the disk holding the program to be compiled.

If there are a number of programs on the disk it is worth checking that sufficient free space exists for the compiled program and for the work-files used by the compiler. These files will be deleted at the end of the compilation but will require space until then. As a rough guide the free space available should be at least equal to the size of the source file for the dual drive compiler and at least twice the size of the source file for the single drive compiler.

It is possible to find the amount of free space by displaying the disk directory, ie. type the commands

```
LOAD "$",8
```

and when READY is displayed type

```
LIST
```

the size of each file and the free space on the disk are given in terms of the number of blocks (a block is 256 bytes).

If there is not enough free space some files will have to be deleted (after being copied to other disks).

Before running the compiler first fit the security key to the cassette interface (lettered side upwards) and then type

```
LOAD "DTL-BASIC",8
```

for the single drive compiler

or

LOAD "DTL-BASIC-E",8

for the dual drive compiler;

and when READY is displayed type

RUN

There will be a pause while the two files (RTL-64 and either DTL-BASIC-MC or DTL-BASIC-MCE) containing machine code are loaded to memory.

4.1.2 Tape version.

Throughout these instructions it is assumed that the user will obey instructions from the operating system to press keys on the tape unit, ie. the messages

PRESS PLAY ON TAPE
PRESS RECORD & PLAY ON TAPE

will not be explicitly mentioned in this manual.

To load the compiler first put the compiler tape in the tape unit and type

LOAD "DTL-BASIC"

and when READY is displayed type

RUN

If this is the first time that the compiler has been run since powering up the machine then there will be a delay while the file "RTL-64" is loaded.

If this is the first time that the compiler has been run since powering up the machine then there will be a further delay while the file "DTL-BASIC-MC" is loaded.

Having once used the compiler then as long as the machine is not turned off then the files RTL-64 and DTL-BASIC-MC will remain in memory (in the area unused by Basic) and will not have to be re-loaded when the compiler is next run.

4.2 Compilation options.

The compiler will display the following list of options

```
source file  ? :
object file  ? :
print source ? : n
print errors ? : n
print stats  ? : n
run identity ? :
```

plus a set of commands selected by the function keys.

Each option field that is input is terminated by RETURN or F1 (function key 1).

Type the name of the source file.

Type the name of the object file.

Unless any printing is required then the compilation may be started by F3.

If printing is required then change the relevant "n"s to "y".

If "print source ?" is "y" then the whole program will be printed during the compilation.

If "print errors ?" is "y" then any error messages will also be printed.

If "print stats ?" is "y" then at the end of the compilation some statistics will be printed giving the relative sizes of the source and object files.

If any printing is selected the contents of "run identity" will be printed at the start of the listing to serve as an identification, eg. it may be convenient to put in the date or time etc. so that when several listings of the same program are kept then the correct sequence can be determined. The "run identity" field can be left blank if required.

The PROTECTOR version of the compiler has one additional option to those listed above, ie.

```
key identity  ? :
```

this specifies the key that is to be used with the compiled program; the possible inputs are -

```
"c"                - for the compiler key
"r"                - for the run-time key
the serial number  - for a software house key
```

4.3 Function keys.

As mentioned earlier F1 moves on to the next option and if there are no more options when it is pressed the compilation will start.

Alternatively as soon as both the source and object files have been named then F3 can be used to start the compilation immediately.

If the user cannot remember the name of the source file then F5 can be used on the disk based versions to display a list of all the program files on the disk.

If it is realised that an option has been input wrongly then F2 can be used to restart at the beginning.

If program to be compiled is on a different disk then F4 can be used to allow the disk to be changed without reloading the compiler.

F6 can be used to exit from the compiler without performing a compilation.

4.4 Compilation.

4.4.1 All versions.

When the compilation has started then if any printing is to be performed the compiler checks that the printer is ready. If it is not the message

**** FIX PRINTER ****

is flashed on the screen. The user can either select the printer or press space to continue without printing.

Note that on some machines there is a problem with the VIC 1515 printer that causes the system to hang up. If this occurs then it is necessary to turn off the printer and turn it back on again. At least one line of printing may be lost because of this.

During compilation the progress is recorded on the screen by displaying the number of the line being processed.

If the compiler detects any errors in the source program an error message will be displayed either on the screen or on the printer. If a number of errors are found then, when the screen is full of error messages the compilation will pause so that the lines in error may be noted before compilation resumes.

As well as error messages it is possible for warning messages to be displayed. These occur when the compiler believes that it has detected an extension to Basic but

may have found a syntax error. The reason for this is explained in section 9.4.

At the end of compilation counts of the numbers of error and warning messages are displayed and the compilation statistics are output.

4.4.2 Tape version.

The tape based versions of the compiler requires some additional operations by the user.

After the compilation options have been specified the message

```
ENSURE TAPE CONTAINING "source file name"
IS IN TAPE UNIT

PRESS SPACE TO CONTINUE
```

is displayed and the compiler tape may be rewound and removed. The tape that contains the program to be compiled should then be installed in the tape unit. When SPACE is pressed then the compilation will start.

When the program has been compiled the message

```
PRESS STOP ON TAPE UNIT

ABOUT TO CREATE "object file name"
ENSURE CORRECT TAPE IN TAPE UNIT

PRESS SPACE TO CONTINUE
```

is displayed.

The user now can leave the existing tape in the tape unit, in which case the compiled program (the object file) will be written to the tape immediately behind the uncompiled program (the source file).

Alternatively, the existing tape may be removed and replaced by a blank tape; in this case the compiled program will then be the first file on that tape.

In either case make sure that the STOP key is pressed on the tape unit before SPACE is pressed.

After the compiled program has been written to tape the user is asked

```
CREATE LINE NUMBER FILE ?
```

ie. should the compiler create a Line Number file (the LN file). If the answer is "y" then the LN file will be written to tape following the compiled program. This can take some time for a large program so that it may

be best to only create an LN file if it is needed, ie. if the compiled program gives a run-time error (see chapter 9). The LN file is used by the ERROR LOCATE program to find the line number upon which the error occurred.

4.5 Compilation statistics.

The compilation statistics produced at the end of the compilation give the sizes of :

- the source program
- the object program
- the object file

The sizes are given in terms of the number of bytes and also in terms of the number of blocks and the number of bytes in the last block (a block is 256 bytes), eg.

SOURCE PROGRAM SIZE - 4253 (16,157)

ie. 4253 bytes is 16 blocks plus 157 bytes (which would require 17 blocks of disk space).

The program sizes are the amounts of memory occupied when the program is run. The two sizes can be compared to see what size reduction has been achieved.

The object file size exceeds the object program size because the file normally holds both the program and the variable list. By comparing the file size with the program size, the size of the variable list can be determined. Note the the variable list holds all the normal variables but not the arrays. The arrays are created dynamically at run-time.

4.6 Termination options.

If any errors were detected during compilation then the object file is not created and the source file will have to be editted to correct the errors before it can be compiled.

If there were no errors the the user has three options

- to key "C" to compile another program;

- to key "L" to load and run the program that has just been compiled;

- to press any other key to exit from the compiler;

Note that if the tape version is being used then the option to compile another program is especially useful as it enables a number of programs to be compiled without having to reload the compiler. Also, the facility for automatically loading the compiled program should only be used with the tape version if the

compiled program is the first one on the tape. If this is not the case then exit from the compiler and load the compiled program by means of a LOAD command.

4.7 Operation of compiled programs.

Operation of compiled programs is identical to that for uncompiled programs, ie. compiled programs are simply LOADed and RUN just like uncompiled programs.

Compiled programs should perform exactly like uncompiled programs - if they do not then refer to section 4.10.

The first time a compiled program is run after the CBM 64 has been turned on there will be a delay while it loads the file RTL-64. Each subsequent time that a compiled program is run then there will not be a delay because the program will detect that RTL-64 is already in memory.

If a compiled program has been loaded from tape and RTL-64 is not in memory then when the program is run it will load RTL-64 therefore either RTL-64 should be on the same tape or the tape will have to be changed for one which contains a copy of RTL-64.

CONT cannot be used with compiled programs. SYS 2061 should be used instead of CONT.

When a compiled program is stopped then variables and array elements can be printed on the screen (for debugging) as with interpreted programs.

4.8 Making copies of compiled programs.

If it is required to move a compiled program to another disk use LOAD and SAVE as for uncompiled programs, eg.

```
LOAD "program name",8

change disk

SAVE "program name",8
```

Note that a compiled program should not be SAVEd after it has been run. Do not forget that a copy of "RTL-64" is normally needed on each disk containing compiled programs.

Copies of compiled programs on tape can be made in a similar manner, eg.

```
LOAD "program name"

change tape

SAVE "program name"
```

Compiled programs on disk may also be copied to tape by means of LOAD and SAVE. Do not forget that any program that may be run immediately after power up should be

followed on the tape by a copy of "RTL-64".

To save time when using programs loaded from tape it may be convenient to have one program that is loaded and run whenever the machine is turned on. This program should be followed on the tape by "RTL-64" and then all other compiled programs need not contain RTL-64 on the tape since whenever they are run "RTL-64" will already be in memory.

If programs being copied involve chaining or have a separate variable list (see section 6.4 and chapter 8) then do not forget to copy the VL file (refer to last paragraph of 6.4 before copying the VL file). Note that the order of files on tape should be

- the compiled program
- RTL-64 (this file is optional)
- the VL file (eg "vl-abcd" where "abcd" is the name of the compiled program)

4.9 Special operation features.

There are two special features designed to make the operation of the compiler even easier.

The first is invoked if the source file name has the last four characters equal to "-src". In this case the object file name will be generated automatically, eg.

if the source file name is

"abcd-src"

then the compiler will call the object file

"abcd"

This feature can best be used by renaming all source files to have the "-src" suffix as this will ensure that the compiled programs will then have the name that the user is familiar with. This is especially useful when program chaining is used (ie. when one program LOADs another program) as otherwise the LOAD statement within the program would have to be altered.

The second special feature is available only on the disk based versions and can be used when a number of programs on the same disk are to be compiled. Rather than compiling each program separately a control file can be used to give the compiler a list of the programs to be compiled. The programs will then be compiled without any further action by the user.

A control file is a normal file that has the last four characters equal to "-con", eg. "compile-con".

A control file is created and editted in the same manner as a program

file and consists simply of a list of file names. Each file name should be on a separate line and the first character of each line should be a quote character (").

The first file name should be the name of the first source file to be compiled and the second file name should be the name of the corresponding object file. The next file name should be the name of the second source file to compile and so on . . .

If the "-src" option is used the the object file name is omitted.

eg. A typical control file could be -

```
10 "file1"
20 "cfile1"
30 "file2"
40 "cfile2"
50 "test-src"
```

(the trailing quote on each line is optional)

In this case three compilations will occur, ie.

"file1" will be compiled to give "cfile1"
"file2" will be compiled to give "cfile2"
"test-src" will be compiled to give "test"

To start the compilation the name of the control file should be given instead of the source file name. The printing options selected (and the key option for the PROTECTOR version) will apply to all compilations. If a printer is available then it is recommended that the option to print errors should be selected to ensure that any errors are not lost.

4.10 Problems ?

If a compiled program does not appear to be running exactly like the interpreted version it is likely that the Special Integer mode must be selected. This is done by means of the SI directive which is explained in more detail in section 6.3.

If the machine is reset to the power up state when the compiler is run then check that the security key is fitted correctly.

If the compiler stops during compilation when the 1515 printer is in use then refer to section 4.4.1.

If a compiled program using either the VL or RO directive at the start of the program crashes when run then check that the VL file is present on the disk. Check also that the VL file has not been renamed.

If a program using a VL file does not work after being copied onto a disk or tape then check that the first variable in the program has only a single character name (see section 6.4).

A compiled program should not be SAVEd to create a new copy once it has been RUN.

If the compiler stops during a compilation on the 1540 or 1541 drives with a 'NO CHANNELS ERROR' or halts with an error indicated on the disk drive the reason is actually a read or write error. The wrong error message is due to a bug in the DOS within the disk drive that means that when an error occurs then if further characters are read or written before a test for an error is made then the wrong error message is generated. The compiler cannot check for an error after every character is read or written because this would slow down disk i/o by a factor of three or four.

If the 'NO CHANNELS ERROR' occurs on a drive that normally does not give any trouble then it is likely to be for one of two reasons. The first is that it is simply a bad disk that should be replaced by one of better quality. The second reason is that the disk may have been written on a different drive (eg. a 4040) that is apparently compatible. Although such disks can be read on a 1540 or 1541 they do appear to be more susceptible to errors than ones written on the same drive. If this is the case make a new copy of the disk on the drive upon which the compilation it to take place.

The 1540 and 1541 can also corrupt files on occasions so take care to have copies of all files and use VALIDATE frequently to ensure that the disk is in a good state. If a program becomes corrupt then perform a VALIDATE and copy the file from a backup. Avoid using

the replace option (@) with the SAVE command as its repeated use can cause corruption. Instead, when editing a program then SCRATCH the old copy and use SAVE without replace to create the new file.

Some Basic programs are 'patched' in a special way by the programmer so that after loading they will run automatically, ie. without RUN being typed. Such a program cannot be compiled directly but if the un-patched program is compiled then it ought to be possible to apply the patch to the compiled program.

Uncompiled programs can load compiled programs but it is not possible for a compiled program to directly load and run an uncompiled program via a LOAD statement within a compiled program. However, this will work if the LOAD statment is obeyed outside the program. One way of doing this is shown in the following sequence which will load the uncompiled program "TEST".

```
1000 PRINT "<cls><home>LOAD "CHR$(34)"TEST"CHR$(34)",8"
1010 POKE 198,6:REM SET BUFFER LENGTH
1020 DATA 19,13,82,85,78,13:REM <home><cr>RUN<CR>
1030 FOR I=1 TO 6 :READ X: POKE 630 + I,X :NEXT
1040 NEW
```

<cls> is the clear screen character
<home> is the home character

Some Basic programs POKE the address of the start of variables (45,46 decimal) to move the variables higher up the memory. Such POKEs are not necessary in compiled programs and may cause the program not to work (see section 6.4 and chapter 7).

5. Making the most of DTL-BASIC 64 Jetpack.

5.1 Achieving the best performance.

Any program that has been compiled without any alteration to the source file will run significantly faster than on the interpreter. However, it is very likely that by making one or two simple changes that considerable additional improvements can be achieved.

The reason for this is that DTL-BASIC 64 supports integer arithmetic as well as floating point arithmetic. Integer operations are used for all operations when both operands are integer. This applies to all arithmetic, logical and relational operations.

Integer arithmetic is many times faster than floating point, and to achieve the best performance as much use of integer arithmetic should be made as possible.

It is important to realise that, although the interpreter supports integer variables, it does not do any integer arithmetic. All integers are converted to floating point before any arithmetic operation. For this reason few existing programs make extensive use of integers.

Obviously, when writing new programs that are to be compiled, integers should be used as much as possible.

In order to save a user the trouble of having to work through and edit an existing program to change real variables to integers, DTL-BASIC 64 provides a way of automatically changing either all variables to integers or certain specified variables. This is achieved by means of the CS and CE directives which are described fully in the next chapter.

All the user has to do is work through the program and decide which variables have got to be floating point; ie. any variables which may hold a value greater than 32767, less than -32768, or which needs to hold numbers with a fractional part, cannot be integers. All other numeric variables can be converted to integers and the speed up improvement can in some cases be dramatic.

The overall speed improvements that can be achieved can vary considerably between different programs. There are three main reasons for this :

- when a program is performing I/O (input/output) then the program can spend most of its time waiting for the peripheral, eg. disk or printer. This waiting time can be so great that even if the statement processing time is many times faster, the overall speed improvement will be not nearly so great;

- the performance of a program on the interpreter can

depend tremendously upon how the program is written. For example, a routine at the front of a large program can run several times faster than a similar routine at the end of the program. When compiled, both routines will take the same time, but the relative speed up factors will vary considerably.

- some programs have to do a lot of floating point arithmetic, eg. statistical programs and ones making extensive use of the trig functions (SIN, COS etc.). Such programs cannot make as much use as normal of integers. However, there will almost always be some variables that can be converted, eg. variables used to access arrays.

5.2 High speed sprite movement.

One common situation where high performance is required is when moving sprites in game and graphics applications, or when POKEing characters directly to the screen. It is worthwhile paying particular attention to the POKE statements involved and especially those that are obeyed many times.

For example a typical statement might be

```
POKE G + 3, YP
```

where G could hold 53248 (the address of the display chip)

Such a statement could be moving a sprite, and may be in a FOR loop, and will probably be obeyed many times. In a compiled program the time for the floating point addition will far exceed the time to do the POKE. A far faster version would be to place a statement outside the loop such as

```
GA = G + 3
```

and change the statement in the loop to

```
POKE GA, YP%
```

However, this is still not as fast as can be achieved, because GA is a floating point variable, and each time the statement is obeyed it has to be converted to integer, which again takes much longer than the POKE. GA cannot simply be made integer because 53248 is too big. DTL-BASIC 64 has a feature to overcome this problem called Special Poke mode which is controlled by the SP and NP directives (described in section 6.6).

Special Poke mode enables an offset to be applied to all subsequent POKEs and PEEKs. In this case the offset will be 53248 so that each POKE can now use an integer.

This means the earlier statements can become -

```
GA% = 3
```

outside the loop and

```
POKE GA%,YP%
```

inside the loop.

Such minor changes can have a dramatic effect on the performance of programs making extensive use of PEEKs and POKEs.

Note also that disabling the stop key can also give a small additional performance improvement - see section 6.5).

5.3 Improved programming style.

One benefit of using DTL-BASIC 64 which is not immediately obvious is that it is possible to write programs that are easier to understand and to modify.

The reason for this is that, in order to get the best performance on the interpreter it is necessary to employ techniques that are bad programming practice, eg.

- not using many REM statements;

- using each variable for many tasks (to reduce the time spent searching the variable list);

- putting several statements on each line (to reduce the time spent searching for line numbers);

- placing the most frequently used statements at the front of the program.

These techniques (and others) can speed a program up on the interpreter a certain amount but they do lead to programs that are almost incomprehensible.

If a program is to be compiled, then none of these techniques are necessary, and the programmer can concentrate upon producing well structured, clearly understandable programs. This saves programming time in the first place, and when a program is later modified then the task is much easier.

5.4 Utilising the extra memory.

When a program is compiled then the reduction in size of the program can be considerable. This means that it can often be worthwhile increasing the size of arrays

to utilise the extra space, or to keep more information in memory to reduce the amount of disk I/O required.

However, it is always convenient to be able to run the same program on the interpreter when debugging, and if arrays are larger, or if there are more arrays, then an 'out of memory error' is possible. A simple way round this is to make the program detect whether it is compiled or not and to act accordingly. The way to do this is to check the first byte of the first line of the program. In a compiled program this byte will always be a SYS token (158 decimal), eg.

Place the following statement near the start of the program -

```
CP% = 0 : IF PEEK (2053) = 158 THEN CP% = 1
```

CP% can then be tested easily when required, eg.

```
A% = 1000 : IF CP% <> 0 THEN A% = 2000
DIM X(A%)
```

6. Compiler Directives.

A compiler directive is an instruction to the compiler stored within the source file. The directives have the form of a REM statement so that a program containing directives may still be run on the interpreter. The format of a directive is

REM ** <directive id> <directive text>

This format has been chosen to minimise the chance that an existing REM will be seen as a directive by the compiler.

<directive id> is a two character identifier.

<directive text> is additional information (not always present) - see the individual directive descriptions.

Most directives can only occur at the start of the program (ie. before any non REM statements) and will be ignored elsewhere in the program. However, some directives can occur anywhere in the program and these are indicated by an asterisk (*) in the list below.

The directives VL,RO and VN are not available on the tape based version.

6.1 List of directives.

Directive	Name	
CS	Convert Specified (for integer conversion)	
CE	Convert Excluding (for integer conversion)	
SI	Special Integer mode	
VL	Variable List address	
RO	Root program (for chaining)	
VN	Variable Name file (for chaining)	
DS	Disable Stop key	*
ES	Enable Stop key	*
SP	Special Poke mode	*
NP	Normal Poke mode	*
NW	No Warning messages	

The directives RO and VN are described in chapter 7.

6.2 Integer conversion directives.

These directives are used to tell the compiler which floating point variables and arrays are to treated as integers.

CS means Convert all the Specified variables to integers

CE means Convert all the floating point variables to integers Excluding those listed in the directive.

The CS or CE should be followed by a list of variable names in brackets with the names separated by commas, eg.

```
REM ** CS (A1,ZZ,X2,X3)
```

means convert all references to the names A1,ZZ,X2,X3 to integer, ie. the program will be compiled as though the variables were A1%,ZZ%,X2%,X3%.

```
REM ** CE (I1,I2,I3)
```

means convert all floating point variables to integer except I1,I2 and I3.

```
REM ** CE ()
```

means convert all floating point variables with no exceptions.

Note that both arrays and variables are converted, eg. in the first example, if there is a variable A1 and an array A1 then both will be converted.

The compiler will generate an error message if an integer already exists with the same name as a converted variable. In such a case it is possible to specify that the variable name is to be changed during conversion, eg.

```
REM ** CS (X,Y => YY%,Z)
```

will convert X and Z to X% and Z% respectively; but Y will be converted to YY%.

```
REM ** CE (A,B => B1%,C)
```

will convert all variables except A and C; B will be converted and will become B1%.

Note:

- that when changing name during conversion, the first character of the two names must be the same;

- CS and CE directives cannot both be used in the same program;

- there may be more than one CS or CE directive in a program, but the number of named variables cannot exceed 128;

Even for new programs there may be a need to use the CS or CE directives, because the interpreter does not allow integer FOR variables, even though in most programs FOR variables only ever hold integers. If it is required to debug the program on the interpreter

floating point variables must be used in FOR statements. When the program is compiled then CS or CE statements can be used to convert the FOR variables to integers. This will enable the best performance to be obtained.

6.3 Special Integer mode.

Special integer mode is selected by the directive

```
REM ** SI
```

This mode only affects the result of divide and exponentiation operations on integer operands.

The reason for this directive is that the compiler cannot always be sure what the programmer intends for these operators, when both operands are integer. This is because the normal action for the compiler to take when both operands are integer, is to perform an integer operation, because (as has already be explained) such operations are very much faster than floating point. With most integer operations there is no problem, but for divide and exponentiation the result can have a fractional part.

Consider the statement

```
A% = B% / 2 * 4
```

now if B% = 3 then if an integer divide is used the answer will be 4, but if a floating point divide is used the answer will be 6.

On the interpreter the answer will be 6, because all operations are floating point. For compiled programs in normal integer mode the answer will be 4, because in most situations when using integers the programmer expects integer operations and they are much faster.

However, occasionally, this can cause the compiled programs to work differently from the uncompiled program. In such cases the use of special integer mode will overcome the problem, ie. it will force the compiler to always use floating point arithmetic for divide and exponentiation.

6.4 Variable list positioning (disk versions only).

Normally the compiler places the variable list immediately behind the program, and the variable list is loaded together with the program from the object file. This is usually precisely what is required.

In some situations there may be a need to position the variable list higher in the memory, to leave space between the end of the program and the start of variables. Such space could for example be used for sprite data.

The VL directive can be used to achieve this and takes the form

```
REM ** VL <size>
```

where <size> is the size in bytes of the area between the start of the program and the start of the variable list. On the CBM 64 a Basic program starts at address 2049 ($0801 in hex) so that, for example, if the directive

```
REM ** VL 15000
```

is used, then the variable list will be placed at absolute address 15000 + 2049, which is 17049. If the program occupies 10450 bytes (obtained from the compilation statistics) then the free space between the program and the variable list will be 15000 - 10450, ie. 4550 bytes.

When the VL directive is used then the variable list will be stored in a separate file called "VL-ABCD"; where "ABCD" is the name of the program. The first time the program is run the VL file will automatically be loaded to the correct address. On subsequent runs of the program the file will not be loaded as the program will detect that it is already in the memory.

Some programs utilise POKEs to locations 45 and/or 46 to set the address of the variable list. Such POKEs are redundant in compiled programs. If a program does POKE different values to 45,46 from those set by the compiled program then problems are likely to occur.

If a program is involved in chaining and shares variables with other programs, then the VL directive should not be used because the RO directive achieves the same result.

Note that a problem can occur when copying a VL file to another disk or to tape. When the VL file is LOADed to memory prior to a SAVE, the system can corrupt the file. This occurs because it thinks the VL file is a program. The problem will not occur if the first variable used in the program has a single character

name.

6.5 Disabling the stop key.

The directive

REM ** DS

disables the stop key, whilst

REM ** ES.

enables the stop key.

When a program is RUN the stop key is initially enabled.

Programs run slightly faster with the stop key disabled.

On the interpreter the stop key is tested on every statement. For compiled programs in order to save time the stop key is only tested on NEXT and IF statements.

When a program uses LOAD to chain in another program, or to load some machine code etc. it is a good idea to disable the stop key for the duration of the load, because if stop is pressed in the middle of a load then the program probably will not be able to be restarted with a SYS 2061 (the compiled equivalent of CONT).

6.6 Special Poke mode.

Special poke mode allows an automatic adjustment of POKE (and PEEK) addresses from those specified in the program. There are a variety of situations where this can be convenient, eg.

- to avoid the use of floating point and thus improve performance (see 5.2 for an example of this)

- when a program has been developed on another machine for which the POKE addresses are different. This is most likely to be useful in programs that make many POKEs to the screen area which is at $8000 on most other CBM machines but is at address $0400 on the CBM 64.

Special poke mode is enabled by the directive

REM ** SP

and disabled by

REM ** NP

Before enabling the mode it is necessary to define the adjustment to be made. This is done by POKEing a value

(while in normal mode) to location 41028. When special poke mode is enabled this value will be exclusive-ORed with the high byte of the address used in any POKE or PEEK statements.

For example the statement

```
POKE 41028,208
```

sets the value to 208 ($D0 in hex). Now since the display chip starts at address 53248 ($D000) in hex) then when the special mode is enabled by

```
REM ** SP
```

a subsequent POKE such as

```
POKE 3,YP%
```

will actually write YP% to 53248 + 3 ($D003).

As another example, suppose a program written on another machine with the screen at $8000 hex was to be run on the CBM 64 (where the screen is at $0400), and the program POKEs information directly at the screen.

To handle this case the special poke mode value should be $84 (132 in decimal). This is because the result of exclusive-ORing $80 with $84 is $04. The easy way to think of it is, a bit set to one in the poke mode value inverts the corresponding bit in the address, whilst a zero leaves the corresponding bit the same.

Therefore, in order for the POKE statements to work on the CBM 64, all that is necessary is to put

```
POKE 41028,132
REM ** SP
```

at the start of the program after whatever POKEs are required to select the colour desired.

6.7 Inhibiting warning messages.

When a program uses extensions to Basic (see section 3.5) then for each extension a warning message is normally generated. Such warnings can be inhibited by the use of the directive

```
REM ** NW
```

7. Chaining programs.

The term chaining is used to describe the practice where one program loads another program on top of itself by means of the LOAD statement. After the load the new program runs automatically.

If a set of programs utilising chaining are to be compiled then there are two possible courses of action. The programs can either be compiled to share variables or not to share variables.

Sharing of variables occurs when a program is written to access variables set up by a previous program, ie. the variables and arrays are preserved when the program is changed.

Some chained programs do not share variables and in such cases each program will normally start with a CLR statement to get rid of the existing variables.

One common practice when chaining is for the first program in the chain to POKE values into locations 45,46 which hold a pointer to the start of variables. This is done to leave space for later programs in the chain which are larger than the first. Such POKEs are not necessary for compiled programs, and may in fact cause the program not to run. In such cases, the statements can either be removed, or made conditional upon whether the program is compiled or not by using the technique described in section 5.4.

7.1 Chaining without sharing variables.

In this case no special action is necessary in addition to possibly removing some POKEs as mentioned above.

Each program is simply compiled as normal and each object file will contain its own variable list as well as the compiled program.

7.2 Chaining with sharing variables (disk versions only).

If variables are to be shared then the use of the directives RO and VN are necessary. This is necessary so that when each program is compiled the compiler can be made aware of the variables used in the other programs.

The first program in the chain should start with the directive

```
REM ** RO <size>
```

where the function of <size> is the same as for the VL directive (see section 6.4 - all points made about VL also apply to RO), ie. it defines the size of the largest program in the chain and thus the position for

the variable list. Note that it is a good idea for the value of <size> to exceed the largest program size by a certain amount to allow for program modifications.

The RO directive tells the compiler that it is compiling the root program of a chain and has the effect that, at the end of the compilation a VN file will be created that records all the variable and array names used and the addresses allocated to them. A VL file will also be created holding the variable list.

The name of the VN file will be "VN-<name>" where <name> is the name of the root program.

All the other programs involved in the chaining that are to share variables should start with the directive

```
REM ** VN "<name>"
```

where <name> is the name of the compiled root program.

The effect of the VN directive is to cause the compiler to read in the specified VN file containing all the variable names and addresses.

At the end of that compilation, if the program used any new variable names, a new VN file will be created that includes the new names.

When the root program is run the VL file will be loaded to the address defined by the RO directive. The program may then be overwritten by other programs as many times as required and each will share the same variable list that will remain in memory the whole time.

Note that there is one restriction on programs that contain the RO and VN directive, and this is that DIM statements must exist for all arrays that are dimensioned in that program, ie. arrays without DIM statements will not be automatically dimensioned to have 11 elements. The compiler will give an error message for any array which does not have a DIM statement and which did not occur in the VN file read in at the start of the compilation.

To summarise, the first program in the chain should include a directive such as

```
REM ** RO 22000
```

where the largest object program in the chain does not exceed 22000 bytes. All other programs that may be chained and share variables, should include a directive

```
REM ** VN "MENU"
```

where "MENU" is the root name.

8. Information for users of machine code with Basic.

Many Basic programs utilise machine code. The machine code may held in RAM or ROM (eg. it may take the form of a plug in cartridge). In general such machine code will work unchanged with programs compiled by DTL-BASIC 64. This chapter aims to provide enough information so that a programmer using machine code together with Basic can ensure that the program works as intended.

There are several ways of getting machine code into memory, eg.

- loading from a file to $C000 - $CFFF;
- loading from a file to top of Basic memory;
- via a plug in ROM chip;
- via POKE statements from code stored in DATA statements to an area outside the program;
- via POKE statements from code stored in DATA statements to an area within the program (eg. to a REM statement);

Of all these techniques problems are only likely with the last one (because REM statements are removed by the compiler). Machine code must be stored outside of the compiled program.

8.1 Variable list and array list formats.

Many machine code routines access the variable and array lists to pass data to, and from, a Basic program. DTL-BASIC 64 Jetpack creates lists in exactly the same formats and using the same page zero pointers as the interpreter. This means that the machine code routines should work without alteration.

There are just a couple of points to watch out for.

The first is that it is possible for the order of variables in the list to differ from the order of variables when the program is run under the interpreter. The variables will be in the order that they occur in the source listing rather than the order in which they are referenced at run-time.

The second point concerns the array list; again the order of entries may be different and there will be one additional array. This will be the first array in the list and its name consists of two null characters so that a routine searching for a particular array will work correctly.

The extra array is used by the compiler to keep track of the addresses of the rest of the arrays as they are created (because their sizes are not always known at compile time) and consists of a 4 byte header plus two bytes for each array used in the program.

8.2 Memory map.

The areas of RAM used by compiled programs are

Address	Use
$0000 to $0800	- as for interpreter (see note below);
$0800 to $9FFF	- holds the compiled program, variable list array list, and strings organised as for interpreter;
$A000 to $BFFF	- holds the run-time library (loaded from file RTL-64);
$C000 to $CFFF	- unused;
$D000 to $FFFF	- used by Garbage Collection (see 8.3);

Note that the only byte in the area $0000 - $0800 used by a compiled program for a purpose different from the interpreter is $02FF which holds the flag to indicate that the file RTL-64 is loaded. A value of $64 indicates that the file is loaded; any other value means that it will be loaded automatically when the next compiled program is run.

Note also that the run-time library has to be located at $A000 to $BFFF so that the only machine code routines that cannot work with a compiled program is one that uses this area of RAM.

8.3 Garbage collection.

Garbage collection is the process of reorganising the string storage to recover unused space. The GC routine in ROM can be very slow.

The run-time library contains its own GC routine that is very fast. This routine works by copying all the strings out of the string area to the normally unused ROM area at $D000 to $FFFF, and then copying the strings back in a collected form.

Normally machine code will be located in the area $C000 to $CFFF (which is not used by the compiler). If a program requires more space than this for machine code then all that is necessary is to adjust the size of the area used by GC by adjusting its pointers. These pointers are :

$A040,41 - address of start of GC area

$A042,43 - address of end (top) of GC area

If GC finds that there is not enough space for all the strings then it will make several passes collecting a portion of the strings each time. In such a case the time for GC will increase a little but will still be many times faster than the GC routine in ROM. Note that the area defined by the two pointers above must be at least 512 bytes in size otherwise the GC routine in ROM will be used. This last point means that if an add on product requires all the RAM from $C000 to $FFFF then a compiled program will still work correctly provided that it sets the size of the GC area (via the two pointers described above) to less than 512 bytes.

Note that a machine code routine entered by a SYS call cannot directly access the two pointers, as, on entry to the routine the Basic interpreter will be mapped into $A000 to $BFFF instead of the run-time library. The routine will have to adjust the 6510 memory management registers itself, or alternatively the pointers can be set from Basic (Basic PEEK and POKEs access the run-time library rather than the interpreter).

8.4 Types of extension handled by the compiler.

There are three ways in which extensions are added to Basic and ALL will work with DTL-BASIC 64. The three techniques are

- additional statement type starting with a non-alphanumeric character;

- additional statement types starting with an unused token (ie. with a new keyword);

- SYS calls with parameters; ie. additional parameters following the address that are processed by the machine code routine;

The only restriction on the use of extensions is that they should not include a colon character (":") other than at the end of the statement. Also, if an extension based on additional keywords is used, then listings produced by the compiler will not print the new keywords correctly.

9. Errors.

The compiler performs exhaustive checks while compiling a program and reports all errors found. Errors can be found during both Pass 1 and Pass 2. In addition, further checks are made while the compiled program is run to detect errors that cannot be found at compile time.

If any compile time errors occur then the object file is deleted by the compiler to ensure that the errors are corrected before the compiled program is run.

There are three types of errors that can occur

Pass 1 errors;

Pass 2 errors;

Run-time errors.

In addition warning messages can occur during Pass 1

9.1 Pass 1 errors.

Pass 1 detects most errors because it checks the syntax of each statement. When an error is detected an error message is output following the line at which the error was detected. The message contains an error number and also indicates the position in the line at which the error was detected.

Note that the error may be before the point indicated. This is because an error cannot always be detected immediately, eg. in an expression, a missing bracket will normally not be apparent until the end of the expression.

Appendix B contains a full list of the error numbers and their meanings.

9.2 Pass 2 errors.

The main errors that can be found during Pass 2 are undefined line numbers; ie. a GOTO or GOSUB to a line number that does not exist.

The error message is simply the line number containing the error followed by a "U" to indicate an undefined line number is referenced from that line, eg.

```
23510  U
```

In addition, at the end of pass 2 an error 41 can occur

if it is found that an array is used in a program containing a VN or RO directive for which no DIM statement has been compiled (see section 7.2).

9.3 Run-time errors.

When a compiled program runs, the run-time library continually checks for errors and the following errors can occur

- NEXT WITHOUT FOR
- RETURN WITHOUT GOSUB
- OUT OF DATA
- ILLEGAL QUANTITY
- OVERFLOW
- OUT OF MEMORY
- BAD SUBSCRIPT
- REDIM'D ARRAY
- DIVISION BY ZERO
- STRING TOO LONG
- FILE DATA

The above error messages are the same as those used by the interpreter. The interpreter detects additional errors not in the above list (eg. syntax error) but the compiler will find these errors at compile time.

The meaning of the above errors are exactly the same as for the interpreter errors. Therefore, refer to the Commodore manual if the meaning is unclear.

The one difference between the run-time errors from compiled programs, and from interpreted programs, is that the compiled program gives the address of the statement containing the error rather than it's line number. A special program called ERROR LOCATE is provided to enable the line number to be found.

The procedure is

-make a note of the address of the error;

-load and run ERROR LOCATE;

-when requested, key in the program name (ie. the name of the object file) and later the address of the error.

ERROR LOCATE will display the line number of the statement containing the error.

Note that the above procedure will only work if the LN file for that program exists on the disk.

9.4 Warnings.

Warning messages occur when the compiler has detected an extension to Basic (see section 3.5) to notify the user that an extension has been found. The reason for doing this is, that if a syntax error occurs at the start of a statement, the compiler will treat it as an extension to Basic rather than an error (there is no way that the compiler could separate the two cases). Therefore if warnings occur for lines on which the programmer did not use an extension then an error must exist.

Warning messages can be directed to either the screen or the printer along with any error messages, and a count of the warning messages is output at the end of the compilation.

If a program frequently uses extensions to Basic then many warnings will occur and in such a case the programmer may not require them. Warning messages can be turned off by the use of the No Warning directive at the start of the program. In this cases no warning messages will be produced but a count will still be generated (see section 6.7).

Appendix A What is a compiler ?

This Appendix tries to outline the main differences between a compiler and an interpreter.

The first point to realise is that a compiler and interpreter are trying to achieve the same end, ie. they are both trying to provide a way of running a program. They both have to perform a similar set of tasks it is just that these tasks are performed at different times.

Consider what has to be done to 'run' a program. A program consists of a set of statements and each statement is simply a sequence of text characters. The program is intended by the programmer to define an algorithm, ie. it defines how a problem is to be solved or how a particular task is to be performed. The algorithm is defined in terms that are meaningful to the programmer but not very meaningful to the computer,ie. in terms of variables, operators, functions and line numbers etc.

The main tasks that have to be performed on each statement before a program can be run are

1. the type of the statement must be recognised;

2. the syntax of the statement must be checked;

3. for each variable name detected then the list of variables must be searched to see if the variable has been allocated an address, if not an address must be allocated;

4. for each reference to a line number (in a GOTO or a GOSUB) the address of the line must be determined;

5. for each expression the operator priority rules have to be applied and any brackets taken in to account in order to determine the order of evaluating the expression;

6. any non executable parts of the program such as spaces or comments (REM statements in Basic) must be skipped and ignored;

7. finally the statement has to be obeyed.

Both compilers and interpreters have to perform all the above tasks (and others); the difference is when the tasks are performed. This is important because most statements in a program are executed more than once and often many times. An intepreter performs the above tasks every time that a statement is executed and this means that the same work can be repeated many times. Such repetition is obviously wasteful and can be very time consuming,eg. a large program can have several hundred variables so that each time a variable is referenced a long search may be required. A compiler avoids such wasteful repetition by processing a program and converting it to a different form. In this way each of tasks 1 to 6 above are performed once only for each statement and only task 7 must be performed many times. Tasks 1 to 6 are performed when the program is compiled and only task 7 need be performed every time the program is run.

With an interpreter a program exists in only one form,ie. the text that the programmer has written. With an compiler the program has two forms

-the text form;

-the converted form;

To distinguish between the two the text form is normally called the source code and the converted form the object (or binary) code. The object code for a statement normally contains addresses where the source code has variable names and/or line numbers. Similarly expressions are normally re-ordered to cater for operator priority and brackets etc. Also all redundant information such as spaces,REMS and line numbers etc. is omitted and complex statements are normally broken down to a number of simple steps.

It should be clear from this that by pre-processing (ie. compiling) a program a compiler can make the program run much faster but obviously the compilation process takes time. The advantage of an interpreter is that when a program is being frequently changed (eg. when it is being debugged or modified) the source can simply be edited and the program re-run. With a compiler the program must first be re-compiled before a change can be tested. The two techniques are thus complementary; interpreters are best during the program development phase, but once a program is working, a compiler is superior because it gives the best program performance.

Appendix B Error Numbers

ERROR NUMBER	CAUSE OF ERROR
1	syntax error
2	wrong type of operand
3	no 'TO' where one expected
4	illegal array subscript
5	no ')' where one expected
6	no '(' where one expected
7	no ',' where one expected
8	no ';' where one expected
9	no 'THEN' or 'GOTO' where one expected
10	no 'GOTO' or 'GOSUB' where one expected
11	no 'FN' where one expected
12	constant too big (either > 255 or < 0)
13	expression too complex (shouldn't occur if program is OK on Interpreter)
14	syntax error in expression
15	too many ')'s
16	illegal operator in string expression
17	type mismatch
18	illegal statement type (CONT or LIST)
19	program too big (shouldn't occur for disk based versions if program is OK on Interpreter)
20	a function name must be real
22	FOR variable cannot be an array element
23	wrong number of subscripts
24	integer too big
25	negative number illegal
26	cannot set ST,TI,DS or DS$
27	function variable must be real
28	no function where one expected
29	no operator or separator where one expected
30	type mismatch in relational expression
31	no line number where one expected
32	no operand where one expected
33	illegal CS or CE statement
34	bracket missing from CS or CE statement
35	too many conversion variables (> 128)
36	error in CS or CE;no ',' or '=>' after name
37	error in CS or CE;no '%' where one expected
38	converted name clash in CS or CE
40	no '=' where one expected
41	default array found in overlay
42	too much DATA text (maximun amount of DATA text is approximately 8500 bytes for the single drive compiler and 6500 bytes for the tape compiler - there is no limit for the dual drive compiler)

DTL BASIC USER REGISTRATION CARD

NAME:

COMPANY:

TITLE: DEPARTMENT:

ADDRESS:

TELEPHONE:

COMPUTER TYPE: DISK y/n: TAPE y/n:

PRINTER TYPE:

OTHER PERIPHERALS:

DATE OF PURCHASE:

FROM:

HOW DID YOU LEARN ABOUT DTL-BASIC 64: DEALER y/n:
RETAILER y/n:
ADVERTISEMENT in:

WHAT PARTICULAR FEATURES OF DTL-BASIC MOST IMPRESSED YOU:

WHAT FEATURES WERE YOU DISAPPOINTED WITH:

SIGNATURE:

Please fill in the above and register with us as a user of DTL BASIC 64 JETPACK so that we may inform you when improvements are made to the product.

Return this form to: DATAVIEW WORDCRAFT LTD,
RADIX HOUSE,
EAST STREET,
COLCHESTER,
CO1 2XB.
ENGLAND.

DTL-BASIC 64 Application Note

RS232 problem

There is a problem that can occur in programs compiled by DTL-BASIC 64 that use the RS232 port. This application note describes the problem, explains the cause and how the problem may be avoided.

The problem is that during RS232 operations the characters recieved or transmitted may become garbled. This effect can apparently occur randomly after a period of correct operation.

The problem occurs if a garbage collection operation occurs whilst there are characters in one of the RS232 buffers.

Garbage collection is an operation performed periodically by the operating system to tidy up the area of memory used to hold strings. The frequency of garbage collection depends upon the the amount of free memory (used to hold temporary strings) and the amount of string processing performed by the program. For some programs garbage collection may occur very frequently which means that the use of the RS232 will almost always go wrong unless special steps are taken to aviod the problem.

The reason why garbage collection (GC) causes problems is that compiled programs use a special GC routine that is very fast (see section 8.3 in the manual). However this routine uses RAM that is 'behind' the kernel ROM. Whilst this RAM is switched in the ROM is not accessible. The kernel ROM holds the timer interrupt routine so the GC routine has to turn interrupts off whilst it is acessing the RAM. This means that occasional timer interrupts may be delayed.

A delayed timer interrupt is normally not a problem unless RS232 operations are in progress as the RS232 driver routines use timer interrupts to control the baud rate. For this reason GC can cause the baud rate to be inaccurate which causes characters to be garbled !

The simplest way of avoiding the problem is to turn off the fast GC so that the standard CBM GC routine will be used. The way of doing this is described in section 8.3 of the manual. The problem with this is that for some programs the CBM GC routine can be very slow.

Another technique is to force a GC before using the RS232 port by issuing a FRE. This will work fine as long as there is not another GC before the RS232 buffer has emptied. Whether this will happen depends upon the amount of free memory, the amount of string processing and the baud rate.

The amount of string processing may be reduced by minimising the amount of string arithmetic that can occur whilst RS232 transfers are in progress; eg. instead of a PRINT statement that includes a complex string expression (which will generate a number of temporary strings) it is better to calculate the string earlier and set the result in a variable,do a FRE and then PRINT the variable.

The baud rate affects the problem because if a slow one is being used then it may take several seconds to empty the RS232 buffer. If this is the case then after an RS232 operation it may be best to insert an appropriate delay to give the buffer time to empty.