SpeedWriter

for the Commodore 64

SpeedWriter™

HIGH-SPEED COMPILER SYSTEM
FOR THE COMMODORE 64 COMPUTER

BY
DAVID HUGHES

CodeWriter™ Corporation

TABLE OF CONTENTS

CodeWriter Corporation
7847 North CaldwellAvenue
Niles, IL 60648

(312) 470-0700
Telex 756942

© CodeWriter Corporation, 1984

WORLD WIDE PUBLISHERS

Dataview Limited
Radix House
East Street, Colchester
Essex, C01 2XB, England
Telex 987562 COCHAC

1. [illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

1. Introduction

SPEEDWRITER is a BASIC compiler for the CBM 64. The programs originate in the United Kingdom and are improved and upgraded versions of DTL BASIC COMPILER, developed for Commodore business computers.

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.

SPEEDWRITER has been specially optimized for the CBM 64 and it not only makes every BASIC program a lot faster but will also make each program significantly smaller, except for programs with only a few lines.

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

SPEEDWRITER is designed so that it can be used by people with no programming knowledge to compile existing programs. Yet for more experienced users, a range of facilities is provided to enable the full potential of the CBM 64 to be realized.

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

1.2 Purpose of this manual.

This manual describes how to use and operate all the versions of SPEEDWRITER

No attempt is made to teach BASIC programming or to define the BASIC language. This is not necessary due to the high level of compatability with the BASIC interpreter in the CBM 64.

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 which cover installation, main features and operation.

Important Note 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 has 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.

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

Chaining If two or more programs are chained together then chapter 7 should be read before compiling; similarly, if extentions to BASIC or machine code are involved then refer to chapter 8.

Chapter 5 explains how to get the greatest benefit from SPEEDWRITER

SPEED Improvements It is important to realize that, whatever SPEED improvement is achieved by compiling a program without any alteration, it is almost certain that 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. Chapter 6 describes directives that are available. A compiler directive is an instruction to the compiler stored within the source program.

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

Additional Notes

The following suggestions pertain to SpeedWriter version 2.0 and we strongly recommend that you read and follow them to assure proper working of SpeedWriter.

1) Problems can occur when using the VN directive if these steps are not taken. If a program is compiled that has a VN directive, and that program referes to variables or arrays that have not been used by any other program that shares the same variable list, then a new variable list will be created which has the wrong start address for the root program.

The solution is to ensure that after compiling any program that has a VN directive and which has been editied to use a new variable or array (or which has not been compiled before), then re-compile the root program after first editing it to refer to a new variable; eg simply add a statement x1=0 (where x1 has not previously been used). The new variable will force the compiler to create a new variable list file with the correct address.

2) Problems may be experienced when using the RS232 interface if Garbage Collect (GC) occurs while there are characters in the RE232 buffers. The problem occurs because the fast GC routine used by the compiled program has to keep turning interrupts on and off. The RE232 driver routines rely upon time interrupts to control the baud rate. If interrupts occur while there are characters in the buffer, then the baud rate is likely to be affected. The solution is to force the compiled program to use the normal GC routine section 8.3 of the manual. The following code will force the slow GC routine to be used:

```
poke41026,peek(41024):poke41027,peek(41025)
```

3) SYS statements are not 100% compatible with the interpreter. The difference is that when the interpreter calls a machine code routine via a SYS it first sets the registers A, X, and Y to the values they held after the previous SYS call (or to zero if no previous SYS call). In compiled programs this does not occur and the contents of these registers is undefined.

This incompatibility does not matter in the vast majority of cases; the only known situation in which it does matter is for a SYS 65520. This routine sets the cursor position from the values in the X,Y registers which under the interpreter will normally be zero (ie. SYS 65520 performs a HOME). In a compiled program a SYS 65520 will set X and Y to non zero values (ie. the cursor will be set to the wrong position). The solution is to not use SYS 65520; instead POKE 216 with 0 (ie. set the x position) and POKE 211 with 0 (ie. set the Y position) and then use SYS 58732.

4) The ABS function has no effect for integer variables when the value is in certain ranges (eg. between -128 and -256). The solution is to use floating point numbers when ABS is invoked or simply multiply by -1 if the value is less than zero.

5) Unpredictable results can result from setting TI$ to the result of a string expression. There is no problem setting TI$ to a constant or to the value of a variable; eg. TI$ = LEFT(A$,6) will not always work, but XX$ = LEFT$(A$,6):TI$ =XX$ will always work.

6) Programs which alter the 6510 i/o port at location 1 and its data direction register at location 0 will experience problems if constant values are POKEd into these locations. The correct procedure is to PEEK the existing value, set or clear the relevant bit, and poke the result back. (eg. the clear CHAREN (to switch in character generator) use POKE1,PEEK(1) AND251.)

1.4 Major benefits of SPEEDWRITER

Fast	* Compiled programs can run to up 55 times faster in ideal situations - typical improvements are normally in the range of 5 to 30 times faster.
Compact	* Compiled programs are between 50% and 80% of the size of uncompiled programs 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.
Protection	* Compiled programs cannot be listed or altered.
Compatible	* The compiler is totally compatible with all the features of CBM 64 BASIC meaning that a BASIC program can be compiled without alteration.
Arithmetic	* The compiler provides true integer arithmetic as well as floating point arithmetic.
Machine Code	* 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.
Extensions	* The compiler can compile programs incorporating extensions, to BASIC; ie. additional BASIC statements implemented by machine code in ROM or RAM.
Games and Sprites	* 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).
Professional	* 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 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.
Garbage Collection	* SPEEDWRITER 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).
Migration	* A facility is provided to ease the transportation of programs from other machines to the CBM 64. This enables PEEK/POKE addresses to be automatically adjusted without changing situations, e.g. when a program POKEs data directly to the screen.

2. Installation of SPEEDWRITER

2.1 Contents

Your copy of SPEEDWRITER should consist of :

- a SPEEDWRITER compiler disk
- this manual

3. Main features of SPEEDWRITER

3.1 Requirements

SPEEDWRITER requires a standard Commodore 64 single disk drive.

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 compiler programs.

3.2 The compilation process

SPEEDWRITER is complementary to the CBM BASIC interpreter due to its total compatability. 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.

Error Checking

It is usual before compiling to make sure that the programs work on the interpreter. SPEEDWRITER does make thorough checks for errors both at compile time (i.e. while the program is being compiled) and at run-time (i.e. 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 :

Source File

- Copying the source file (i.e. the program to be compiled) on to the compiler disk or copying the compiler files onto the program disk;

- Loading and running the compiler;

- Telling the compiler the name of the source file; i.e. the file containing the program to be compiled;

- Telling the compiler the name of the object file; i.e. the file to be created by the compiler to hold the compiled program;

Two Pass Process - 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 recreated. Note that for SPEEDWRITER the work-file is held in memory in the area unused by BASIC.

Run Time 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.

Program Limitations Note that 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.

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

Special File 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 (e.g. 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 <u>must</u> be in memory when a compiled program is run.

Auto Load 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 disk i.e. from wherever the program was loaded. This means that the disk from which the first compiled program is loaded after power up should contain a copy of RTL-64.

8K Bytes 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, i.e. the 38K available to BASIC, the run-time library is stored outside this area in some of the RAM that would otherwise be unused.

Assembler For the benefit of machine code programmers, the run-time library is stored in the 8K from $A000 to $BFFF which is an area of RAM that cannot be directly accessed from BASIC. This leaves the 4K of accessible RAM at $C000 (i.e. 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 utilize machine code subroutines to perform tasks that are not possible or are difficult in BASIC. With the greatly improved performance provided by SPEEDWRITER it is possible to replace many machine code routines with BASIC code.

However, there will always be situations where some machine code is desireable. SPEEDWRITER 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 compile preserves precisely the same format for page zero, the variable list, the array list and string storage etc.

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

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 BASIC by means of machine code routines either in ROM or RAM.

SPEEDWRITER 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 as the compiler checks the syntax of a statement and if it cannot recognize the first character of the statement (i.e. if the character does not start with either a legal keyword or an alphabetic character), then it assumes that the statement is valid and is an extension to standard BASIC.

How it Works

The compiler embeds the text of the extension statement in the compiled program exactly as it occurs in the source program. Then it precedes it by a special code and follows it by a SYS call to the run-time library at the time the program is executed. Next, the run-time library detects the special code, sets up the page zero pointers to the extension statement and calls the interpreter to process it.

Now the interpreter processes the statement just 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 works because the machine code finds the variable and arrays lists, etc. exactly as it expects.

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

4. Operation of SPEEDWRITER

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 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 amont of free space by displaying the disk directory, ie. type the commands

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

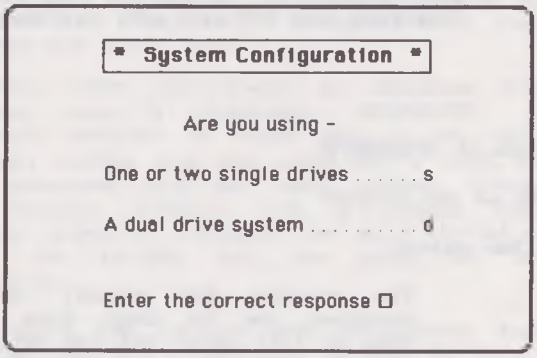
and when READY is displayed, type

```
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).

```
TYPE: LOAD"*",8,1
```



```
* System Configuration *

       Are you using -

One or two single drives ......s

A dual drive system ...........d


Enter the correct response □
```

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 into memory.

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. Once you have used the compiler and the machine has not been turned off, 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 run again.

4.2 Compilation options

The compiler will display the following list of options

```
source file ? :
object file ? :
print source ? :
print stats ? :
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).

Starting up

Type the exact name of the source file

Type the name of the object file (the file to be compiled).

Unless you wish to print, the compilation may be started by pressing F3.

Print Options

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

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.

Statistics

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 (these are always displayed).

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.

4.3 Function keys

As mentioned earlier, F1 moves you to the next option. If there are no more options and it is pressed, compilation will start.

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

Directory

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

Restart

If it is realized that an option has been input incorrectly then F2 can be used to go back to the beginning.

Change Diskettes

If it is discovered that the wrong disk is being used then F4 can be used to allow the disk to be changed without reloading the compiler. Note that the new disk should contain at least the file DTL-BASIC-MC, as this will be re-loaded from disk.

Exit

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

4.4 Compilation

4.4.1 All versions

As the compilation begins, 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.

Printer Note — 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.

Error Messages — 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 and the screen becomes full of error messages, the compilation will pause so that the lines in error may be noted before compilation resumes.

Warning Messages — 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 the compilation, a count of the number of error and warning messages are displayed and the compilation statistics are displayed or printed.

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 reported in bytes, blocks and the number of bytes in the last block (a block is 256 bytes), e.g.:

SOURCE PROGRAM SIZE - 4253 (16,157)

So, 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.

Finding Variables

The object file size exceeds the object program size because this 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 that 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 edited to correct the errors before it can be compiled.

If there were no errors the user has three options:

- Press key "C" to compile another program;

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

- Press any other key to exit from the compiler;

4.7 Operation of compiled programs

Operation of compiled programs is identical to that for uncompiled programs, i.e. 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 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 no delay will occur because the program will detect that RTL-64 is already in memory.

Note CONT cannot be used with compiled programs. SYS 2061 should be used.

When a compiled program is stopped, variables and array elements can be displayed (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, just the same as for uncompiled programs, e.g.

```
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.

4.9 Special Operation Features

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

Automatic object file name

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, e.g.

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 (i.e. when one program LOADS another program). Otherwise, the LOAD statement within the program would have to be altered.

Compilation of several programs

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 <u>control file</u> is a normal file that has the last four characters equal to "-con", e.g. "compile-con".

Control file

A control file is created and edited 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 <u>first</u> file name should be the name of the first source file to be compiled and the <u>second</u> 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 then 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 complications will occur, i.e.

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

Error detection — To start the compilation, the name of the control file should be given instead of the source file name. The printing options selected will apply to all complications. 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 Trouble shooting

Special Integer Mode — 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.

Premature Stop — If the compiler stops during compilation and the 1515 printer is in use, refer to section 4.4.1.

VL or RO file — If a compiled program using either the VL or RO directive at the start of the program crashes when run, check that the VL file is present on the disk (or tape). 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 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.

DOS errors — 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 is to take place.

The 1541 can also damage files on occasion 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 damaged then perform a VALIDATE and copy the file from a backup.

Avoid the replace option

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

Patches

Some BASIC programs are "patched" in a special way by the programmer so that after loading they will run automatically, i.e. 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.

Loading

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 statement 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=1to6:READ X:POKE630 + I,X : NEXT
1040 NEW
```

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

Pokes

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 SPEEDWRITER

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 considerable additional improvements can be achieved.

Integer Arithmetic

The reason for this is that SPEEDWRITER supports integer arithmetic as well as floating point arithmetic. Integer operations are used for all operations when both operands are integers. 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 realize 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 often as possible.

CS and CE Directives

In order to save a user the trouble of having to work through and edit an existing program to change real variables to integers, SPEEDWRITER 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 should be floating point; i.e. any variables which may hold a value greater than 32767 or less than -32767, 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 of improvement can, in some cases, be dramatic.

Speed Improvements

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

1.) When a program is performing I/O (input/output) the program can spend most of its time waiting for the peripheral, e.g. disk drive 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.

2.) The performance of a program on the interpreter depends 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.

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

5.2 High speed SPRITE movement

Games and Graphics

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. 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. SPEEDWRITER has a feature to overcome this problem called <u>special poke</u> mode which is controlled by the SP and NP directives (described in section 6.6)

Special Poke Mode

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 statement 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.

Stop Key

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 SPEEDWRITER which is not immediately obvious is that it is possible to write programs that are easier to understand and to modify.

Better Programming Techniques

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. For example:

- 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 an interpreted program somewhat, but they can become almost imcomprehensible to follow.

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 from the beginning and makes subsequent modifications much easier.

5.4 Utilizing the extra memory

When a program is compiled the reduction in size of the program can be considerable. This means that it is often worthwhile to <u>increase</u> the size of the arrays to utilize the extra space or to keep more information in memory in order to reduce the amount of disk I/O required.

Out of Memory error

However, it is always convenient to be able to run the same program on the interpreter when debugging. If arrays are larger or if there are more arrays, then an "out of memory error" is possible. A simple way around this is to make the program detect whether it is compiled or not compiled 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), e.g.

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, e.g.

```
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 take 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 minimize 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.

Use at the Beginning

Most directives can only occur at the start of the program (i.e. 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.

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 be 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, e.g.

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

means convert all references to the names A1,ZZ,X2,X3 to integer, i.e. 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 integers except I1, I2 and I3.

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

means convert all floating point variables with no exceptions.

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

Error Messages

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, e.g.

```
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

When changing names 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 name variables cannot exceed 128.

Integer "For" Variables

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 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 division 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. As has already been exlained, such operations are 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 and integer division is used the answer will be 4, but if floating point division 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.

Note However, occasionally this can cause the compiled program to work differently than the uncompiled program. In such cases the use of special integer mode will overcome the problem, i.e. it will force the compiler to always use floating point arithmetic for division and exponentiation.

6.4 Variable list positioning

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.

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

VL 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 Commodore 64 a BASIC program starts at address 2049 ($0801 in hex). For example, if the directive

REM VL 15000

is used, then the variable list will be placed at absolute addresses 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, 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 be automatically loaded to the correct address. On subsequent runs of the program the file will not be loaded since the program will detect that it already exists in memory.

Pokes Some programs utilize POKES to location 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 and 46 from those set by the compiled program then <u>problems are likely to occur</u>.

Chaining RO 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 Note that a <u>problem</u> can occur when copying a VL file to another disk. When the VL file is LOADED to memory prior to a SAVE, the system can damage the file. This occurs because it thinks the VL file is a program. The problem will not occur if the first used variable in the program has a single character name.

6.5 Disabling the stop key

The directive

REM ** DS

<u>disables</u> the Stop key, while

REM ** ES

enables the Stop key.

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

Faster — Programs run slightly faster with the stop key disabled.

In 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 will probably not be able to be restarted with SYS 2061 (the compiled version 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, e.g.

- 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 64.

Special poke mode is enabled by the directive

REM ** SP

and disabled by

REM ** NP

Enabling — Before enabling the mode it is necessary to define the adjustments 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).

Example

As another example, suppose a program written on another machine with the screen at $8000 hex was to be run on the 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, while a zero leaves the corresponding bit the same.

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

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

at the start of the program <u>after</u> a POKE is required to select the color desired.

6.7 Inhibiting warning messages

When a program uses extensions to BASIC (see section 3.5), 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 that utilize chaining are to be compiled then the programs can either be compiled to share variables or not to share variables.

Sharing Variables

Sharing variables occurs when a program is written to access variables set up by a previous program, i.e. 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.

Poke Values

One common practice when chaining is for the first program in the chain to POKE values into locations 45 and 46 which hold a pointer at 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 shared variables

In this case no special action is necessary save 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 shared variables

RO and VN

If variables are to be shared then the use of the directives RO and VN are necessary. This is 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), i.e. it defines the size of the largest program in the chain and thus the position for the variable list.

Note: 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.

VL file: The RO directive tells the compiler that it is compiling the root program of a chain and that at the end of the compilation, a VN file will be created which 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.

Restrictions: Note that there is one restriction to 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, i.e. arrays without DIM statements will not be automatically dimensioned to have 11 elements. The compiler will not 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.

Summary

To summarize, 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 utilize machine code. The machine code may be 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 a 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. SPEED-WRITER creates lists in exactly the same format and uses 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 things to watch out for.

Variable Order 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.

Array Order 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 2 bytes for each array used in the program.

8.2 Memory Map

The area of RAM used by compiled programs is:

Addresses	Use
$0000 to $0800	Same as interpreter
$0800 to $9FFF	Holds the compiled program, variable list, array list and strings organized as for interpreter
$A000 to $BFFF	Holds the run-time library (loaded from file RTL-64)
$C000 to $FFFF	Used by garbage collection (see note next page)

8.3 Garbage Collection

Garbage collection is the process of reorganizing 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 $C000 to $FFFF, and then copying the string back in a collected form.

This means that if a program uses machine code located in the GC area there is a possibility of it being overwritten. Whether this happens depends upon the maximum amount of space that may be required by the program to hold strings and where the machine code routines are located.

12K byte string limit

The GC routine works from the top of memory down. Therefore, if the routines are put in the block $C000 to $CFFF, as long as the size of all the strings do not exceed 12K bytes there will be no problems.

If you are not sure whether some machine code may be overwritten by GC, the machine code can be protected by adjusting the pointers used by the GC routine. 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.)

Note — Note that a machine code routine entered by a SYS call cannot directly access the two pointers above, 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 themselves or alternatively, the pointers can be set from BASIC (BASIC PEEKS 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 SPEEDWRITER . The three techniques are:

1.) Additional statement type starting with a non-alphanumeric character.

2.) Additional statement types starting with an unused token (i.e. with a new keyword).

3.) SYS calls with parameters i.e. additional parameters following the address that are processed by the machine code routine.

Restrictions — The only restriction on the use of extentions 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.

Note — 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.

Syntax Checks — 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 — Note that the error may be before the point indicated. This is because an error cannot always be detected immediately, e.g. 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

Undefined Line Numbers — The main errors that can be found during Pass 2 are undefined line numbers; i.e. 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 that an undefined line number is referenced from that line, e.g.

```
23510 U
```

Error 41 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).

.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 (e.g. syntax errors) 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.

Run-Time Error Difference The difference between 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 its line number. A special program called ERROR LOCATE is provided to enable the line number to be found.

Error Locate The procedure is

- Make a note of the address of the error

- Load and run ERROR LOCATE

- When requested, key in the program name (i.e. 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 Note that the above procedure will only work if the LN file for that program exists on the disk.

9.4 Warnings

Extension to BASIC 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. A count of the warning messages is output at the end of the compilation.

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

What is a Compiler?

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

Interpreter vs. Compiler - Results are what Count

The first point to realize is that a compiler and interpreter are trying to achieve the same end i.e., they are both trying to provide a consistent and logical format for implementing a program. They both have to perform a similar set of tasks. It is just that these tasks are performed at different times.

The Components of Running a Program

Consider what has to be done to "run" a program. A program consists of a set of statements or instructions. Each statement is simply a sequence of text characters. The program is intended by the programmer to define an algorithm, i.e. 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, i.e. in terms of variables, operators, functions, 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 recognized.

2.) The syntax of the statement must be checked.

3.) For each variable name detected 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 GOTO or GOSUB statements) the address of the line must be determined.

5.) For each expression the operator priority rules have to be applied (including taking into account any brackets) 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 ignored.

7.) Finally, the statement has to be obeyed.

Compare and Contrast

Both compilers and interpreters have to perform all the above tasks (and others). The difference is evidenced when the tasks are performed. This is significant because most statements in a program are executed more than once and often many times.

An interpreter performs the above tasks every time a statement is executed. This means that the same work may be repeated many times. Such repetition is obviously wasteful and can be very time consuming, e.g. a large program can have several hundred variables requiring long searches every time a variable is referenced.

A compiler avoids such wasteful repetition by processing a program and converting it to a different form.

In this way each of tasks 1 through 6 are performed only once for each statement and only task 7 must be performed repeatedly. Tasks 1 to 6 are performed when the program is compiled and only task 7 need be performed every time the program is run.

The Compiler has two Forms - Source and Object

With an interpreter a program exists only in one form, i.e. the text that the programmer has written. A compiler has two distint forms:

1.) The text form

2.) 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 to operator priority and brackets, etc. Also all redundant information such as spaces, REMS, line numbers, etc. are omitted. Moreover, complex statements are normally broken down into a number of simple steps.

Summary

It should be clear from the preceding that by pre-processing (i.e. 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 (e.g. when it is being debugged or modified) the source can be simply edited and the program re-run. With a compiler the program must first be re-compiled before a change can be tested. These 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 overall program performance.

A Note on SPEEDWRITER

You will notice that the SpeedWriter manual makes occasional references to **DTL-BASIC**. DTL-BASIC, as the original program is known (and as is listed on your media), is owned and copyrighted by Drive Technology Ltd. (David Hughes, designer) and published worldwide by Dataview, Ltd. Both firms are located in the United Kingdom.

SpeedWriter, is an enhanced version of DTL-BASIC and is available under this name in the United States. SpeedWriter is a trademark of CODEWRITER Corporation and is protected under U.S. trade laws.

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 > 225 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 (maximum amount of DATA text is approximately 8500 bytes for the disk versions

DTL-BASIC 64 Release 1 - known problems

1. The compiler (both disk and tape versions) will report a syntax error for PRINT statements that contain TAB or SPC functions when the closing bracket is followed by an alphanumeric character, eg. the statement

```
100 PRINT TAB(8)A$
```

would generate an error message.

The solution is to use a ";" after the TAB or SPC functions, eg.

```
100 PRINT TAB(8);A$
```

would compile without an error message.

This problem will be corrected in the next release of the compiler.

2. If the compiler stops during compilation with a "NO CHANNELS ERROR", the reason is actually a read or write error. This is due to a bug in the 1541 DOS which means that when an error occurs then if further characters are read or written after the error (ie. 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 slows down the disk i/o by a factor of three or four.

If the "NO CHANNELS ERROR" occurs 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 a disk 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 by a 1541 they do appear to be more susceptible to errors than ones written on a 1541.

3. Problems will occur when trying to OPEN a channel to the RS232 port. After such an OPEN the operating system performs a CLR which confuses the compiled program and causes it to restart at the first line of the program with some of the page zero pointers damaged.

This problem will be cured in the next release of the compiler. In release 1 the problem can be overcome by the addition of two lines at the front of the program, eg. the program

```
10  AP = 128
20 OPEN AP,2,3,CHR$(7)+CHR$(0)
30 <rest of program>
```

will not run correctly when compiled. However. When the following two lines are added then it will run;

```
1 IF AP=0 THEN FOR I=0 TO 5:POKE 736+I,PEEK(45+I):NEXT:GOTO10
2 IF AP=128 THEN FOR I=0TO5: POKE45+I,PEEK(736+I):NEXT:GOTO30
```