VERSION 2 RELEASE 4 FOR 4032/8032 PET

DTL BASIC

a basic compiler

by Drive Technology

Dataview

USE SYS 1069 instead of CONT

STOP key – Section 4.6

Copyright

License

DTL-BASIC is licensed to the registered user on the clear and explicit understanding that compiled programs, whether or not owned by the registered user, shall not, by way of trade or otherwise, be lent, re-sold, hired out or otherwise circulated. The registered user of DTL-BASIC is licensed to compile programs for the sole use of the user.

Where a user wishes to circulate copies of compiled programs, then details of the necessary license may be obtained from the Distributors.

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

Scope

DTL-BASIC should only be used to compile programs that are the purchaser's property. Programs that are licensed for use by a third party may have their copyright violated if compiled. The purchaser is advised to check with the program supplier before compiling any programs that are not owned by the purchaser.

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.

Reg. User .....

Serial No. ...1594.....

Sole Distributor
Dataview Limited
Portreeves House
East Bay, Colchester
Essex, C01 2XB, England
Tel: Colchester (0206) 865835
Telex: 987562

DTL-BASIC Release 5

IMPORTANT NOTES

1. DTL-BASIC version 4 (for the CBM 8096) works in conjunction with SYSTEM 96 and requires release 1.1 of SYSTEM 96 or a later release. DTL-BASIC will NOT work with release 1 of SYSTEM 96.

2. Release 5 is at present only available for version 4 (ie. not for versions 1,2 &3). The remaining versions will be upgraded at a future date.

Changes from Release 4.

1. Release 5 introduces version 4 of DTL-BASIC for the CBM 8096 running under SYSTEM 96. This allows Basic programs requiring up to 78K of memory to be compiled. Note that the existing version 3 of the compiler will also run on the 8096 for programs that do not require more than 32K for Basic (see section 1).

2. New facilities are provided to give a powerful overlay system - see section 7.

3. A control file facility is provided to enable a number of programs to be compiled with a single run of the compiler (see section 4.3).

4. Release 5 is even more compatible with the CBM interpreter as chained programs may now share variables (see section 7). In addition CONT may now be used with compiled programs (version 4 only).

5. The compiler can now compile files with upper-case characters in the file name.

6. Compiled programs will now work satisfactorily with assembler routines that move the array list at run time.

7. A number of minor faults have been fixed.

8. New directives are provided to disable and enable the Stop key without affecting the clock. Note that release 4 programs that POKE location 1072 to achieve the same effect will have to be altered when used with release 5 (see section 4.6).

we have release 4 version 2

Contents

1. Introduction.

DTL-BASIC is a Basic compiler for Commodore machines and is fully compatible with the Commodore Basic Interpreter (with a few very minor exceptions).

This manual describes how to use and operate DTL-BASIC. The manual does not try to define the Basic language or to teach Basic programming as these subjects should be adequately covered by the Commodore documentation.

Readers of this manual who are not familiar with the differences between interpreters and compilers should refer to Appendix A.

Four versions of DTL-BASIC are available for the following machines :

DTL-BASIC version	Machine	available at release
1	CBM 3032	4
2	CBM 4032	4
3	CBM 8032	4
4	CBM 8096	5

The release number indicates the level of Basic facilities supported; ie. each release is intended to be upwards compatible from the previous release and to have extra features to provide additional advantages to the user.

Throughout the manual, where references are made to features that are specific to a particuler version this will be indicated (eg. version 5 only), similarly where a feature is available only from a certain release this will also be indicated (eg. release 5 or later).

Please note :

1. A program compiled by Version 1 will only run on a 3032.

2. A program compiled by Version 2 will run on an 8032 as well as a 4032.

3. A program compiled by Version 3 will run on a 4032 as well as an 8032.

4. Version 4 runs on SYSTEM 96 on an 8096 and produces programs that also run on SYSTEM 96. SYSTEM 96 is a language system for the CBM 8096 that enables Basic programs of up to 78K (46K for program & 32K for data) to be run. Without SYSTEM 96 the 8096 can only support programs of up to 32K.

SYSTEM 96 can be obtained from your DTL-BASIC supplier.

5. Version 3 may also be used on an 8096 and does not need SYSTEM 96 but in this case Basic programs will be restricted to the normal 32K limit as on the 8032. Users of the 8096 who do not require Basic programs of more than 32K total size (ie. program plus data) and who wish to access the extended memory via assembler code (eg. via the Expanded Basic package supplied by Commodore) should use version 3 of DTL-BASIC.

The main advantages of DTL-BASIC are :

-compiled programs run much faster than on the interpreter. The speed improvement depends to a large extent upon the size and nature of the program and upon the time involved in fixed overheads that cannot be speeded up (eg. disk accesses and printing) but statements within very large programs can run up to 20 times faster. However, a typical overall improvement for large programs is probably 4 to 8 times faster and existing programs which have already been worked on to minimise the overheads of the interpreter will probably be 2 to 5 times faster.

- the compiler is almost totally compatible with the interpreter so that existing programs can simply be recompiled without alteration to obtain all the benefits of compilation (see section 10 for a list of the few incompatibilities). The very high degree of compatibility means that programs may be developed on the interpreter (as this is easiest for debugging) and when working can be compiled.

- the compiler implements true integer arithmetic as well as real arithmetic. The interpreter allows integers but converts them to reals for all operations and real operations are very slow. Therefore, the use of true integer arithmetic can lead to significant speed improvements. Special facilities are provided to automatically convert reals to integers when compiling existing programs.

- the compiler will accept extensions to Basic implemented by assembler routines in RAM or ROM (such as used with Computhink disks and such products as Command-O etc.) and in fact the compiled programs are compatible with the vast majority of assembler routines that are used with interpreted programs.

- compiled code requires approximately 50-80% as much store as un-compiled code. However, for versions 1,2 & 3 there is a fixed overhead of about 16 blocks so that small/medium sized programs are unlikely to achieve a store saving. For version 4 programs should always be significantly smaller when compiled.

- compiled programs cannot be listed or altered by a user.

- the use of the compiler can result in significantly reduced development and maintenance costs.

- from release 5 onwards the compiler provides powerful Overlay facilities that enable several separately compiled blocks of program code to be in memory at once. Each block may be overlayed from disk independantly of the other blocks and a program in one block may call subroutines in other blocks or jump into other blocks.

- compiled programs utilise stack space more efficiently than the interpreter thus enabling more complex programs to be run.

- the compiler is fast. Compilation speed is typically 1-2 statements/second. The compiler itself is mainly written in Basic and is used to compile itself. This demonstrates that the compiler can be used to compile very large, complex programs (the source of the compiler is a file of 109 blocks).

- compiled programs can disable the stop key without affecting the clock.

Section 6 describes the advantages of using DTL-BASIC in more detail to enable users to fully obtain the benefits possible with the compiler.

2. General Description of DTL-BASIC.

This section describes the main features and capabilities of DTL-BASIC.

The original aims in designing the compiler were :

- that any program that will run on the interpreter should be able to be compiled without modification;

- that compiled programs should run much faster than interpreted programs;

- that apart from running faster the compiled programs should operate in exactly the same way as un-compiled programs;

- that compiled programs should if possible be smaller than un-compiled programs.

The first requirement listed above was seen as the most important as there are many very large Basic programs and if the compiler applied arbitrary limits to the size of programs that could be compiled then many users would find that the programs that they most wanted to compile could not be compiled. Secondly it was felt that although a compiler has many advantages over an interpreter the great feature of an interpreter is the ease with which a program may be debugged. However fast a compiler is to compile a program it will almost always be faster to develop a program using an interpreter. Consequently the major aim was to achieve almost total compatibility with the CBM interpreter thus enabling the compiler and the interpreter to be used together so that program developers can take advantage of the best features of each system.

The second requirement was to make compiled programs as fast as possible. This requirement to some extent conflicts with the first in that if the compiler was made to produce the fastest compiled programs possible, then the programs would be very much larger than un-compiled progryms and often would be too large to fit in the machine.

DTL-BASIC has been designed to achieve the ideal balance between speed and size so that compiled programs will run much faster and will normally require less store than uncompiled programs. Note that as explained in section 1 then for some versions of the compiler small/medium sized programs may become somewhat larger when compiled.

It should be noted that it is impossible to accurately predict the speed and size improvements for one particular program because this depends very much upon the way the program is written. A program that has been written to be as fast as possible and as small as possible on the interpreter will not improve by the same factor as a program that has been written to be readable and easy to modify.

2.1 The User Interface.

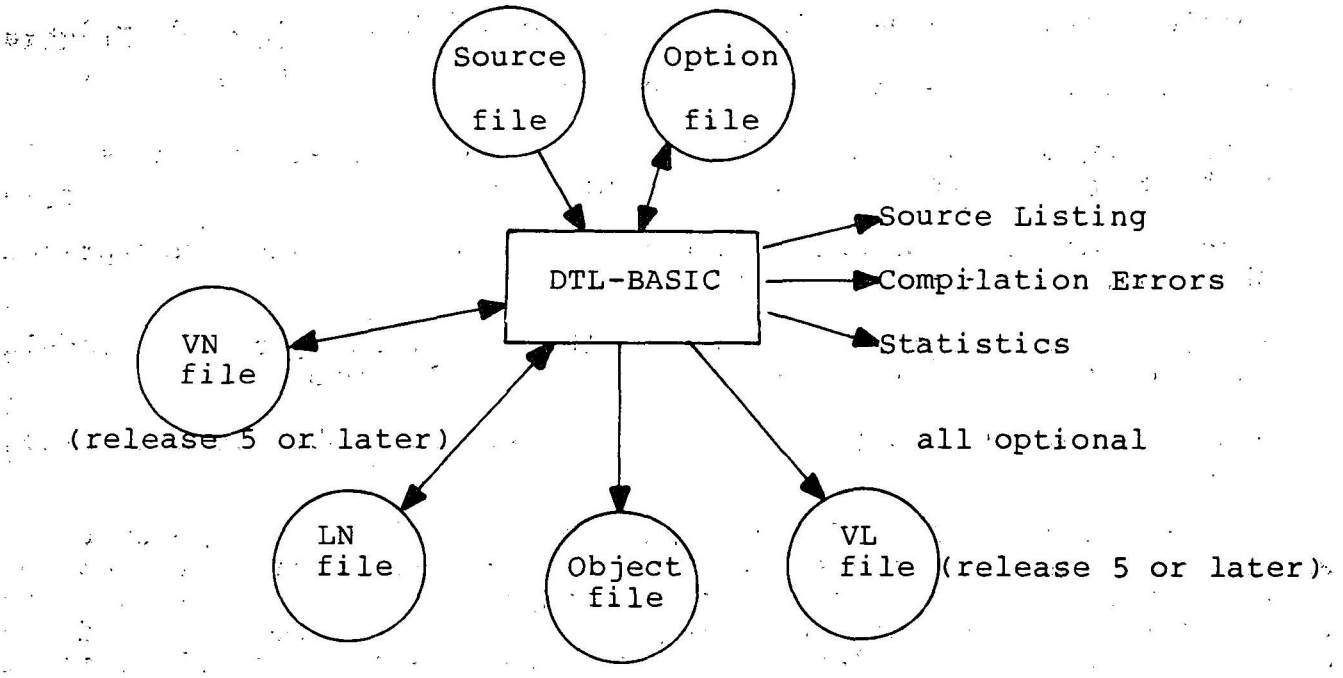

FIGURE 1

The above diagram shows the main points of interaction between the compiler and the user.

The Source file is the main input to the compiler and is a perfectly standard Basic program file,ie. the Source file contains a Basic program that can be LOADed and RUN. The file can also be edited and LISTed in the normal manner for Basic programs.

The Object file is the main output from the compiler and contains the compiled program. As far as the interpreter and DOS are concerned this file is also a program file and can also be LOADed and RUN. However, because the file contains the compiled version of the source file it cannot be edited or LISTed. Changes can only be made to an object file by editing the source file and re-compiling.

The LN (Line Number) file is essentially a workfile of the compiler and is used to relate line numbers to their addresses within the Object file (the Object file does not contain any line numbers). The LN file is not required to run the program and may be SCRATCHed. However, if the program is not fully debugged the LN file should be retained as it can be used to locate run time errors by the ERROR LOCATE program (see section 8.3).

The VL (Variable List) file is used to contain the variable list for use by the compiled program; the variable list is created by the compiler to save the compiled program from having to create the variable list at run-time. The VL file is not always created because in some situations the the variable list is included in the object file. The VL file is created when the variable list will not reside immediately following the compiled program at run time.

This occurs in two situations :

- whenever version 4 is used because SYSTEM 96 stores the variable list always at the same address and separate from the program;

- when the Overlay facilities are in use;

In such situations a compiled program will not run unless the VL file exists on one of the two disk drives.

The VN (Variable Name) file is created by the compiler when a program using the overlay facilities is compiled (available from release 5 onwards). The file is necessary because individual programs that are to work together as overlays normally share the same variable list. For this reason when an overlay is being compiled it is important that the compiler knows the identities and addresses of all the variables that are used by the overlays that have already been compiled. This is achieved by means of the VN file.

When the compiler is run it can be instructed to list the source file on the printer whilst it is being compiled.

The compiler performs exhaustive syntax checks and an error message is output for each error found. The compiler can output the error messages to either the printer or the screen. If errors are output to the screen the compilation is suspended when the screen is full of error messages so that the errors may be noted before the compilation is resumed.

The statistics that can be produced by the compiler relate to the Object file and give the size of each of the main sections within the file (see section 2.3). This information is provided to enable the programmer to see how the store is being used. It is important to use the statistics when comparing the size of compiled programs with uncompiled programs (rather than simply comparing the file sizes). This is because in some situations (as has already been explained) the Object file includes the variable list (ie. the list of all non-array variables). This means that the Object file may be several blocks larger than the actual program size. In such a situation the statistics given by the compiler include both the actual program size and the Object file size.

The Options file is used to record the options selected by the user when the program is run,ie.

- the Source file name;
- the Object file name;
- whether the source is to be listed;
- whether errors are to be listed;
- whether the statistics are required;
- the text message to be used to identify any listings.

When the compiler is run it looks for the Options file and if it exists then the data is displayed to the user who can change the data if necessary before starting the compilation. Often when a program is compiled a number of times the options will not need to be changed and the user will avoid a lot of repetitive typing.

2.2 The Internal Structure.

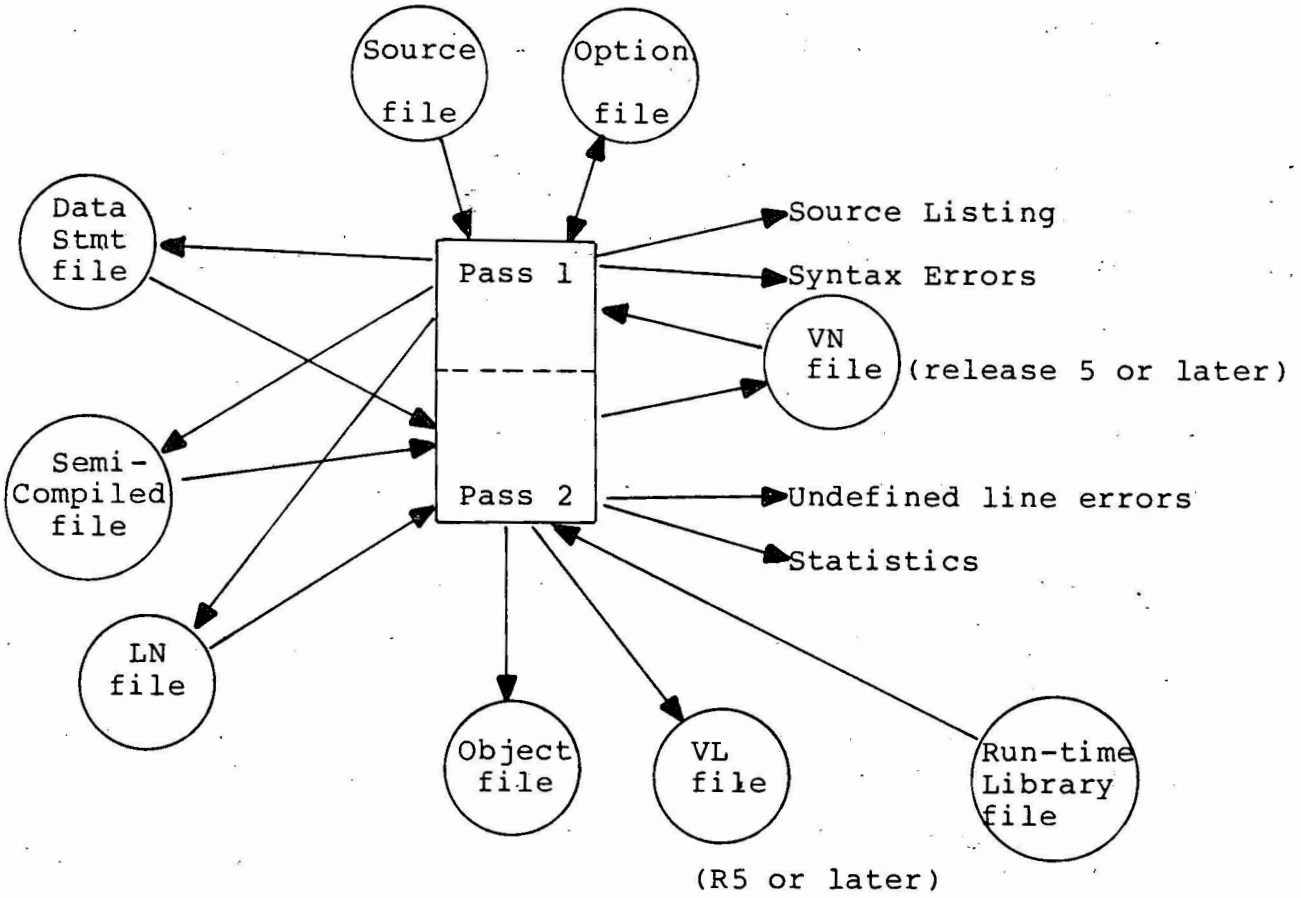

FIGURE 2

This section describes the internal working of the compiler in more detail. The figure above is the same as figure 1 with additional detail added to show all the files involved during compilation.

DTL-BASIC is a two pass compiler,i.e. it processes the program twice to produce the Object file.

The main stages involved in the first pass are :

- the Options file is read (if it exists) and the contents displayed to the user.

- the User changes (or inputs) the option data as necessary and starts the compilation.

- the compiler deletes any existing Options file and creates a new one.

- if a directive is found indicating an overlay then the VN file is read to memory (if one exists).

- the specified source file is opened and is processed statement by statement. Whenever a line number is found an entry is made in the LN file. If a statement is a DATA statement it's contents are output to the Data Statement file. For other statements the syntax is checked and code is output to the Semi-compiled file. This code is not complete because any line numbers will have not been converted to an adddress.

- as each line is processed then if source listing was selected then the line is output to the printer.

- if a syntax error is detected an error message is output to either the screen or the printer.

At the end of pass 1 the compiler has built up internally a list of all variables and knows the amount of memory required for compiled code and data statements. It can therefore calculate the addresses of all variables and it also has (in the LN file) the address of each line.

The operations involved in pass 2 are :

- the Object code file is created and if necessary the contents of the Run-time library file copied to the Object file.

- the contents of the Data Statement file are copied to the Object file.

- the Semi-compiled file and the LN file are processed together and the code for each statement is first completed by changing line numbers to addresses. The completed code is then output to the Object file.

- as the code is completed any references to non-existent lines are detected and an error message is output.

- once all the completed code has been output to the Object file then the Data Statement and Semi-compiled files are deleted.

- if necessary the VL file is created (if not the variable list is appended to the Object file).

- if an overlay file is being compiled and some new variables names have been found during the compilation then a new VN file is created.

- the compilation statistics are output (if required).

2.3 The Object file structure.

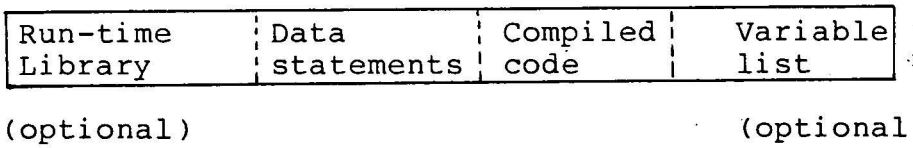

FIGURE 3

Figure 3 shows the internal structure of the Object file.

The Run-time library is a set of routines that are incorporated in each compiled program and which implement such functions as integer arithmetic, FOR loop control etc. In version 4 the Run-time library is incorporated within SYSTEM 96 and is therefore not needed with the compiled program.

The Data statements section consists of the text from all Data statements in the program .

The Compiled code is the actual program and consists primarily of calls to routines that are within the Run-time library.

The Variable list contains an entry for each scalar variable (ie. each non-array variable) used in the program. The format of the variable list is the same as the list used by the interpreter. The Variable list is included in the program so that it does not have to be re-created every time that the program is run as is the case for the interpreter. In version 4 or when the overlay facilities are in use then the variable list is not in the Object file as it will not necessarily be stored immediately following the program. In such cases the variable list will be loaded to memory from the VL file.

Note that a facility is provided to create an Object file without a Run-time Library for versions 1,2 & 3 (see ** NL command in Appendix B). This facility can be used for programs that are never run on their own but are always LOADed from other programs (so that the Run-time library will already be in memory). The advantage is that the Object file will be significantly smaller and will therefore occupy less disk space and will LOAD faster.

3. Installation of DTL-BASIC.

Contents.

DTL-BASIC compiler disk
DTL-BASIC manual
'Compiler' security key
'Run-time' security key

DTL-BASIC is supplied on a single floppy disk which is write protected and the first action should be to take several copies of the disk for backup purposes.

In order to protect Drive Technology's copyright and the copyright of DTL-BASIC users over their own programs DTL-BASIC incorporates a protection feature that ensures that the compiler and compiled programs will only run in the presence of a special security key attached to one of the machine's cassette ports.

The security key marked 'Compiler' supplied with DTL-BASIC must be fitted to one of the cassette ports before the compiler will run. Each CBM model has two cassette ports (refer to your Commodore manual if you do not know the location of the cassette ports) and the Compiler key may be fitted to either. It is important to fit the key the correct way;ie. ensure that the labelled side faces upwards.

A second key marked 'Run-time' is supplied to enable compiled programs to be run on a second machine. If it is required to run compiled programs on a number of machines then additional 'Run-time' keys may be obtained from the Distributors.

```
* * * * * * * * * * * * * * * * * * * * * * * *
*                                               *
*                   IMPORTANT                   *
*                                               *
*    Always ensure that the machine is switched *
*    off before fitting or removing  a security *
*    key that is fitted internally, eg. to      *
*    the first casettte port on a 3032.         *
*                                               *
* * * * * * * * * * * * * * * * * * * * * * * *
```

4. Operation of DTL-BASIC.

4.1 Operating Instructions.

DTL-BASIC has been designed to be very simple and easy to use.

To run the compiler : NOT US — WE USE VERSION 2.

-if using version 4 of the compiler then first ensure that SYSTEM 96 is installed on the machine. Once SYSTEM 96 is initialised then the SYSTEM 96 key may be removed if required;

-insert the compiler disk in drive 0 and the disk containing the program to be compiled (ie.the Source file) in drive 1;

-ensure that the 'Compiler' security key is fitted;

-if the compiled program is to be 'keyed' to a key other than the 'Compiler' key then ensure that the other key is fitted as well (see section 9);

-load and run the first program on drive zero;

The compiler will display a list of options on the screen. If it is the first time that the compiler has been run on the disk in drive 1 then the option input fields will be blank. Otherwise the fields will display the options selected the last time the compiler was run on that disk and it will only be necessary to type in any alterations.

On the first run the option input fields will be blank and it will be necessary to type :

-the name of the Source file.

-the name of the Object file (ie. the name to be used for the compiled program). Any existing file of this name on drive 1 will be overwritten.

If no listing is required simply key ↑ to start the compilation. The ↑ key is normally used to start the compilation and indicates to the compiler that the information displayed is correct and the compilation is to start.

If any form of listing is required then answer Y or N to the remaining questions (followed by Return). Before typing any field ↑ can be keyed to start the compilation,ie. a blank field indicates a No answer.

If the printer is to be used the compiler will first print a heading containing the data from the screen. The contents of the 'Run Identity' field are included in the heading so that individual listings can be identified. For example the 'Run Identity' could be the time and/or date so that when a number of listings are produced their chronological sequence can be determined.

4.2 Operating Notes

1. The compiler requires a 32K machine with disks but depending upon their size compiled programs may run on 8 or 16K machines and may be loaded from cassette.

2. As explained earlier the compiler performs two passes of the source program. During each pass the line number being processed is displayed and any errors detected are displayed on the screen or printer depending upon the option selected (see section 8).

3. If any output is to be made to the printer the compiler will check that the printer is ready before starting the compilation;ie. that the i/o status variable ST equals 0. If ST is not equal to zero the compiler will wait until it either becomes zero or until a space is pressed (the option to press space is provided because some printers are non-standard and have a non-zero ST value when they are ready).

4. As was explained in section 2 the compiler creates several files on drive 1 so that a certain amount of free space must be available when the compilation starts (the amount of space needed depends upon the size of the program being compiled).

5. When the options are displayed before the start of a compilation then if the user can not correctly recall the name of the file to be compiled the directory from drive 1 may be displayed by keying @. If the wrong disk is found to be in drive 1 then / may be keyed to enable the disk to be changed.

6. At the end of the compilation the work files W1 and W3 are automatically deleted but if for some reason the compilation is not allowed to complete normally then the files may be left on Drive 1. If this occurs they will be deleted by the next compilation or may be deleted manually.

7. If for any reason it is not required to proceed with a compilation (eg. because the file disk is full) then ← should be keyed to exit from DTL-BASIC.

NOT US we have release 4.

8. From release 5 onwards the compiler deletes the Object file at the end compilation if any errors are found during the compilation (see section 8 for a description of possible errors). This is done to ensure that the occurence of errors at compile time cannot be missed by the user.

4.3 Special Operation Features

1. The compiler allows the user to specify any name for the Object file but if the last four characters of the Source file name are "-src" the the object file name will be generated automatically.

e.g. if the Source file name is

"test-src"

then the compiler will call the Object file

"test"

2. If it is required to compile a number of files then they may all be compiled individually or a Control file may be created to enable them all to be compiled in one run (release 5 or later). Not US – release 4

A Control file should have a name which has the last 4 characters equal to "-con",eg. "compile-con".

A control file is a normal program file that contains 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 compile and the second file name should be the name of the corresponding Object file (unless the "-src" option is used in which case the compiler will generate the Object file name automatically). The next file name will be the name of the second Source file to compile and so on . . .

eg. If the contents of the Control file are :

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

Then 3 compilations will take place, 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 will then apply to all the compilations from the Control file. It is strongly recommended that the option to print errors should be selected to ensure that any errors are not lost.)

3. The 'Run Identity' field of the option display enables listing to be identified. One simple way of using this field to ensure that the correct chronological sequence of listings may always be determined is to input a 'Run Identity' of ✻1. The compiler makes a special check for the ✻ and then expects to find a number; when the compiler re-writes the Options file it will increment the number found. This means that if the same file is compiled a number of times the listings will be numbered in ascending order.

4.4 Operation of compiled programs

1. Compiled programs are simply loaded and run just like un-compiled programs and should perform exactly the same. If they do not work as expected the see section 4.5.

2. For version 1,2 & 3 then when CONT would be used for uncompiled programs SYS 1069 should be used for compiled programs. The program will crash if CONT is used.

For version 4 CONT may be used exactly as for uncompiled programs (SYS 1069 should not be used).

3. When a program is run for which a VL file exists then the VL file will automatically be loaded to memory the first time that the program is run. If the VL file is not found on either disk then a FILE NOT FOUND error will occur.

4.5 Problems ?

1. When a compiled program runs then if the system it totally re-initialised (ie. does a 'warm' start) then check that the correct security key is fitted with the labelled side upwards.

2. When compiled programs run then thorough run-time checks are applied and may cause the program to stop (see section 8.3). If the error cannot be explained then check that uncompiled program works correctly. If it gives the same error then correct the fault.

3. If an unexplained "FILE NOT FOUND" error occurs on version 4 or when the overlay facilties are in use it is probable that the VL file does not exist. Note that this file must not be renamed after being produced by the compiler; the Object file may be renamed but not the VL file.

Not v3 — version 2

4. For version 1,2 & 3 it is not possible for one compiled program to LOAD another unless the second is also compiled.

5. If the result of arithmetic appear to be wrong or the compiled program appears to be running but is not performing correctly then it is likely that special Integer mode must be selected (see section 5.4).

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

7. When compiled programs are to be used together (ie. they load each other) then all the programs must be compiled by the same version of the compiler at the same release number (the release number is the second digit of the compiler identifier, eg. DTL-BASIC 2.4 is version 2 release 4).

8. The compiler will not run correctly if the DOS support utility is loaded.

9. When compiling programs from a disk drive with DOS 1 (ie. a 2040 or 3040 with the original ROMs) then then the Source file may become corrupt if it is compiled immediately after being RENAMEd. The solution is to INITIALISE the drive after a RENAME.

4.6 Programming Notes

1. If one program loads another program (eg. vie a DLOAD or LOAD) then the source file will have to be RENAMEd before compilation. For example, if there are two programs in files PROGA and PROGB and they are compiled to produce the Object files C-PROGA and C-PROGB, and if PROGA contains the statement

```
DLOAD "PROGB"
```

then when C-PROGA runs it will load the source of the second program rather than the Object file. The best solution would be to RENAME the files to be PROGA-SRC and PROGB-SRC before starting the compilation. This would produce Object files called PROGA and PROGB respectively so that the DLOAD statement would work correctly.

2. Compiled programs can disable the Stop key in exactly the same way as under the interpreter. However there is an additional means of achieving this for compiled programs that has the advantage of not affecting the clock and makes compiled programs marginally faster.

The method of doing this is different for release 5 from that used in release 4. This means that programs being moved from release 4 to release 5 may need a slight alteration.

Release 4 - to disable the Stop key use POKE 1072,1 and use POKE 1072,0 to enable it.

Release 5 - to disable the Stop key use REM ** DS and use REM ** ES to enable it.

3. It has already been stressed that it is desirable to be able to run a program in both compiled form and uncompiled form. In some situations there may be a few statements that are required to be executed when the program is compiled but not when it is uncompiled (and vice versa). This may be achieved by means of a simple test of the form :

```
CP = PEEK(PEEK(41)*256+peek(40)+4) = 158
```

this will set CP to the value 0 in an uncompiled program and -1 in a compiled program. CP may then subsequently be tested as required, eg.

```
IF CP THEN PRINT "THIS PROGRAM IS COMPILED"
```

Note that the test used to set CP is actually testing the first character of the first line of the program (which is always the SYS token (ie.158) in a compiled program). The test will therefore only work if the first line of the uncompiled program does not start with SYS.

4. Release 5 of DTL-BASIC compiles standard CBM Basic plus the extensions provided to support the Overlay system (CALL,ENTER,OLOAD etc.). These extensions use token numbers that are unused by CBM Basic but which may be used by a user's own extensions (implemented by assembler subroutines). In such a situation the the compiler can be instructed by means of a REM ** NE directive to not compile any of its own extensions but to treat any non standard tokens as user extensions.

Not US.

5. After a compilation then if compiled programs are copied to another disk then any VL files should be copied as well as the Object files (there is no need to copy the VN files).

CANNOT USE DS$ IN IF STMT eg IF DS$ = TS$ THEN ...

∴ USE DZ$ = DS$: IF DZ$ = TS$ THEN ---.

MUST SPACE OUT "OR" IN IF STMT ie

IF A$ = B$ ORC$ = D$ THEN --- will fail

∴ USE IF A$ = B$ OR C$ = D$ THEN ---

CONVERTING to integers items turned in a basic extension may not work. eg !print "-"; pn; "-" fails to convert pn to pn%. Whether the quotes are essential to the failure is not known. 4/2/85.

5. Integer Arithmetic Facilities.

Commodore Basic supports integer as well as real variables. Unfortunately, the interpreter does not perform true integer operations as it converts all integer values to real format before performing any operation. The result of the operation is then converted back to integer. A consequence of this is that if integers are used then the program will actually run slower because of the mode conversions on each operation. For this reason most existing programs do not use integers even though the vast majority of variables normally hold integer values. Integers are sometimes used for arrays as this gives a space saving.

DTL-BASIC differs from the interpreter in that it implements true integer operations for all operators when both operands are integer,i.e. all arithmetic,boolean and relational operators. New programs which are to be compiled should therefore use integers wherever possible.

The arithmetic package within the Run-time Library has been designed to achieve true compatability with the interpreter and to ensure that the best results are achieved it will help to understand the following points :

1. Wherever possible parts of expressions are calculated in integer mode and the result converted to real where necessary, e.g. in the expression -

```
A = J%*(I%*10+6) + B
    ------------
```

the underlined section would be calculated in integer mode and the result converted to real for the rest of the statement.

2. Whilst in integer mode then if integer overflow occurs the operands are converted to real and the operation is repeated in real mode, e.g. in the above example if I% holds 30,000 then the integer multiplication by 10 would cause overflow as the value would exceed 32K and the operation would be repeated in real mode.

3. If integer overflow is very likely the programmer can avoid it by using real constants.

eg. if the above statement is written as :

```
A = J%*(I%*10.0+6) + B
```

then all operations will be in real mode.

4. Special consideration is needed for the operators divide and exponentiation because when applied to integer operands they can yield a fractional result. The compiler cannot know whether the programmer requires the fractional part or not,e.g. the expression.

```
3/2*4
```

will yield 4 if integer operations are used and 6 for real operations.

If not instructed otherwise the compiler will perform integer divide and exponentiation when both operands are integer (the above example would therefore normally give the answer 4). This option was chosen because it is what the programmer most often requires and because it runs the fastest.

If the prograammer wants a real divide or exponentiation then the Special Integer mode should be used. This mode can be selected by including the statement :

```
REM ** SI
```

before any statements (other than REMs) at the start of the program.

5. If an existing program is being compiled and any problems are found with the results of calculations then special integer mode should be used as this makes the arithmetic fully compatible with the interpreter.

6. As has been explained programs that use integers as much as is possible will run faster than if reals are used. This is no problem for new programs because the programmer should simply use integers whenever possible. However, it is not so simple for existing programs as it can be a lot of work to change many variables to integers and the edits may introduce errors.

DTL-BASIC includes a special feature that can convert variables (both scalars and arrays) automatically to integers without any changes to the program. There are two ways of doing this depending upon the number of variables to be converted.

The first method is to use the 'Convert Specific' statement at the start of the program. This statement instructs the compiler to convert the named variables to integers, eg.

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

This instructs the compiler that the named variables are to be converted to integers. In the example the variables will become A1%,ZZ%,X2%,X3%. The compiler will flag an error if variables with these names already exist. In order to avoid name clashes the CS statement can specify new names for some or all of the variables, eg. if ZZ% already exists but ZQ% does not exist then the statement would become :

```
REM ** CS (A1,ZZ => ZQ%,X2,X3)
```

Note:

- when a name change is specified then the first character of the two names must be the same;

- there can be several CS statements but the maximum number of variables that can be converted is 128;

- if the variable to be converted has more than 2 chracters in it's name then only the first 2 should be used in a CS or CE statement (this restriction does not apply from release 5 onwards).

When a large number of variables are to be converted it will be better to use the second method of conversion. This is specified by the 'Convert Excluding' statement and converts ALL real variables excluding those explicitly named, eg.

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

will convert all reals;

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

will convert all reals except those named I1,I2,I3.

The conversion rules are the same as for the CS statement and name clashes can be avoided in a similar way, eg.

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

will convert all reals excluding those named A and C. Variables named B will be converted and will be converted to B1%.

Note that CS and CE statements cannot both be used in the same program but an SI statement can be used with either. Also, these statements should be the first in the program.

7. Even for new programs there may be a need to use the CS or CE statements because the interpreter does not allow integer FOR variables even though in most programs the FOR variables will only hold integers. Therefore, if it is required to debug the program using the interpreter then real variables must be used in FOR statements. When a program is compiled then the best performance will be achieved if a CS or CE statement is used to convert the FOR variables to integers.

6. Making the most of DTL-BASIC.

The most obvious benefits of using DTL-BASIC are :

-the improved performance for all compiled programs;

-the reduced size for large compiled programs;

-the compatability with the Commodore Interpreter.

There are several other benefits which are possibly not so obvious and this section will describe these benefits in more detail.

1. The compiler will accept extensions to Basic implemented by assembler code in ROM or RAM.

This sounds almost too good to be true but in fact it does work. What happens is that when the compiler is checking the syntax of a statement then if it cannot recognise the first character of the statement (ie. if the statement does not start with either a legal token or an alphabetic character) it assumes that the statement is valid but uses an extension to standard Commodore Basic. The compiler embeds the text of the statement in the program and precedes it by a special code. When the program is run the Run-time library detects this code and sets up the appropriate pointers for the interpreter and calls the interpreter (ie. the standard Commodore interpreter in ROM). The interpreter processes the statement and providing the program has already inserted Wedge code in page 0 (eg. by a SYS call or by POKEs) and the assembler routine to process the statement is in RAM or ROM then the statement will be obeyed. The compiler plants a SYS call after the special statement so that once the statement has been obeyed the interpreter returns control to the Run-library. See Appendix D for some notes on the use of the compiler with some common ROM chips.

2. Compiled programs cannot be listed or altered by the user.

When un-compiled programs are run it is comparatively easy for the user, especially one who does not know much about programming, to delete lines or to add new lines which will obviously have unpredictable effects and can appear to be errors in the original program. Such accidental or intended alterations to programs can cause many problems. With compiled programs such problems are avoided. In addition, as compiled programs cannot be listed other programmers are not able to copy individual routines or find out how a program works.

3. Use of the compiler can result in significantly reduced development and maintenance costs.

Software development and maintenance costs have always been frighteningly high. As hardware costs come down software costs continue to rise. When programming in Basic on Commodore machines these costs can be much higher than they need to be because of the bad practices that have to be adopted to make interpreted programs as fast and as small as possible.

These practices (some of which are advocated by Commodore in Appendix E of their Basic manual) can result in programs that are exceptionally difficult to understand and to modify.

Also the program cannot be organised as a set of modules. This means that when new programs are developed then even if they share a function with an existing program it is normally not possible to simply extract useful routines from the program without the routines requiring modification and re-testing. If it is intended to compile a program once is is working then none of these bad practices need be used and the program can be made much easier to understand and to change. The program can also be made more modular.

The bad practices referred to above include :

-declaring the most commonly used variables at the head of the program;

-placing commonly used statements at the head of the program;

-using each variable for many different purposes;

-not using many REMs;

-putting as many statements as possible on one line;

-declaring all variables before any large arrays are declared;

-not using any spaces in the program;

-replacing parts of the program by assembler subroutines. This is bad because assembler code is far more expensive to develop and to maintain than Basic.

If the program is to be compiled the sort of techniques that can be adopted instead of those above are :

-organising the program as a set of independant modules with all related code and variables together;

-preceding each module with a description of it's function and a definition of it's variables in REM statements;

-allocating each module a set of line numbers and variables,eg. Module A could use lines 5000-6000 and only use variables starting with the letter A;

-defining a small set of entry points for each module;

-having a defined interface between modules.

If such techniques are used then each module can be considered in isolation from the rest of the program which makes the module much easier to understand and to modify. Similarly when a new program is to be developed then it is likely that modules can be extracted from existing programs and re-used without modification or re-testing.

Use of DTL-BASIC can enable many bad practices to be avoided without sacrificing performance and memory and if the above techniques are used, or others like them, then it is possible to produce well structured, maintainable programs (this is especially true if the Overlay facilities are used).

4. Compiled programs utilise the stack more efficiently than the interpreter thus enabling more complex programs to be run.

When the interpreter runs it uses 18 bytes of stack for each FOR statement (not 16 as quoted in the Commodore manual) and 5 bytes for each GOSUB. When a compiled program runs 10 bytes are used for each FOR (providing the FOR variable is an integer otherwise it also uses 18) and 3 bytes for each GOSUB. Compiled programs can therefore have a greater depth of nesting than un-compiled programs. Compiled programs can also evaluate more complex expressions than the interpreter.

5. Use of the Overlay facilities that are available from release 5 onwards (see section 7) can result in a number of advantages :

- the time involved in loading Basic code can be reduced;

- the disk space required to hold a package of Basic programs can be significantly reduced by the use of subroutine libraries;

- the use of subroutine libraries where only one copy of each routine need be maintained means that development and maintenance will be faster and more reliable (and will therefore cost less);

7. Chaining and Overlaying programs.

Program chaining is the practice of one program loading another program on top of itself and for the second program to then be automatically run. Chaining is commonly used to link a number of programs together to form a more powerful package than would be possible with a single program.

Overlaying is similar to chaining but rather than overwriting all the the program code in memory only part of it is affected. Overlaying is a more sophisticated technique than chaining and generally should give better performance and greater flexibilty. When overlaying is used the program space in normally divided into a number of separate program areas or overlay areas. The first overlay area is special and is known as the Root area. The program in the root is responsible for the definition of the other overlay areas and for loading the correct program overay to each area when required.

A typical organisation for a simple menu driven application package would be to have three overlay areas used as follows :

overlay 0 - this is the root overlay and holds the menu program;

overlay 1 - holds the current application program (this is loaded by the menu when the user selects an option);

overlay 2 - holds a library of all subroutines that are common to two or more of the application programs.

The advantages of overlaying when compared to chaining for the above example are:

- the menu program is store resident giving faster response to the user;

- only one copy of each common subroutine is required rather than one for each application program; this makes for easier and faster debugging and maintenance;

- each overlay program will be smaller than the equivalent program in a chained system and this means less time spent loading program code and less disk space needed.

DTL-BASIC supports both chaining and overlaying and the rest of this section outlines how to use these techniques for the various versions of the compiler that are available.

7.1 Release 4 facilities.

Release 4 supports chaining but not overlays.

When programs are chained on the interpreter then problems can occur if the program being loaded is larger than the program in memory. This problem can be overcome by POKEing the start of variable pointer before loading the program or increasing the size of the first program in the chain. Such practices are not necessary with compiled programs as each program includes its own variable list in the program so that the existing variable list is also overwritten. However, this does mean that for release 4 it is not possible to share variables between programs that are chained.

When chaining is used then the No Library feature (REM ** NL) may be employed to omit the Run-time library from each program except the first (see section 2.3).

7.2 Release 5 facilities.

Release 5 of DTL-BASIC supports both chaining and overlays.

The chaining facilities are improved from release 4 in that chained program may share variables or may each have their own set of variables

At present release 5 is only available for version 4.

7.2.1 Program chaining in version 4.

If one program loads another program via a DLOAD or LOAD command then the second program will share variables with the first. However, this will only work correctly if the first program in the chain is compiled with a ** RO directive (see below) to force the compiler to create a VN file and if all the other programs are compiled with a ** VN directive to make the compiler read the VN file. This ensures that all the programs are compiled to use the same variable addresses.

If it is not required to share variables between chained programs then this can be achieved by not using either of the two directives mentioned above and by preceeding the call to DLOAD (or LOAD) by the following statement :

```
POKE 1204,0
```

If this technique is used then the program loaded will load its own variable list to memory when it starts to run. The POKE statement will have no effect when the uncompiled program is run.

7.2.2 Program overlays in version 4.

Version 4 supports the Overlay facilities provided by SYSTEM 96 (see section 5 of the SYSTEM 96 manual). This means that a package may be developed and debugged un-comipled and when working may then be compiled.

Two directives are provided by DTL-BASIC to enable overlays to be successfully compiled :

- the Root directive REM ** RO

  this tells the compiler it is compiling the root overlay and has the effect that at the end of the compilation a VN file is generated that records all the variables and arrays used and the addresses allocated to them. The name of the VN file will be "VN-<name>" where <name> is the name of the compiled root program. There should only be one root program for each overlayed package and it should occupy overlay 0 when it runs and should be the first program of the package to be run.

- the Variable Name directive REM ** VN "<name>"

  this directive tells the compiler that it is compiling an overlay and that the compiler should access the file "VN-<name> to find the names and addresses of all variables and arrays that exist in the root overlay and the other program overlays that have already been compiled. At the end of the compilation then if that overlay has referenced any names that were not in the VN file then a new VN file will be created that includes the new names.

To summarise, the root overlay should include the statement

```
REM ** RO
```

at the start of the program and all the other overlays should include the statement of the form

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

(in this example MENU is the name of the root).

Notes.

1. There is a restriction for all overlays other than the root that DIM statements must be used for all arrays that are dimensioned in the overlay,ie. arrays without DIM statements will not be automatically declared. The compiler checks for this case and will flag an error if no DIM statement is found for any arrays that are not explicitly dimensioned. Note that this only applies to arrays that are dimensioned in that overlay,ie. those that do not exist in the VN file at the start of the compilation.

2. The code produced by the compiler is relocatable so that there is no need to recompile all the overlays if the sizes of the overlay areas is altered;ie. if the MAP statement is changed.

3. Once a variable name exists in the VN file the it will be allocated space in the variable list even if as a result of alterations to the package it is no longer used by any program. This does not cause any problem other than that a certain amount of data space will be wasted. If it is required to remove such variables from the variable list then it will be necessary to delete the VN file and recompile all the programs (starting with the root).

4. Although overlays are normally loaded from the root it is possible for any overlay to load a overlay to another overlay area. However, the overlay structure may only be defined in the root (ie. MAP statements may only be obeyed in the root).

5. DLOAD or LOAD may be used within a program in an overlay area to load another program into the same area.

8. Errors.

The compiler performs exhaustive checks whilst 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 whilst the compiled program is run to detect errors that cannot be found at compile time. From release 5 onwards then 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 programm 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

The following sections describe each type of error in more detail.

8.1 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 C contains a full list of the error numbers and their meanings.

8.2 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,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 an overlay for which no DIM statement has been compiled (see section 7.2.2).

8.3 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 errors 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 Basic 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 its 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 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 drive 1. Also, if the error occurs within an overlay then ERROR LOCATE should be run within that overlay,ie. at the same address as the overlay in which the error occurred.

8.4 Warnings.

Warning messages occur when the compiler has detected an extension to Basic (see section 6.1) 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 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(**NW) at the start of the program. In this cases no warning messages will be produced but a count will still be generated.

9. Use of Security Keys.

Section 4 describes how to compile a program that will run on a machine fitted with the 'Compiler' security key. 'Compiler' keys are only supplied together with the DTL-BASIC compiler and can therefore not be used by users who wish to run compiled programs on a number of machines. A second key is supplied with DTL-BASIC and this is known as the 'Run-time' key and this may be used to run compiled programs on a second machine. Additional 'Run-time keys can be supplied to enable compiled programs to be run on any number of machines.

In order to produce a program that will run with a 'Run-time' key all that is necessary is to ensure that such a key is fitted to the machine during compilation (as well as the 'Compiler' key,ie. one key on each of the two cassette ports). When the compiler detects the two keys it will produce a program that will run with the 'Run-time' key fitted. By making a number of copies of the compiled program the user can then run the program on any machine with a 'Run-time' key fitted.

As compiled programs will only run on machines fitted with keys the user gains valuable protection for the compiled programs but if the standard 'Run-time' key is used then this protection is not absolute as other users of DTL-BASIC can also obtain the standard keys. However, users who require complete protection for their product can be supplied with unique keys which contain their own serial number. Once a serial number has been allocated to a particular user then only that user will be able to obtain keys containing that number.

When the compiler detects such a 'Software House' key during compilation it will first ask the user to input the serial number and will not compile the program unless the number is input correctly. In this way only the registered user of a key can compile programs to use that key (as long as the the user does not reveal the serial number to other users).

By using DTL-BASIC together with 'Software House' keys a supplier can gain comprehensive protection for a product. The protection system will work equally well with cassette based and Computhink disk based products as well as CBM disk based products. In addition, the end-user is able to take backup copies for security.

10. Compatability.

As has already been explained DTL-BASIC has been designed to achieve a high degree of compatability with the Commodore BASIC Interpreter. Obviously no two products can be totally compatible otherwise they would be effectively the same. For example compiled programs run faster than un-compiled ones and whilst this is normally an advantage there can be situations where this will require the source to be altered to slow the compiled program down,eg. when a FOR loop is used to create a specific delay.

This sections lists all the known incompatabilities with the Interpreter in the current release (some may be removed in future releases) :

-compiled programs cannot be re-started with a CONT although the same effect can be achieved by typing SYS1069 (this restriction does not apply to version 4).

-calls to a user function cannot be made from the following statements or from statements which are extensions to Basic :

PRINT,PRINT#,INPUT,INPUT#,GET,GET#,OPEN,CLOSE

it is very rarely that FNs are used in such statements but if they occur the compiler will report an error and a minor change to the Source file will be required,eg.the statement :

```
PRINT "THE VALUE IS ";FNA(B)
```

could be changed to

```
A1=FNA(B):PRINT "THE VALUE IS ";A1
```

-a compiled program can only be started at the first line by a Direct command,ie. by RUN and not by RUN <line number> (eg. not by RUN 200). Note that this does not apply to RUN statements within the program.

-because the program is compiled the source text is not in memory when the program runs so that any statements that access the source text in any way will not work.

-the format of the variable list is fully compatible with the interpreter so that when the program is stopped variables can be displayed/changed via Direct commands. The only difference is that the order of variables in the list may be different from when the program runs under the interpreter. With the interpreter the order will be the order that the variables are referenced at run-time; for compiled programs the order will be the order in which the compiler meets the variables at compile-time. This difference could only conceivable affect assembler routines and even then it is very unlikely.

-the format of the Array list is also the same as for the interpreter although the order of the arrays in the list may also differ and there is one additional array in the list called the Array Table. The Array Table is the first array in the list and has a non-standard header. The Array Table is used to hold pointers to the start of each normal array and is necessary because the addresses of arrays cannot always be calculated at compile time. When a program makes an array access then the Run-time library accesses the Array Table to find the array address.

-when a compiled program is LOADed from another compiled program then the two programs cannot share variables,ie. the new program has its own set of variables after being LOADed (this restriction does not apply to release 5).

See also section 4.6

See also section 5.4

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 it 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 complimentary; 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 Summary of Compiler Directives

There are a number of Compiler Directives (ie. instructions to the compiler) that may be incorporated within a program. The directives have the form of REM statements so that programs containing directives may still be run under the Interpreter. The standard format of a directive is -

```
REM ** <directive id> <directive text>
```

This form has been chosen to minimise the chance that an existing REM will be seen as a directive by the compiler. Most directives should only be used at the start of a program (ie. before any non REM statements have been found) but two directives (ES & DS) can occur within the body of a program.

<directive id> consists of two alphabetic characters and at present there are ten separate directives -

Directive	Reference	
CS - Convert Specific	5.6	
CE - Convert Excluding	5.6	
SI - Special Integer	5.4	
NL - No Library	2.3	
NW - No Warnings	8.4	
RO - Root Overlay	7.2	**
VN - Variable Name file for overlay	7.2	**
NE - No standard extensions	4.6	**
DS - Disable Stop key	4.6	**
ES - Enable Stop key	4.6	**

** - not available in release 4

The rest of the Appendix summarises the function of each directive.

REM ** CS (A,B,X1,G5,GG,Z9=>ZZ%)

means convert A,B,X1,G5,GG,Z9 to A%,B%,X1%,G5%,GG%,ZZ% respectively.

REM ** CE (W4,W,FF=>F1%,MM)

means convert all reals to integers of the same name except W4,W,MM which are not to be converted and FF which is to be converted and is to be called F1%.

REM ** SI

instructs the compiler that when both operands are integer and the operator is either divide or exponentiation then perform a real operation instead of an integer operation.

REM ** NL

instructs the compiler to not include the Run-time library in the Object file. This will mean that the Object file will only run when it has been LOADed from within another compiled program but has the advantage that the Object file will need less disk space and will LOAD faster. Programs compiled with **NL should not use a simple RUN; RUN statements should be replaced by RUN <n> where <n> is the number of the first line of the program.

For programs using this directive any REM ** SI directives will be ignored and the integer mode used will be determined by whether the program that loaded it (ie. the one with the Run-time library) had a REM ** SI directive.

This directive is not needed for version 4 because the Run-time library is never incorporated in the program.

REM ** NW

specifies that no warning messages are to be output during compilation.

REM ** RO

indicates to the compiler that it is compiling a root overlay program and that a VN file is to be generated if one does not exist already.

REM ** VN "<name>"

indicates to the compiler that an overlay is being compiled and that the list of variables names will be found in file "vn-<name>" where <name> it the name of the root overlay.

REM ** NE

inhibits the compiler from accepting any of the extensions to Basic provided by SYSTEM 96 (they will be treated like any normal extensions to Basic).

REM ** DS

disables the STOP key at run time (without affecting the clock). This directive (and REM ** ES) can occur anywhere in a program

REM ** ES

enables the STOP key at run time.

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, LIST or SELECT)
19	program too big (shouldn't occur 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
39	no FNs allowed in this statement type (an edit is required - see section 7)
40	no '=' where one expected
41	default arrays found in overlay

Appendix D Use of ROM chips with compiled programs

There are an increasing number of ROM chips which can be used in CBM machines and many of these provide extensions to Basic. DTL-BASIC has special features that are designed to enable such extensions to be compiled. There are three main types of extensions that are used and these can all be handled by DTL-BASIC :

- wedges; ie. extensions to Basic that are detected by a routine called from the CHRGET routine in page zero and to keep the ROM routine simple the extensions normally start with a non alpha-numeric character. When the compiler detects such a statement it embeds the whole statement in the compiled program and at run time passes it through to the interpreter which picks up the wedge (see section 6.1). ROM routines often access Basic variables and arrays and as compiled programs store variables and arrays the same way as the interpreter this presents no problems.

  Note that the only situation where wedges will not work with the compiler is those that start with an alphabetic character or which include a ":" within the statement (the compiler will treat the ":" as the end of the statement).

- extra keywords; this technique is very similar to a wedge except that the statement starts with one of the unused token characters (this effectively produces extra keywords). When the compiler detects a statement that starts with an unused token it treats the statement in the same way as a wedge. The only complication in this case is that if a program is listed by the compiler the compiler does not know what keyword to associate with the extra tokens so that the listing will be incorrect for the lines containing extensions. This problem can be overcome by using the direct command LIST to obtain a listing in the normal way instead of instructing the compiler to produce a listing.

- extended SYS calls; some ROM chips use SYS calls followed by parameters. This technique works on the interpreter because the routine moves the text pointer past the parameters so that on exit from the routine the interpreter 'sees' the end of the statement rather than the parameters. Such SYS calls will work with the compiler because on entry to the routine the text pointer is set to point to the first character after the routine address.

  Note that the parameter list should not include any ":" (other than at the end) as the compiler treats ":"s as the statement terminator.

The following ROM chips have been used successfully with DTL-BASIC :

- Kram
- Super Kram
- Computhink disk system
- JCL Business ROM
- Command-O
- Disk-O-Pro
- Taylor Wilson MPS system (not a ROM but does use extensions to Basic see note below)

This is not a complete list of ROM chips that will work with the compiler it is simply a list of some of the chips that have been tested so far. As yet no product has been found not to work so that it is very likely that other products not in the above list will work.

The following notes are relevant to users who wish to use the above products with DTL-BASIC

1. Whilst compiled programs can include Computhink commands and can be run from Computhink drives the compiler itself can only be run from Commodore drives.

2. The only known problem that may occur when using the JCL Business ROM with compiled programs relates to the multiple statement feature of the PRINT AT statement. This feature allows several PRINT AT statements to be cascaded by the use of ":"s to separate the individual fields. The problem occurs because the compiler will treat the ":" as the end of the statement (the effect will be to cause a syntax error at run time); the problem can be avoided by writing the statements out in full, eg.

```
410 PRINT (X,X)S$:(24-X,X)S$
```

should be written as

```
410 PRINT (X,X)S$:PRINT (24-X,X)S$
```

3. All Command-O and Disk-O-Pro program statements work when compiled with the exception of MERGE and MERGE (which expect Basic source to merge and which therefore get confused by a compiled program). Also the DS and DS$ features of Disk-O-Pro will not work.

4. Before running the compiler then if either Command-O or Disk-O-Pro is enabled then perform an OUT statement from the keyboard to disable the enhanced screen editor. This is because the enhanced screen editor does not allow LOAD or DLOAD statements to be obeyed.

5. The MPS system uses some assembler routines loaded to RAM to implement the BOPEN and BCLOSE functions. These extensions cause a problem because they are not implemented as true extra keywords and consist of the letter B followed by a keyword. As explained previously the compiler does not recognise extensions starting with an alphabetic character. This problem can be overcome by altering the routines in file MPS1 to look for a % instead of a B. This is achieved by altering location $058F from $42 to $25 and by using %OPEN and %CLOSE in compiled programs.

[illegible] have been used successfully with DIC-BASIC :

[illegible]

[illegible] will work with the [illegible] that it is very [illegible]

[illegible] commands and can be [illegible]

[illegible] DIC. Business [illegible] AT statement [illegible] fields. [illegible] as the end [illegible] at run [illegible] the screen out in [illegible]

[illegible]

[illegible] Basic source to [illegible] program). Also the [illegible]

[illegible] Command-O or Dick-O-Pro [illegible] to disable [illegible] editor [illegible]

[illegible] to TRAN to [illegible] extensions cause a [illegible] keywords and [illegible] followed by a keyword, as explained previously [illegible] This problem can be overcome by altering the routines in [illegible] looking for a $ instead of a %. This is achieved by altering [illegible] to $24 and by making COPAN and $CLOSE in [illegible].