
BLITZ

INTRODUCTION

BLITZ is a versatile and easy to use compiler for Commodore computers which makes BASIC programs run much faster. Most programs can be compiled without any alteration whatsoever (although you may wish to take advantage of some of the features that BLITZ adds to standard BASIC), and the whole compiling process usually takes no more than a few minutes!

Compiled BASIC programs cannot be listed or altered by users and so compiling provides an element of protection against unauthorised copying and alteration of programs.

BLITZ is available for the Commodore 64, 128, and PET computers (2001/3000,4000 and 8000 series). Compiled programs can be used with either disk or tape, but BLITZ 64 is the only version of the compiler which is available on tape as well as disk.

The compiler translates BASIC programs into a far more efficient form (called *P-code*), which can be executed much more quickly than the original program. Large BASIC programs also become shorter and occupy less memory and disk space after compilation.

BLITZ can recognise most extensions to Commodore BASIC, so you can compile programs which make use of them (the utility supporting the extensions must be present when the compiled program is run).

BLITZ works very well with a single disk drive, but if you have a PET or a 128 with a dual disk unit or two single drives, there are some more advanced compiling options that you can use.

The main part of this manual assumes that you are using BLITZ 64 or BLITZ 128. If you have purchased BLITZ for a PET computer, you should also refer to the section at the back of the manual which explains how your version of BLITZ differs.

GETTING STARTED

BLITZ for the 64 is available on disk or tape, but BLITZ for the 128 is for disk users only. If you have BLITZ 64 on disk type **LOAD"*",8** followed by **RUN**. If you have the 128 version, or the Commodore 64 version on tape, hold down the *shift* key and press *RUN/STOP*.

When the menu appears remove the BLITZ disk or tape and insert one containing the program to be compiled.

Using BLITZ is very simple (although on the 128 there are several compiling options to choose between). Here's how to compile a program in three easy steps:

Choose compile options
On a 64 all you need to do is press **1** or **2** to specify disk or tape. On the 128 there are six options to choose from - but normally you will select option 1. The full range of options is explained in the next section.

Enter Program Name
Enter in full the name of the program you wish to compile, then press *RETURN*.

Compilation
Wait whilst BLITZ 64 loads the program, compiles it in two passes, and reports any errors it finds.

When compilation is complete the compiled program will be in memory (64) or on disk (128). If you are using a 64 don't forget to save the compiled program before you run it - just in case!

COMPILING OPTIONS ON THE 128

There are six different options:

1.	Single Drive	No COLLISION
2.	Dual Drive	RESUME with
3.	2 Single Drives	line number only
4.	Single Drive	RESUME and
5.	Dual Drive	COLLISION with
6.	2 Single Drives	full syntax

The first three options are for programs which do not use the COLLISION command, and which use RESUME followed by a constant line number, and not an expression. Programs compiled using options 1, 2 and 3 will run approximately 15% faster, and take up 20% less space than programs compiled under options 4, 5 and 6.

If you are using a dual disk drive such as the 4040, 8050, or 8250, select option 2 or 4. Place the disk containing the program(s) to be compiled in drive 0 (the source drive), and a disk for the compiled programs in drive 1 (the destination drive).

If you have two single drives select option 3 or 6. Enter the device numbers of the source and destination drives when prompted.

If you are using two drives BLITZ will prompt you to insert the source and destination disks - it is prudent to put a write protect tab on the source disk in case you get them the wrong way round! BLITZ will then ask whether you wish to format the destination disk. Whenever possible use a newly formatted disk to ensure that you have sufficient space for the compiled program and the files that BLITZ creates during compilation.

Having chosen the appropriate option, there are five different compiling *modes* to choose from on the 128. Mode 1 is selected automatically if you have a single drive - other modes require two drives.

MODE 1 - Single Program Compilation

Mode 1 compiles a chosen program from the source disk onto the destination disk. If you have a single drive system, the source and destination disk are one and the same so BLITZ prefixes the finished compiled program with **C/**. The compiler also generates a second file with the prefix **Z/** containing cross references to the line numbers in the original program for use when debugging.

During compilation, BLITZ uses two other files, prefixed **P/** and **D/**, which hold pure P-code and data. These files are scratched at the end of pass 2, but may be present on the destination disk if compilation is aborted for any reason.

MODE 2 - Batch Compilation

Mode 2 compiles all the programs on the source disk one at a time, and writes them to the destination disk. The run-time routines required to execute the P-code are incorporated in each program so that they can be run independently. This means that you can compile a batch of unrelated programs in one go.

To save time and disk space, no cross reference files are created, and so, if an error occurs at run-time, you should compile the offending program on its own using mode 1. No prefixes are added to the names of the compiled programs.

MODE 3 - Linking Programs; No Variable Passing

All programs on the source disk are compiled onto the destination disk, but the run-time routines are included in the first program only - this has the advantage that less disk storage is needed and loading times are shorter. No cross reference file is created, so mode 1 should be used to compile individual programs if run-time errors occur.

Only the first program can be loaded and run (this would normally be a menu program), but programs can be used to call each other - this is called *linking*. All variables and arrays are cleared when a new program is loaded, and if it is necessary to pass parameters between programs you should use mode 4 instead.

MODE 4 - Chaining; Full Variable Passing

Mode 4 works in the same way as mode 3, except that BLITZ inserts an additional pass at compile time which creates a list of all variables and arrays from every program module. This enables variables to be shared between programs - this is called *chaining*. BLITZ takes account of the length of the longest module, thus overcoming the problems usually encountered in passing variables to a longer BASIC program.

All arrays must be dimensioned in the program module that will reference them first - if this is not done then BLITZ will report a BAD SUBSCRIPT ERROR at compile time. The simplest solution is to dimension all arrays in the first program of the suite.

MODE 5 - Single Program; No Run-Time Routines

This mode is similar to mode 1, however, no run-time routines are incorporated in the compiled program. This mode can be used instead of mode 1 to create a cross-reference file to trace run-time errors. It can also be used to re-compile individual modules in a suite compiled under mode 3.

ERROR CORRECTION AND DEBUGGING

If you have incorrectly entered details about your disk drives, BLITZ may stop with a BASIC error message. If this happens, you should enter **RUN** to start again.

During compilation, BLITZ checks for SYNTAX, TYPE MISMATCH and UNDEFINED STATEMENT errors. Errors in the logic of the program will be detected only when you run the program, and these will be reported at run-time in the normal way. However, since the compiler discards all line numbers, the error is reported with a program counter value, *not* the original BASIC line number. On the 64 no cross-reference files are created, so the only way you can identify the source of the error is by rechecking your program. It is particularly important, therefore, to check your program thoroughly before compilation.

On the 128 the cross-reference file generated by modes 1 and 5 can be used to find the corresponding line in the original (uncompiled) BASIC program. Suppose you have run a program called **C/TEST** and the computer has replied with ILLEGAL QUANTITY ERROR IN 5912. In this case you would load the cross-reference file **Z/TEST** into memory and list the relevant section. A cross-reference file consists entirely of lines of the form *program counter=line number*, with one line (and one program counter value) for each line in the original program.

The program counter references individual BASIC statements, not entire lines, so there may not be a program counter value exactly matching that at which the error occurred. To find the line number where the error occurred list a range of lines in the cross-reference file, for example **LIST 5900-5912**. Look for the nearest program counter value less than or equal to the one in the error message and read the line number from there. For example, if the last line listed is **5909 = 620** then the error must have occurred somewhere in line 620 of the original BASIC program. This may seem complicated, but remember that compilation should be the last stage in program development. If your program is fully debugged, run-time errors should not occur.

BLITZ vs. COMMODORE BASIC

Although BLITZ is highly compatible with Commodore BASIC there are some minor differences (and improvements). You will find, however, that the vast majority of programs can be compiled without any alteration whatsoever.

Variables and arrays
Variables and arrays are stored in the same way as they are under the BASIC interpreter. However, they are positioned in the order they are encountered at compile time rather than the order they are encountered at execution time.

Integers
BLITZ uses true integer arithmetic whenever possible. This produces considerable improvements in execution time, since Commodore BASIC converts integers to floating point, does the calculations, then converts back to integers! It is therefore worth using integers where possible in order to save storage space and improve execution time.

Arrays
Dynamically dimensioned arrays are not allowed (this is the case with most compilers). For example, **DIM A(200)** is all right, but **X=200:DIM A(X)** is not. In addition, only single dimension arrays with a maximum of 11 elements (0..10) may be left undeclared. All multi-dimension arrays, and all arrays with more than 11 elements must be dimensioned by a DIM statement.

RUN/STOP key
BLITZ disables the RUN/STOP key after RUN, LOAD and CLR. There are two special REM statements which allow you to enable and disable the STOP key as you require:

REM ** SE	Enable *STOP* key
REM ** SA	Disable *STOP* key

STOP
The STOP command is treated as if it were an END.

CONT
You cannot use CONT to restart a compiled program.

FOR...NEXT
Commodore BASIC does not allow you to write FOR...NEXT loops using an integer as the control variable - but BLITZ does. For example, **FOR I% = 1 TO 10: A = A+1: NEXT I%** would not be valid in normal BASIC, but it is allowed by BLITZ. Integer loops are executed faster and use up less space on the BASIC stack.

LIST
LIST is not recognised by the compiler as it serves no useful purpose in a compiled program.

Machine code
The easiest way to combine a short machine code routine with a BASIC program is to include it as data statements which are read and poked into the appropriate area. Longer routines can be loaded separately by the main program, though the normal practice of using a variable as a flag to skip the LOAD statement the next time through will not work because the variable will be cleared. A simple alternative is to check whether the first one or two bytes of the routine are correct, for example:

```
10 IF PEEK(49152)<>76 THEN LOAD "CODE",8,1
20 SYS 49152
```

Extensions
BLITZ will automatically identify most extensions to standard Commodore BASIC and leave the relevant statements uncompiled. Then, at run-time, BLITZ will pass those statements to the BASIC interpreter for execution. Provided that the extension is installed (and enabled if necessary) the statement should be executed correctly. Extensions are indicated during compilation by the message ?EXTENSION. If you find that BLITZ does not recognise a particular extension, you can force the compiler to ignore the statement by preceeding it with two colons (::). Note that this will result in three colons where the statement is not the first on the line.

FURTHER INFORMATION

The high speed P-code produced by BLITZ will generally be about 60% of the size of the original program. There is, however, a 6k overhead (11k on the Commodore 128) of run-time routines incorporated in all programs compiled using modes 1 and 2, and in the first program compiled using mode 3 or 4.

During pass 1, the program is translated into P-code and checked for SYNTAX and TYPE MISMATCH errors. The numbers of erroneous lines are displayed as they are encountered - these lines must be corrected and the program recompiled. BAD SUBSCRIPT ERROR will be reported if you change the dimensions of an array within a program, but since it is legal to CLR variables and then re-dimension arrays that have previously been used this message should be considered a warning only. OVERFLOW ERROR will occur if a number greater than 1E38 is encountered - if you run a program with this error spurious results will occur. Extensions are indicated by the ?EXTENSION message as they are encountered.

At the end of pass 1, all multi-dimensional arrays are checked. If any of these have not been declared by a DIM statement, the message BAD SUBSCRIPT ERROR OF *arrayname* is displayed. Single dimensional arrays which have not been dimensioned default to 11 elements as they would in Commodore BASIC.

During pass 2, BLITZ replaces all variable and line references with their exact locations in memory; if a referenced line has not been found, the error UNDEFINED STATEMENT IN *line number* is displayed. At the end of pass 2 the Work files (prefixed **P/** and **D/**) are scratched, and compilation ends.

When mode 4 is selected, an additional pass (pass 0) is inserted to prepare a list of all the variables referenced by each program module and find the length of the longest module.

Programs compiled using BLITZ use less stack space. For example, each GOSUB or FOR...NEXT loop uses two bytes less, whilst integer FOR...NEXT loops save a further 7 bytes.

SPECIAL FEATURES OF BLITZ 128

High Resolution Graphics
Whenever BLITZ meets a GRAPHIC command during compilation it reserves 9k for the high resolution screen - at $1C00, so the program starts at $4000. Because a **GRAPHIC 5** command does not necessarily mean that you are using hi-res graphics, the compiler could be misled. However, if you include the directive **REM ** NG** in your program, BLITZ will drop the graphics routines from the the run-time library and start the compiled program from $1C00.

If you use **REM ** NG** and graphics commands together the compiled program will crash.

BASIC Input Buffer
BLITZ 128 has its own input buffer. The normal BASIC 7.0 buffer ($0200-$02A0) is used by BLITZ for bank switching routines which must not be overwritten. If you have SYS calls or extensions which need to use this area, you should mark them with the double colon (::) and use the **REM ** RI** (restore input buffer) directive at the beginning of the program.

Floating Point to Integer Conversion
The **REM ** FI** directive followed by a list of floating-point variables will instruct BLITZ to treat the variables in the list as integers wherever they are encountered. This is particularly useful for changing the control variables of FOR...NEXT loops. For example:

100 REM ** FI I,J,K

Variables defined as integer in this way must not appear in statements which contain extensions since they are not compiled.

No Extension Listing
The **REM ** NE** directive will suppress the ?EXTENSION warning message which might otherwise obscure more serious errors. The extension count will still appear at the end of compilation.

TRAP
The TRAP statement must not be used with variables or expressions, though it may appear alone or with a constant line number. Unlike the BASIC interpreter, BLITZ 128 turns TRAP off when a program is terminated by END or STOP, or when a program is loaded from within another.

RESUME
Compiler options 1, 2 and 3 support RESUME only when followed by a constant line number. This gives an approximate 15% speed gain and 20% size improvement compared to the full implementation supported by options 4, 5 and 6.

COLLISION
The COLLISION statement is supported only by options 4, 5 and 6. The constraints which apply to TRAP also apply to COLLISION.

GRAPHIC CLR
This is treated as if it were GRAPHIC 0. The 9k block of RAM reserved for high resolution graphics is not released.

LIST, DELETE, RENUMBER etc.
Commands of this type are not recognised by the compiler as there is no use for them in a compiled program.

BLITZ ON THE COMMODORE PET

BLITZ is available for 2001/3000, 4000 and 8000 series PET computers. To load BLITZ place the disk in drive 0 and type:

LOAD "*",8

On a 4000 or 8000 series computer you could alternatively hold down the *shift* key and press *RUN/STOP*.

You can use a single drive (option 1) or a twin drive (option 2). BLITZ for the PET does not allow you to use two single drives.

If you have an 8096 or 8296 computer there are four options in all - the first two will locate arrays immediately after the table of variables in the 32k of memory usually accessible to BASIC, but extending if necessary into the 64k expansion memory.

If you select option 3 or 4 then the array table is located in the expansion memory, starting at address $8000 (32768). This leaves a useful gap for any machine code routines which reside at the top of normal memory.

* * *

(c) SUPERSOFT 1986
Winchester House, Canning Road, Wealdstone,
Harrow, Middlesex HA3 7SJ Tel 01-861 1166