THE BASIC SPECTRUM COMPILER

BLAST

OXFORD COMPUTER SYSTEMS (SOFTWARE LTD.)

CONTENTS

INTRODUCTION

BLAST is the first fully compatible optimising BASIC compiler to be released for any Sinclair computer. Its principle aim is to provide the maximum possible execution speed for programs written in Spectrum BASIC without producing prohibitively large object programs. BLAST can increase the speed of BASIC by up to a factor of 40.
Issues of BLAST marked 4.0 (or above) or marked SUPER BLAST+ include an additional integer only compiler IBLAST for programs which do not use floating point maths. Programs compiled with IBLAST run considerably faster even than those compiled by BLAST.

WHAT IS A COMPILER?

In this section we explain the basic facts about compilers and introduce a small amount of terminology. An understanding of the basic concepts introduced here, although not essential, will greatly enhance your ability to exploit BLAST. The short section on terminology should be read and understood.

A BASIC program is simply a piece of text in which we specify the actions we want the computer to take when the program is `RUN`. The Spectrum's Z80 microprocessor understands only a language called machine code. To the Z80, a BASIC program is complete gibberish. In order to `RUN` a program we need some software which can understand BASIC and translate it into a form that the Z80 can understand. There are two types of BASIC translating program, the BASIC interpreter and the BASIC compiler.

1) Interpreters

A BASIC interpreter such as the one supplied in ROM as part of your Spectrum, reads each statement of the program and as it does so, performs the actions specified. Interpreters are very useful for program development because it is the actual BASIC text that is being interpreted. One can edit a program, run it and then re-edit rapidly and without fuss. The disadvantage of an interpreter is that programs run slowly. This is because most of the time, the interpreter is trying to make sense of the BASIC rather than carrying out the specified actions.

2) Compilers

Unlike an interpreter, a compiler translates a whole program into something that the machine can understand in one operation, called a compilation. When it has finished we are left with a block of machine code which is the translated version of the BASIC text. Compilers are much less useful for program development than interpreters since even the smallest change to the BASIC necessitates a complete re-compilation of the program. However, once a program has been compiled it will run at much greater speed.

TERMINOLOGY

In the remainder of this manual an understanding of the following terms will be assumed.

compile time	The time at which BLAST compiles a program
run time	The time at which the compiled program is actually executed.
source file	The input file to a compiler, in this case BASIC text, sometimes called the source code.
object file	The output from a compiler, in this case the machine code translation of the BASIC text, sometimes called the object code.
machine code	The internal language understood by the Spectrum's Z80 microprocessor.
p-code	An intermediate representation of a program somewhere between BASIC and machine code. P-code is an alternative to machine code which requires considerably less space. It does however require a mini interpreter at run time and is therefore slightly slower than machine code although much faster than interpreted BASIC. BLAST can compile programs into p-code or machine code or, into a combination of the two.
compiler directive	A message to the compiler which is added to the text of a source program and which affects the way the compiler behaves. BLAST features a number of useful compiler directives which are added to the program in the form of special `REM` statements.

PRODUCT DESCRIPTION

Your BLAST package should contain the following:

1) This BLAST manual
2) One cassette tape containing the following programs
   Side A: BLAST / COPIER
   Side B: TOOLKIT / IBLAST*

* IBLAST is only supplied with BLAST issue 4.0 or above.

GETTING STARTED

This section explains how to use BLAST in its simplest mode. We take a BASIC program already loaded into the computer and compile it directly into RAM without any tape or microdrive access. In this mode (called the RAM to RAM mode) we are limited to very short programs since the compiler, the source program and the object program all have to be in memory at the same time.

Load BLAST as follows:

`LOAD "BLAST"` <enter>

BLAST will autorun and ask the question `BACKUP TO MICRODRIVE (Y/N)`

If `Y` is selected, instructions for the microdrive backup will follow.

At this point BLAST makes a protection check in order to establish you as an authorised user of the software. We hope you will not find the procedure too tiresome. The protection sequence occurs once only when BLAST is first loaded. Once the check has been made, BLAST will allow you to compile as many programs as you wish without additional hassle.

On the inside cover of this manual or on a separate sheet will be found a matrix of coloured squares. Each square can be identified by means of a simple grid reference. For example to find square E-13, identify column E (marked along the bottom of the matrix) and row 13 (marked along the left hand edge of the matrix) square E-13 is where column E and row 13 meet. (See figure 1).

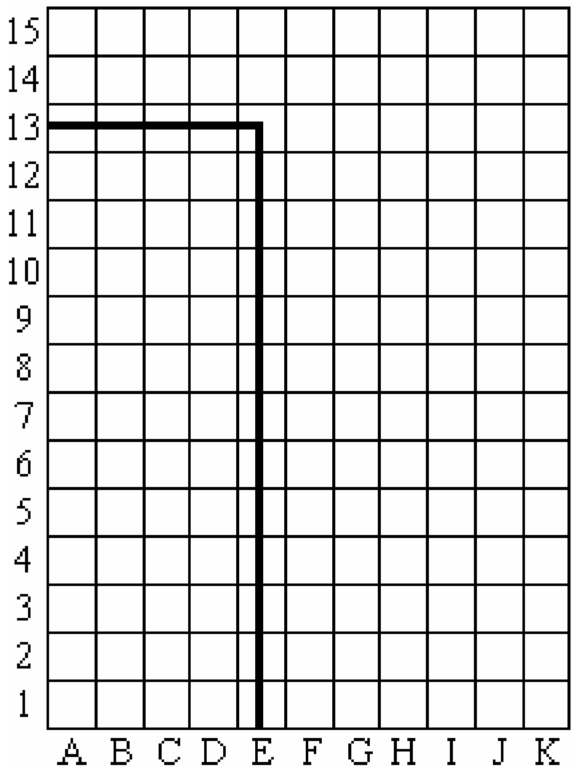

figure 1.

The protection check is very simple; all you have to do is correctly identify 4 squares and enter their colours. BLAST will give instructions as follows:

```
ENTER THE COLOUR IN SQUARE X-XX (W, Y, G, R)?
```

where X-XX is a grid reference. When you have found the square, enter one of the letters `W`, `Y`, `G` or `R` depending upon whether the square is white, yellow, green or red.

When you have correctly answered four such questions the protection check is complete.

BLAST is now fully initialised and ready to compile programs. From now on, until you type `NEW` or switch off the machine, your Spectrum will respond to a set of additional commands used to communicate with BLAST. The BLAST commands are preceded by an asterix (`*`) to distinguish them from the regular Spectrum commands.

First we shall use BLAST to compile a program already in memory and have the resulting object code stored also in memory. This is how BLAST behaves by default.

The purpose of this RAM to RAM mode is to demonstrate the use of BLAST. Since only about 2k of memory is available in this mode it can not be used to compile programs of more than a trivial size.

Type in or load a few lines of BASIC and type `*C` to compile.

Do not be alarmed at any gibberish which may appear on the screen; this is simply BLAST making the best use of the memory available. Assuming that there are no problems, after a minute or two control will be returned to you with the message:

```
(0) WARNINGS (0) ERRORS
```

To run the compiled version of your program, type `*R`.

While BLAST is in memory, you will be able to edit your source code, run under the interpreter or compile and run the BLASTED program as often as you wish. While working with BLAST, you will probably find that from time to time you wish to clear a BASIC program from memory without destroying BLAST. To do this, instead of typing `NEW` (which clears the whole of memory including BLAST) use `*N`. This will remove any BASIC text without affecting the compiler.

N.B. Although BLAST can cope with user written machine code called from a BASIC program, it can not do so when used in RAM to RAM mode. See the section on BLAST and user written machine code.

Since RAM to RAM mode is only intended as a demonstration, no method is provided of saving the compiled program. The same effect is easily achievable however by compiling RAM to TAPE (see below).

BLASTING LARGE PROGRAMS

So far we have used BLAST to compile from memory into memory. As explained earlier, this is only possible if the program to be compiled is small. To get round this problem, BLAST provides the option to read the source code from tape or microdrive and to write the object code out to either of these peripherals. Below we explain how to use BLAST in its various input/output modes.

Setting the input and output device

To set the device from which BLAST is to read the source code, select the `INPUT` option by typeing `*I` and answer the question

```
ACCEPT INPUT FROM: RAM, TAPE, MICRODRIVE
```

by typeing `B`, `T` or `M`.

To set the device to which BLAST is to write the object code, type `*O` and proceed as above.

At this point, BLAST will request information appropriate to the input/output options that have been chosen. For example, if microdrive has been selected BLAST will request a drive number and filename. If tape is selected BLAST will ask for a filename only.

The most useful combination of input and output devices are TAPE to TAPE and MICRODRIVE to MICRODRIVE. In both of these combinations, BLAST will be able to compile programs of any size.

If the output device selected is tape or microdrive, the compilation will end with the object program written to that device. Obviously the object program must be loaded before it can be run. Bear in mind that BLAST itself consumes a considerable portion of the Spectrum's memory. This does not prevent you from compiling large programs but it does mean that when you load a large BLASTED program from tape or microdrive, BLAST will have to be removed from memory first. This is done with the command `*Q`

Microdrive BLASTING

This is the best way to compile a large program. If you have large programs to compile and no microdrive, we suggest that you consider obtaining one.

The procedure is

1) Type `*I`, select `M` and give drive and filename details as requested.

2) Type `*O`, select `M` and give drive and filename details as requested.

When BLAST compiles to microdrive, 2 or 3 files are written, all of which are required in order to run the program. Their filenames are as follows: -

the object filename specified
the same name with .P appended
an optional .V file (see saved variables)

The compiled program is executed simply by loading the first of these (the user specified name) and typing `RUN`. Note that BLAST should be removed from memory before the object program is loaded.

Tape BLASTING

If you do not have a microdrive and you wish to compile a large program this is the way you will have to do it. Due to the limited nature of tape I/O, the program to be compiled must first be saved to tape in a special format. Facilities to do this are included in the toolkit supplied on the reverse side of the BLAST tape.

The procedure is

1) Load the toolkit (see The BLAST Toolkit)

2) Load the program to be compiled

3) Insert a blank tape and type `*B`

Once the source program has been saved on tape in the correct format, BLAST can be loaded and the program compiled.

Programs may be compiled TAPE to RAM. However this is only possible for very short programs. The most useful way of using BLAST and cassette is to compile TAPE to TAPE.

Tape to tape BLASTING

In this mode we use two tapes; a source tape and an object tape. The source tape contains your program in the special form suitable for compilation (see above), the object tape is blank.

We assume that you have selected tape for input and output. When you type `*C` to compile, BLAST will instruct you to insert the source tape and press play. After a short time the computer will beep and ask you to change tapes. You must stop the source tape within five seconds of the beep. If you do not do so it is likley that data will be lost.

After a short time you will be requested to change tapes again. Timeing is not so critical when the object tape is being swopped for the source tape; you are nevertheless advised to be alert during the whole process in order to minimise loading time of the final BLASTED program.

During the compilation you will be asked to swap tapes a number of times that depends upon the amount of BASIC being compiled. Finally the compilation will end with the usual error status report and the message `HIT ANY KEY`.

As soon as a key is struck, the computer will be re-set. To load the BLASTed program, rewind the object tape and `LOAD ""`.

Notes

All combinations of RAM, TAPE and MICRODRIVE are allowed by BLAST except for TAPE to MICRODRIVE.

Although in general BLAST, once loaded can be used to compile as many programs as necessary there are two restrictions:

1) If BLAST is used to compile a program with TAPE selected as the output device, the compiler must be re-loaded before it can be used again.

2) If BLAST is used to compile a program with MICRODRIVE selected as the output device, the compiler must be re-loaded before it can subsequently be used to compile a program with TAPE selected as the output device.

P-CODE AND MACHINE CODE

BLAST can compile programs either directly into Z80 machine code or into a compact pseudo machine code called p-code. The pros and cons of these two types of object code can be summarised as follows:

	p-code	**machine code**
SPEED	*faster than BASIC, slower than machine code.*	*Fastest possible.*
SIZE	*Smaller than BASIC. Smaller than machine code*	*Usually larger than BASIC. Always larger than p-code.*

The memory map for a BLASTED program is provided in Appendix 1. From this it can be seen that in addition to the object code and data, a BLASTED program also contains a block of code called the Run Time System (RTS). The RTS is essentially a library of subroutines which the object code calls for such purposes as multiplication, division and string handling. The RTS is always included in a BLASTED program and it imposes an overhead of about 5k. Because of the RTS, no BLASTED program can ever be less than 5k long. However, since p-code is about two thirds the size of the equivalent BASIC, large programs which are compiled into p-code can actually become smaller than the original BASIC. For example a 3k program, when BLASTED into p-code would occupy approximately 7k; 2/3 * 3k for the p-code and 5k for the RTS.
Similarly, a BASIC program 30k long would compile to about 25k. Obviously there is a break even point somewhere at which the two sizes are about the same. This occurs at approximatly 5k.
Of course the figures quoted here are necessarily very rough. Some types of program generate less p-code than others and a program which contains many comments will show a much more dramatic reduction in size than one which does not.
If BLAST is directed to generate machine code rather than p-code there will nearly always be an increase in size. Such programs will run slightly faster but this is of no use if the object code will not fit into the machine. Fortunately, BLAST can be directed to generate machine code for those sections of a program that are speed critical and p-code for the remainder. Very often, compiling quite a short section of BASIC into machine code and the rest into p-code can provide almost the same speed as the program would have had if it had been compiled entirely into machine code.

The type of object code that BLAST generates is specified by means of compiler directives. (See the section on this topic)
To instruct the compiler to generate p-code we write

```
REM! PCODE
```

and to instruct it to generate machine code,

```
REM! MACHINE CODE
```

BLAST generates p-code by default.

Caution

Whenever p-code and machine code are mixed within a single program, there is an important rule which must be observed:
There may be no jumps (`GOTO/GOSUB`) into the middle of a p-code section from a machine code section and there may be no jumps into the middle of a machine code section from a p-code section.
All `GOTO/GOSUB`'s from p-code into machine code must jump to the `REM! MACHINE CODE` directive and all `GOTO/GOSUB`'s from machine code into p-code must jump to the `REM! PCODE` directive.
Additionally, a p-code user defined function may not be called from machine code and visa versa.

BLAST AND USER WRITTEN MACHINE CODE

BLASTING a program which calls machine code subroutines should present no problems. BLAST has been designed for maximum compatability with Spectrum BASIC and this compatability extends to variable and program storage formats. In particular, the following practices commonly adopted by Spectrum users have been specifically provided for.

1) A BLASTED program may reserve space for machine code by lowering RAMTOP in the normal way.

2) BASIC variables are stored by BLAST in exactly the same way as they are in Spectrum BASIC. Consequently, machine code which picks these up and manipulates them will still work under BLAST.

3) Machine code routines which add extensions to BASIC by intercepting the Spectrum's error routine (or by most other methods) should still work. The explanation for this surprising fact is as follows: When at compile time, BLAST encounters an alleged statement that appears syntactically erroneous, the compiler will copy the offending text into the object file preceded by a special escape code. When at run time the RTS encounters this escape code, it will call the BASIC interpreter to handle it. If the text is a genuine syntax error, the interpreter will report the fact and exit in the normal way. If it is an extension to BASIC which has been provided for, the interpreter will behave just as it would had the program not been BLASTED.

Whenever BLAST encounters such a statement it will stop compiling with the message

```
WARNING HIT ANY KEY
```

As soon as a key is depressed compilation will continue.

BLAST's compiler directives are entered in the form of special `REM` statements which are recognised by BLAST at compile time. It is possible that in the future other commercial packages or indeed user written machine code will make use of the same technique to add extra commands to BASIC. For this reason a facility has been built into BLAST which allows `REM` statements to be passed to the interpreter if they begin with the escape character %. If BLAST encounters a `REM` statement beginning with this character, it will generate code which causes the `REM` statement, with the % stripped to, be passed to the interpreter at run time.

It is quite possible that certain obscure practices will cause problems. For example, machine code held in `REM` statements will certainly not work when the program is compiled because this method of storing routines depends on the way BASIC text is held in memory.

NB. Because of possible clashes between BLAST itself and user written machine code, the compiler will not allow programs calling Z80 routines to be compiled into RAM. Such programs should be compiled using tape or microdrive for output.

COMPATABILITY WITH SPECTRUM BASIC

BLAST has been designed for maximum compatability with Spectrum BASIC. This compatability extends not only to the language itself but also to the programming environment.
In BASIC, it is possible to stop a program while it is running, look at variables, execute statements and so on. The run can then be continued or re-started. This is all possible under BLAST except for one difference. The `CONTINUE` statement will not work on a BLASTED program.

Calling a BLASTED program as a subroutine from BASIC
Whenever a BLASTED program is resident in memory it can be run by executing the command `RANDOMIZE USR 23792`. There is nothing to prevent an ordinary BASIC program from co-residing with a BLASTED program and BASIC text may be entered or loaded in the normal way. Normally when a BLASTED program is executed it clears any variables in memory. This can be disabled with a `POKE 23803,1` or by using the `AUTORUN` directive. In order to make the BLASTED code return to a given line in BASIC, say line 1000 use the statement `REM % : GOTO 1000` in the BLASTED program.

AUTORUN AND SAVED VARIABLES

In order to save space, many BASIC programs are saved along with some or all of their variables. When compiling to MICRODRIVE BLAST copes with this using the `REM! AUTORUN` directive as follows: If the `AUTORUN` directive appears at the top of a program, BLAST will create a separate file (filename .V) containing any saved variables. When the BLASTED program is loaded, this file will be brought into memory automatically. If the `AUTORUN` directive is not included, BLAST will assume that there are no saved variables.
Unfortunately due to the limited nature of tape I/O, the method for dealing with saved variables when a program is compiled to tape is somewhat more complex. It is as follows.
Load the toolkit and the BASIC program to be compiled. Use the `*D` command to delete the program and then save the variables under a suitable filename as a normal program. Insert a line at the start of the program to be compiled to `MERGE` this file. When the BLASTED program is run it will merge these saved variables.
The Spectrum's operating system allows programs to be saved in such a way that they will run automatically from a specified line number when loaded. Such a facility is not directly available under BLAST since the BLAST equivalent (`REM! AUTORUN`) will always cause execution to begin at the first line of a program. The solution is simple. Add a `GOTO` statement to the beginning of the program (directly after the `REM! AUTORUN`) which will cause a jump to the desired line when the program runs. Re-save the program and compile.

COPYING BLASTED PROGRAMS

Microdrive copying

On the A side of the BLAST tape directly after BLAST is a utility for copying BLASTED programs from one microdrive to another. To use this program type `LOAD "COPIER"` and follow the instructions in the program.

Tape copying

Programs which have been compiled to tape may be copied using a tape to tape copier. Alternatively, as many copies as required may of course be made by compiling the program as many times as necessary.

ERRORS

1) Compile time errors

Although it is only possible to enter syntactically correct BASIC via the Spectrum editor, there are ways that incorrect code can be submitted to BLAST. The output from a program generator for example might contain errors and it is always possible that BASIC text may become corrupted on tape or microdrive! Additionally, it is perfectly possible for erroneous compiler directives to be entered by the user. For these reasons, BLAST rigorously checks the syntax of any text submitted to it.

However, things are not quite as simple as this. It may be that a statement which appears incorrect to BLAST while it is compiling, is in fact a perfectly proper extension to BASIC, possibly of the sort provided by certain commercial extended BASIC packages. Such extensions are perfectly permissible under BLAST, the problem is simply that at compile time, BLAST does not have sufficient information to distinguish them from genuine errors.

The solution adopted by BLAST is as follows. Whenever BLAST encounters a possible syntax error, it displays the offending text, issues a warning and waits for any key to be struck. Compilation then continues. If it turns out at run time that the questionable statement was in fact an error, the run will abort with the message:

```
NONSENSE IN BASIC
```

2) Run time errors

At run time, with a single exception, a BLASTED program will respond with errors such as `NUMBER TOO BIG` or `RETURN WITHOUT GOSUB` in exactly the same way as the interpreter. The exception concerns the error `SUBSCRIPT WRONG`. In order to avoid continual checking of array subscripts at run time, the BLAST RTS will ignore this error. If subscripts go out of range, the results will be unpredictable.

3) I/O Errors

If BLAST encounters a problem while attempting to read or write files to tape or microdrive, it will usually respond with the message `I/O ERROR`. This message can be displayed as the result of a number of different error conditions. For example the source file can not be found, the microdrive cartridge is full or the cartridge is corrupt. Certain types of I/O error are however undetectable and will simply cause faulty or aborted compilation. For a fuller explanation see TROUBLE-SHOOTING.

COMPILER DIRECTIVES

BLAST provides certain compilation options which can be invoked by means of compiler directives. These appear in special `REM` statements of the form
`REM!` <compiler directive>
That is to say, all compiler directives are preceeded by `REM!`. The shreak (`!`) provides an easy way for BLAST to tell whether or not to ignore the text following `REM`. There is one other kind of special `REM` statement recognised by BLAST; `REM%` causes the text of a comment to be passed to the interpreter at run time (see BLAST and user written machine code).
The compiler options available under BLAST are as follows:

Directive	**Meaning**
1) `REM! PCODE`	Have BLAST generate p-code until told otherwise. This is the default setting.
2) `REM! MACHINE CODE`	Have BLAST generate machine code until told otherwise.
3) `REM! AUTORUN`	Cause the object program to run automatically when loaded. This must be the first line of the program. `AUTORUN` programs when compiled to microdrive will include an additional saved variable file. (see AUTORUN AND SAVED VARIABLES)

TROUBLE-SHOOTING

If a program fails to compile or run apparently without cause, the first thing to try is to remove anything which is not strictly BASIC. All user written machine code and calls should be removed along with all compiler directives such as `REM! MACHINE CODE` and `REM! AUTORUN`.
Occasionally, BLAST may object to a line of BASIC which appears perfectly correct on inspection. If this happens, BLAST will stop with the message
`WARNING - HIT ANY KEY`. As soon as a key is pressed, BLAST will arrange for that statement alone to be passed to the interpreter at run time and then continue with the compilation. Although this event will be flagged by BLAST as a warning, it will make no difference to the running of the final compiled program. BLAST may also be forced to pass a line to the interpreter by inserting `REM%` at the beginning of the line. This can be useful if a particular line seems to be giving trouble when compiled. This facility can not be used with user defined functions, or `DATA` statements.
If a program fails to run successfully when compiled, the problem maybe that the source program contains saved variables. Methods for dealing with these are explained under AUTORUN AND SAVED VARIABLES.
Quite a frequent cause of program failure at run time is that the program contains `POKE` statements which corrupt the BLASTed program. Remember that short programs can increase in size when BLASTED and that what was once free RAM may be free no longer. Often programmers are not aware that their programs are POKEing where they shouldn't. If you wish to `POKE` do it above RAMTOP.

Very occasionally an extremely large BASIC program may cause BLAST to run out of space to hold all the program's variables. If this happens one of two messages will appear
1) `100 MANY VARS`
Solution: reduce the number of variables.
2) `NAMES TOO BIG`
This means that the space allocated to holding the text of variable names is exhausted.
Solution: reduce the length of some of the variable names.

Tape and Microdrive errors
BLAST makes extensive use of the Spectrum's I/O. Unfortunately this is not always reliable and may cause undetected compilation errors. It is for example possible for a microdrive which works correctly with many programs to fail when used with BLAST. If BLAST can detect the error it will respond with the message `I/O ERROR`. Often however the error is undetectable and the program will simply fail. If a program fails to compile or run correctly either to tape or to microdrive it is recommended that the compilation is attempted a second or even a third time.

Loading problems
In some circumstances, a program that has been compiled to tape or microdrive may fail to load correctly. Either the microdrive goes on whirring endlessly or the tape continues to the end with the computer still in loading mode. The probable reason for this malfunction is that there is insufficient room in memory for the object program. Try using the reset button on the side of the computer and attempting the load again. If this does not solve the problem it is possible that the object code will not fit into memory. This is unlikely to occur with a p-code file but is quite possible if the program has been compiled to machine code which tends to consume more memory than BASIC source code.
Another potential source of problems is the possibility that the source program is corrupt in some way. BLAST is as tolerant as possible of corrupt programs but there are certain types of corruption which will prevent a program from compiling even though it runs under the interpreter. Commercial programs which have been unprotected sometimes exhibit this problem. In particular, the system variables which give program and variable sizes must be accurate.

REDUCING LOADING TIMES
The COPIER (see copying BLASTED programs) is useful in another way:
Programs compiled microdrive to microdrive using a single cartridge can often be slow to load due to the format in which the code is stored. If the COPIER is used to copy a compiled program, the new copy will load more quickly than the original. Copied programs can of course themselves be re-copied.

COMMAND SUMMARY

The following commands are recognised by BLAST in its initialised state. All commands may be typed in either upper or lower case.

Remember that a `NEW` command will clear BLAST entirely from memory. If you wish merely to clear a BASIC program from memory, use `*N`.

1) `COMPILE` **syntax: `*C`**

Compile a BASIC program using the previously set input device (see `*I`) for the source code and the previously set output device (see `*O`) for the object code. The default setting for input and output is RAM.

2) `RUN` **syntax: `*R`**

Run a compiled program. The `*R` command is used only to run a compiled program that has been compiled into RAM. If tape or microdrive has been selected for output the object program should be loaded from this device and executed with `RUN`.

3) `INPUT` **syntax: `*I`**

Set the device from which BLAST is to read the source code for a compilation. BLAST will prompt with the message

```
ACCEPT INPUT FROM: RAM, TAPE, MICRODRIVE
```

for which the responce is `R`, `T` or `M`. The default device is RAM.

4) `OUTPUT` **syntax: `*O`**

Set the device to which BLAST is to write the object code for a compilation. BLAST will prompt with the message

```
ACCEPT OUTPUT FROM: RAM, TAPE, MICRODRIVE
```

for which the responce is `R, T` or `M`. The default device is RAM.

5) `QUIT` **syntax: `*Q`**

Quit BLAST and release the portion of memory used by BLAST for other code.

THE BLAST TOOLKIT

BLAST comes complete with a comprehensive toolkit designed to aid program development.

The TOOLKIT is the first program on side B of the cassette. To load the toolkit type

`LOAD "TOOLKIT"` <enter>

The TOOLKIT will autorun and sign on with the message

```
BLAST TOOLKIT (c) OCSS 1983
```

Like the compiler, the TOOLKIT loads into high RAM and sets RAMTOP below itself.
The TOOLKIT reduces user RAM by approximately 3.5k.
Note: The TOOLKIT cannot co-reside in RAM with the BLAST compiler.
The facilities available are listed below. Each function is executed by entering an asterix (`*`) followed by the single letter command given and the indicated parameters.
In the following n, n1 and n2 denote integers.
The part of the program over which a given command is to take effect is specified by a line range as follows.

nl-n2	means lines nl to n2 inclusive
nl-	means from line n1 to the end of the program
-n2	means from the beginning of the program to n2, including n2.

If a line range is omitted altogether, the toolkit assumes that the line range intended is the whole program.
A dot (`.`) may be used to stand for the current line.
If the line range is omitted from a toolkit command, any comma which would normally follow the line range should still be included.

LINE COMMANDS

1) `EDIT` **syntax: `*E n1`**

The line n1 is displayed for editing.

2) `COPY` **syntax: `*C n1,n2`**

Copy line n1 to line n2, over-writing any existing line.

3) `DELETE` **syntax: `*D n1`**

Delete line n1.

4) `MOVE` **syntax: `*M n1,n2`**

Move line nl to line n2, deleting line n1.

BLOCK COMMANDS

1) `COPY` **syntax: `*C <line_range>,n`**

Copy lines the line range to line n, over-writing any existing lines. The lines will be numbered consecutively from n.

2) `DELETE` **syntax: `*D <line_range>`**

Delete the line range. `*D` on its own will delete the entire program leaving all variables intact.

3) `MOVE` **syntax: `*M <line_range>,n`**

Move the line range to n, deleting the original lines.

4) `RENUMBER` **syntax: `*R <line_range>,nl,n2`**

Renumber the line range starting at n1 with step n2. The default for n2 is 10. Renumber will not renumber more than 1792 lines at a time. Larger programs can of course be renumbered in two or more stages.

STRING FUNCTIONS

1) `FIND` **syntax: `*F <line_range>,string`**

Search the line range for the first occurrence of the string.

2) `SEARCH AND REPLACE` **syntax: `*S <line_range>,stringl,string2`**

Search the line range for string1 and replace it with string2. The new line will be checked for syntax. If there is an error, the first line that contains the error will be displayed. Any previous lines in which the search and replace was successful will remain modified. Note that the replacement string may not be null.

`*F` & `*S` commands:

It is not directly possible to enter BASIC keywords into these commands. There is, however a trick which will solve this problem. When a keyword is required, first type `THEN` and after, the keyword. Now move the cursor back and delete the `THEN`.

OTHER COMMANDS

1) `TRACE` syntax: **`*T`**

When the program is run display the linenumber of the statement currently being executed. The space key can be used to slow down execution and `SHIFT O` to hold up execution.

Trace does not take a linenumber. `*T` will turn on the trace facility which will be operative thereafter during any program run; trace can be turned off with `*U`

2) `KILL` syntax: **`*K`**

Delete all `REM` statements in the program not beginning with `!` or `%`

3) `WRITE` syntax: **`*W <line_range>,<filename>`**

Save the line range to cassette under <filename> (maximum 10 characters).

4) `BLAST SAVE` syntax: **`*B <filename>`**

Save the program in a form suitable for compilation from tape by BLAST. The program will be saved in blocks together with the information BLAST needs to compile it.

The `*B` command (BLAST SAVE) is intended for use with cassette only. BLAST can compile a program which has been saved to microdrive in the normal way.

5) `VARS` syntax: **`*V`**

List various useful system variables including the amount of memory free.

6) `LIST` syntax: **`*L`**

List all BASIC variables together with their values.

7) `JOIN` syntax: **`*J <line number>`**

Join the indicated line to the next.

8) `GLOBAL ON/OFF` syntax: **`*A/*G`**

In normal operation the find and substitute functions will stop and wait for a key to be pressed after each find or substitution. To turn this feature off type `*G` and to turn it on again use `*A`

9) `QUIT` syntax: **`*Q`**

Quit the toolkit.

Re-initialising the toolkit

`RANDOMISE USR 61950` will reactivate the toolkit after a `*Q.`

Loading the toolkit while a program is in memory

Type the following:

```
CLEAR 61946
LOAD "tkcode" CODE
RANDOMIZE USR 61950
```

HIGH SPEED INTEGER ONLY COMPILATION

All issues of BLAST marked issue 4.0 or above include an additional high speed integer only compiler called IBLAST. This compiler is designed to produce much faster code than BLAST for programs that do not make use of floating point numbers. IBLAST is located on side B of the BLAST tape directly after the TOOLKIT. It is loaded and used in exactly the same way as BLAST.

What kind of programs can be compiled with IBLAST?

All numeric variables and arrays used in a program compiled with IBLAST are assumed to be of integer type. These variables can only hold signed whole numbers in the range -32768 to +32767. Additionally a compiler directive may be used to cause IBLAST to treat variables as unsigned integers in the range 0 to 65535.

If you are not sure whether or not a program will work under IBLAST the best solution is to try the experiment. However, bear in mind that even though a program may appear to compile successfully this does not necessarily mean that it will behave as expected at run time. For example a statement like `LET A = (50000 + 50000)/10` would require a partial result when the bracketed expression is evaluated of 100000 this is out of range and will evaluate incorrectly.

Because IBLAST can only deal with integers, there are certain operations which are no longer useful or that behave differently in an IBLASTed program. These are as follows:

1) The following functions will give a `RUN` time error if compiled with IBLAST `PI, SIN, COS, TAN, ASN, ACS ,ATN, LN, EXP, SQR` and exponentiation

2) No string contained in a `VAL` or `VAL$` argument may contain variable or array names. For example `VAL("A + 1")` will not work although `VAL(A$)` where `a$ = "123"` is perfectly allowable. Any attempt to access a variable in this way will give rise to a `VARIABLE NOT FOUND` error at run time.

3) Division (slash) will round to the nearest integer. e.g. `7/2 = 3`.

4) `BEEP, CIRCLE, DRAW` and `RND`

In Spectrum BASIC the above commands require fractional arguments. Because of this they work slightly differently under IBLAST. Both parameters of `BEEP`, the third parameter of `CIRCLE` and the (optional) third parameter of `DRAW` accept integers which are internally divided by 100 before being submitted to the graphics routine for execution. This means that these parameters should be 100 times larger than they would be in Spectrum BASIC in order to run correctly under IBLAST. The `RND` function returns a random integer within the full integer range.

The SIGNED and UNSIGNED directives

The compiler directive `REM! UNSIGNED` will cause IBLAST to treat all numeric objects as laying in the range 0 to 65535.

The compiler directive `REM! SIGNED` will cause IBLAST to treat all numeric objects as laying in the range -32768 to +32767.

The default is `SIGNED`.

Some notes on the treatment of integers.
It may be helpful in some cases for users to understand precisely what happens when IBLAST integers are manipulated. We first discuss the situation that obtains when the `UNSIGNED` directive is in force.
All numeric variables and arrays are held in a 16 bit word of memory. Such variables can hold numbers between 0 and 65535. Since IBLAST does not check for overflow, operations which cause variables to go out of range may give incorrect results. Let us consider the statement `LET A = 65530 + 10`. Since the expression evaluates to 65540, the answer will not be correct. What will actually happen is that the variable A will "wrap around" that is, as soon as it reaches 65535 and is incremented by one it will start over at zero. The result of the expression will therefore be 4. Equally, if `A =0` and is then decremented by one, the wrap around will occur the other way about and the result will be 65535.
When the `SIGNED` directive is in force (as it is by default) numbers obey 2’s compliment rules that is, any number N in the range 32768 <= N <= 65535 is treated as a negative number equal to N-65536. This allows certain operations such as comparisons to manipulate negative numbers correctly. Of course wrap around still occurs and should be avoided except when specifically required.
It should be noted that even in `UNSIGNED` mode negative constants are still allowed and will be converted to positive integers using the 2’s complement rule quoted above. Because of the wrap around rules, this will normally yield the correct result in a calculation. Any floating point constant will be rounded down to the nearest integer.
In the absence of the `REM! UNSIGNED` compiler directive, all but three operations will assume a signed format. The three exceptions are `PEEK, POKE` and `IN` where parameters are assumed to be unsigned; this allows `PEEK, POKE` and `IN` to access the whole of memory.

Compatibility with the interpreter
Unlike BLAST, some versions of IBLAST do not hold their variables in the same format as the interpreter. This has certain consequences.
1) No statement passed to the interpreter (see BLAST and user written machine code) may contain references to variables or array elements.
2) No Interface 1 command may contain references to variables or array elements.
3) So that these may be identified, IBLAST will still write a .V file containing any saved variables when compiling to microdrive using the `AUTORUN` directive. These will be ignored by the IBLASTed program.
4) Variables can not be `SAV`ed or `MERG`ed.
There is of course no reason why the value of variables may not be passed to interpreted statements using `POKE` and `PEEK`.

Errors
In order to allow IBLASTed programs to run at maximum speed, the IBLAST run time system does not perform as many run time checks as BLAST or interpreted BASIC. Programs should be checked carefully under the interpreter before being submitted to IBLAST.

APPENDIX 1

BLAST MEMORY MAP

```
P-RAMT    --------------------------------------
                  USER DEFINED GRAPHICS
UDG       --------------------------------------
                          BLAST
RAMTOP    --------------------------------------
                       GOSUB STACK
                      MACHINE STACK

                        spare RAM

STKEND    --------------------------------------
                     CALCULATOR STACK
STKBOT    --------------------------------------
                        WORKSPACE
WORKSP    --------------------------------------
                       EDITING AREA
E LINE    --------------------------------------
                     BASIC VARIABLES
VARS      --------------------------------------
                      BASIC PROGRAM
PROG      --------------------------------------
                   CHANNEL INFORMATION
CHANS     --------------------------------------
                     MICRODRIVE MAPS
               INTERFACE 1 SYSTEM VARIABLES
                     SYSTEM VARIABLES
                      PRINTER BUFFER
                        ATTRIBUTES
                       DISPLAY FILE
                           ROM
          --------------------------------------
```

RUN TIME MEMORY MAP

```
P-RAMT    --------------------------------------
                    USER DEFINED GRAPHICS
UDG       --------------------------------------
RAMTOP
                         GOSUB STACK
                        MACHINE STACK

                          spare RAM

STKEND    --------------------------------------
                      CALCULATOR STACK
STKBOT    --------------------------------------
                          WORKSPACE
WORKSP    --------------------------------------
                        EDITING AREA
E LINE    --------------------------------------
                     RUN TIME VARIABLES
VARS      --------------------------------------
                       BLASTED PROGRAM
PROG      --------------------------------------
                     CHANNEL INFORMATION
CHANS     --------------------------------------
                       MICRODRIVE MAPS
                       RUN-TIME SYSTEM
          --------------------------------------
                INTERFACE 1 SYSTEM VARIABLES
                      SYSTEM VARIABLES
                       PRINTER BUFFER
                         ATTRIBUTES
                        DISPLAY FILE
                             ROM
          --------------------------------------
```