HiSoft TurboBASIC

Amstrad BASIC Compiler for the *CPC464/664/6128*

HISOFT

High Quality Microcomputer Software

CONTENTS

	PAGE
Introduction	1
The Compilation Process	1
Getting Started With TurboBASIC	4
Saving compiled code	8
Program Development &Compilation - A Quick Guide	8
Compilation Errors	10
Runtime Errors	14
BASIC keywords, functions and statements accepted by TurboBASIC	15
Technical Information	16
Interpreted & Compiled Code	17
Using the Printer from compiled code	17
Passing Variables Between Compiled and Interpreted Code	18
Example Program - INVADERS.BAS	20
Listing of INVADERS.BAS	22

Introduction

Welcome to the world of fast programs and easy development given to you and your Amstrad CPC464, CPC664 or CPC6128 with HiSoft's TurboBASIC compiler. TurboBASIC is a totally new product which transforms your Amstrad BASIC programs in to pure Z80 machine code, making them run between 7 and 80 times faster!

TurboBASIC is easy to use and is small enough to allow most programs to be compiled. Before we start to use it we'll look in to the process of compilation to see why having a program such as TurboBASIC is so handy.

The Compilation Process

When you write a BASIC program on your Amstrad you're taking advantage of the machine's in-built *interactive features*. This means that you can write part of a program, run it, correct any errors, add to the program, run it etc. This can be done because the BASIC system used by the Amstrad machines is known as an 'interpreter'.

When a program is run, the BASIC interpreter's *syntax checker* looks at the BASIC in each line and determines if it makes sense or not. If a line does not make sense the syntax checker stops the interpreter and throws out a message to the screen, along with the offending line. This syntax checker is able only to detect what is known as a *syntax error*, which is a programming error stopping a line forming part of a BASIC program because the line does not conform to the *syntax rules* of the language, i.e. does not make any sense to the interpreter. For example, just as we cannot say in English:

Time four half past is the

and expect to be understood by anyone, we cannot assign a string to a numeric variable in Amstrad BASIC:

```
10 a = "This will cause a 'Type mismatch' syntax error"
```

and expect the interpreter to understand what we want.

The syntax checker cannot detect *algorithmic* errors, which are programming errors that although perfectly legal BASIC, do not really do what you expect or require. You may have a line which adds 10 to a variable, for instance, when in fact you are supposed to be adding 5. Errors of this type are always the hardest to find!

Once you have a valid BASIC program in the Amstrad's memory you may execute it by typing `RUN` and pressing the `RETURN` or `ENTER` key. Errors can still occur, of course, if you miss out lines or perform incorrect calculations. The program will then stop with an *error report,* telling you at which line the error occurred. You can then list the rogue line, correct the error (hopefully) and run the program again. This is why the system is known as *interactive* as it allows you to correct errors as they occur rather than having to run the program, find all the errors and correct them all before running the program again.

Interaction is a feature of interpreters; when you `RUN` a program the interpreter finds the first line of the program, which is stored in the computer as BASIC, and works out what it is supposed to do with the line. This may involve continuing on to other lines, it may involve calling subroutines or it may just print something on the screen and finish. The interpreter then either reports an error and allows you to interact again or executes the instructions in the line and then it moves on to examine the next one.

Interpreted programs are automatically much slower than the computer truly is capable of because each individual line has to be examined by the interpreter program every time it is met. A `FOR..NEXT` loop is not automatically converted in to a machine code loop which ends after a certain number of iterations, but is examined each time the line containing the `NEXT` is encountered and a check is made to see if the end of the loop has been reached.

The Amstrad BASIC interpreter lives in the ROM (read-only memory) inside the machine and cannot easily be removed. TurboBASIC supplants this interpreter with a program known as a *compiler.*

A compiler is a program which converts a *source* programming language code in to an *object code*. In the case of TurboBASIC the source code is Amstrad Locomotive BASIC and the object code is pure Z80 machine code.

A compiler which converts a language in to a computer's *native* language is particularly useful because computers - or the microprocessors inside them - can only understand their own language (machine code) and therefore we obtain much faster-running programs if we convert the whole program to native code and *then* give it to the computer rather than convert each BASIC line into native code *every* time we run the BASIC program.

Thus one main advantage of using a compiler is the increase in speed with which our programs can run when compiled to machine code.

Other good reasons for using compilers are that it removes the need for us to load an interpreter (no problem in the case of the Amstrad, of course, as the interpreter is in ROM) and the object code which we run usually is smaller than the source code we started off with. Smaller source code usually means that programs load from disc or tape faster, too, so having a compiler is much more fun than being stuck with an interpreter.

The TurboBASIC compiler produces machine code which does exactly what the original BASIC source program does, but it does it considerably faster. The object code produced by TurboBASIC is also its executable cod~ and may be stored in memory or saved on disc as a normal AMSDOS '.BIN' file.

As it is now in machine code the program cannot be run simply by typing RUN, while it is in memory, as this is a BASIC interpreter command rather than a native machine one. If the object code has been left in memory rather than sent to a file it can be started using the BASIC interpreter's CALL statement, using the address shown by TurboBASIC as the start of the program. If this address were 26121, for example, we could start the program by typing

```
CALL 26121
```

and pressing RETURN or ENTER.

Typing in CALL followed by an address often is inconvenient and it's certainly not as tidy as using the RUN command of the built-in BASIC interpreter, so TurboBASIC adds a new *external command* to the system: |RUN. An external command is one which is prefixed by the | symbol, such as |CPM or |ERA. TurboBASIC actually adds three external commands, |RUN, |MAKE and |COMPILE. We'll look at the purpose and usage of |MAKE and |COMPILE later; the |RUN command causes the program just compiled by TurboBASIC to be run, exactly as if we had typed the requisite CALL statement. Only one of these commands should be used on one line.

In those circumstances where we have written the compiled code out to a file (we'll see how later) we are left with a normal Amstrad .BIN file on cassette or floppy disc. This may be run in the usual way by typing

```
RUN"filename [ENTER]
```

where filename is the name under which we saved the program. There is no need for the TurboBASIC compiler to be in memory for programs created by it to be run later; this allows you to sell programs compiled by TurboBASIC without having to sell TurboBASIC as well!

There are a very few BASIC features which TurboBASIC cannot handle but we needn't consider them until later. To see how the whole system works, let's start from the beginning.

Getting Started With TurboBASIC

The TurboBASIC compiler is supplied on cassette tape or 3" floppy disc, both of which incorporate a short BASIC loader to get the compiler in to memory and initialise it. If you have the floppy disc version we recommend that you back it up straightaway using one of the utilities supplied by Amsoft with CP/M.

The cassette version has a fast loading version and a slow loading version on the other side. Try the fast side first; if this will not load on your recorder don't worry - use the slower side instead. If you cannot get this to load please return the cassette to us for replacement.

To load the cassette version, ensure that the tape is fully rewound and then place it in your cassette player ready to be loaded. Type

```
RUN"
```

and press ENTER followed by the PLAY button of your cassette player. The loading process takes a few minutes.

To load the disc version place the disc in the drive and type

```
RUN"TURBOBAS
```

followed by ENTER or RETURN.

Both versions clear the screen and produce a title sign-on message. Shortly you will be asked where you would like the compiler to be loaded. For the minute, press RETURN by itself as we don't know any other answers to this question yet. The loading process continues until the compiler is safely in memory.

When the load is complete, the BASIC interpreter's READY prompt re-appears; to all intents and purposes nothing has changed. This isn't really true, of course. If you ask how much free memory there is, by typing

```
PRINT FRE(0) [ENTER]
```

the answer will be smaller than previously as the compiler has grabbed a chunk for itself. Also, the three external commands |MAKE, |COMPILE and |RUN have been installed, but DON'T try them out just yet!

The interpreter's still there, so we now have a machine which contains both a BASIC interpreter and a BASIC compiler. Since they are co-residing we often can take advantage of features of each. For example, while developing programs we may test them using the interactive features of the interpreter and still run the compiled versions! Let's do that now. Type

```
AUTO
```

and press RETURN or ENTER.

The built-in interpreter produces a line number, 10, for us and awaits our input. Anything we type now becomes part of a BASIC program for the interpreter but because the TurboBASIC compiler also is present, we can compile the same program at any stage!

Type in the program below, using the ESC key to stop the sequence of line numbers generated by AUTO once all the lines have been entered.

```
10 N%=1
20 DIM FLAGS%(7000)
30 t=TIME
40 PRINT N%;" iteration(s)..."
50 FOR M%=1 TO N%
60 COUNT%=0
70 FOR I%=0 TO 6199
80 FLAGS%(I%)=1
90 NEXT I%
100 FOR I%=0 TO 6199
110 IF FLAGS%(I%)=0 GOTO 190
120 PRIME%=I%+I%+3
130 K%=I%+PRIME%
140 WHILE K% <= 6199
150 FLAGS%(K%)=0
160 K%=K%+PRIME%
170 WEND
180 COUNT%=COUNT%+1
190 NEXT I%
200 NEXT M%
210 PRINT COUNT%;"primes in";(TIME-t)/3;"1/100th seconds"
220 INPUT "Press [ENTER] to return to system...";Z$
230 END
[ESC]
```

This is a BASIC version of the famous BYTE Sieve benchmark program which calculates prime numbers up to 7000. If you run this program from the interpreter, by typing

```
RUN
```

and pressing RETURN or ENTER, you'll find that it takes about 94 seconds (on a CPC6128). Now we can compile it and see if there's a difference in execution speeds.

To compile this program, type:

```
|MAKE
```

and pressing RETURN or ENTER. Notice that when you execute the |MAKE command the compiler prints a sign-on message on the screen and prints the message

```
Compiling...
```

and proceeds to do just that. |MAKE is in fact a special case of the more general |COMPILE command but we need not be worried about that yet. If there are any errors in the code it is trying to compile then the compiler will report the fact by displaying an error message on the screen and returning to the interpreter. If the program has been typed in as shown, has executed correctly under the interpreter then it will compile without error, producing a message saying where the code starts and how long it is. Both the numbers displayed are in decimal.

To run our compiled program we can take advantage of TurboBASIC's added external command, |RUN. So, type

```
 |RUN
```

and press the RETURN or ENTER key. The compiled code will run, producing the same output as the interpreted version but rather faster. Our timings show that the compiled code runs about 37 times faster than the interpreted version - not the largest speed increase which TurboBASIC can offer, but indicative of things to come.

Saving compiled code

Although the `|MAKE` command compiled a program in memory and produced its machine code in memory with the `|COMPILE` command, TurboBASIC is fully capable of compiling programs stored on disc and cassette and also of saving the code it generates on to disc or cassette. Programs stored in this way may then be run as normal machine code programs using the interpreter's `RUN"filename` command in the usual way.

To try out the `|COMPILE` command type:

```
|COMPILE
```

fiollowed by `RETURN` or `ENTER`. You will then be asked a series of questions. If you type `RETURN` or `ENTER` in reponse to each one the compiler uses its built-in 'default' values. This is exactly equivalent to the `|MAKE` command.

To make the compiler save the code to a file on floppy disc or cassette, simply type a file name when asked

```
Output file ?
```

by the compiler's `|COMPILE` command. Don't forget to press `RETURN`! So long as the file name you type conforms to the normal Amstrad file name conventions everything will go according to plan. Disc files you create will have an extension of `.BIN` (unless you specify otherwise, which isn't normally wise) so that AMSDOS can recognise them as binary (machine code) files ready to be executed when you `RUN"` them.

Program Development & Compilation - A Quick Guide

Programs may be developed in the interactive interpreter environment with the compiler loaded until the combined program and variable size exceeds about 8K. From that point on the program must be compiled from tape or disc. As BASIC programmers tend to make efficient use of available memory HiSoft has made TurboBASIC relocatable in order to allow the programmer to maintain his/her absolute addressed data areas. Relocation takes place after the compiler has been loaded, as follows:

Load the compiler using `RUN"` or `RUN"TURBOBAS` as appropriate. This places the compiler at its load address, which is NOT the same as its execution address. The loader program asks for a run address and presents a minimum and maximum value. Pressing `RETURN` or `ENTER` alone uses the default run address of 7000 (decimal) and relocates the compiler to that address.

If a lower address is specified and if this address is within range then the compiler is moved there. A low address allows a larger compiled program to be generated but reduces the space available to co-resident and interactively developed interpreted programs.

Entering a higher address results in the reverse situation where there is less space available to the compiler but more to the interpreter and its programs.

Initially the default run address should be used, but as programs are developed which require a higher or lower ratio of compiled to interpreted code the run address should progressively be altered to allow the use of interactive features for as long as is possible. Naturally, once the program and its variables get beyond about 8K it can no longer be developed with the compiler in memory. If you wish to compile the largest possible program then you should use as low an address as possible.

When a compilation is invoked with the new `|COMPILE` external command, the programmer is asked a series of questions. The first is

```
 Input file ?
```

and can be ignored (by pressing `RETURN` or `ENTER`) to compile the program currently resident in memory. Otherwise a valid filename may be entered; this will be opened and taken as the source of input for the compiler. If the file does not exist or is bad in some way, the compiler will produce an error message when all the questions have been answered. This program must be a `.BAS` file as TurboBASIC only compiles tokenised BASIC. TurboBASIC is a two pass compiler and so the familar

```
Press PLAY then any key:
```

will appear twice and the second time you should rewind the tape before hitting a key.

The next question concerns the output file:

```
 Output file ?
```

and this also may be ignored (by pressing `RETURN` to `ENTER`) if the code is to be compiled in to memory. If a valid filename is entered then it is used as the destination for the compiled binary object code. Again, an error is generated once all the questions have been answered if the file cannot be created or is otherwise bad. If you use both an `Input file` and `Output file` when using cassette things can get very boring because of the tape swapping involved.

The third question concerns the usage of events. If you require events (basically, interrupts and software 'exceptions') to be recognised then press `RETURN` or `ENTER` in response to this question, as the default reply is `Yes`. If events are enabled then the `ESC` key may be used to break in to running compiled programs in the normal way. Remember that if events are disabled (by typing `N` or `No` in response to the question) then `ESC` is not recognised and the `AFTER` and `EVERY` keywords are ignored.

The final question asked by the compiler concerns the maximum string length. This defaults to 255 on the compiler and this value is selected by pressing `RETURN` or `ENTER` in response to the question. Other values below 255 may be entered if the program does not contain - ***and will not generate*** - strings longer than the entered value. A lower value saves memory space as each string is allocated its maximum space at compile time. For many programs a value of 32 is appropriate.

Compilation Errors

When TurboBASIC compiles a program it will from time to time encounter syntax errors, compiler restriction errors and various other errors. Whenever it finds one of these, it prints an error message along with the line and statement numbers on to the screen and stops compilation immediately, returning control to the BASIC interpreter.

These are the messages which TurboBASIC can produce, along with the situations which can cause them:

`Array exists`

An array is being used in a `DIM` statement despite having been dimensioned already.

`Array not dimensioned`

The program contains a reference to an array prior to the dimensioning of that array in a `DIM` statement.

`Bad subscript`

This error occurs if the subscript of an array evaluates to less than 0 or greater than the dimensioned size. Please note that subscripts are NOT checked at runtime!

`Broken in`

This message is produced when the `ESC` key is pressed during cassette or disc input and output.

`Can't match NEXT`

A `NEXT` statement refers to a variable which has not bee used in the corresponding `FOR` statement.

`Can't match WHILE`

This message should not normally occur.

`Can't write to output file`

An error was reported by the system as it was writing the compiled code to disc or cassette.

`Expression too complex`

This error message means that an expression is too complicated to evaluate. Simplify the relevant expression by using parentheses.

`Function not defined`

A user-defined function is called in a program but there is no preceding definition of the function.

`Function not supported`

A Locomotive BASIC function which TurboBASIC does not support has been used. Please see below for the small list of functions and statements which TurboBASIC programs may not use.

`I/O error. Can't open input file`

The name you specified for the input file is illegal as a filename, the file doesn't exist or some other file error occurred.

`I/O error. Can't open output file`

The name you specified for the output file is illegal as a filename, there is no directory space or some other file error occurred.

`IFs too deeply nested`

This error is generated when the compiler's internal limit on `IF` statement nesting is encountered.

`Illegal code generated`

This is an internal compiler error and should never occur; please contact HiSoft with full documentary evidence if one of your programs generates this error.

`Insufficient room for Object code`

There is not enough memory available to the compiler to allow it to generate any more object (compiled) code.

`Invalid tree node`

This is an internal compiler error and should never occur; please contact HiSoft with full documentary evidence if one of your programs generates this error.

`Line does not exist`

The destination of a `GOTO`, `GOSUB` or another line-referencing statement does not exist.

`Missing bracket`

An expression or parameter list has an unbalanced number of brackets.

`Missing NEXT`

A `FOR .. NEXT` loop is being processed but there is no `NEXT` statement to match the earlier `FOR`.

`Multiple function definition`

A user-defined function definition using `DEF FN` occurs more than once.

`Must RESTORE to DATA line`

The statement on the line referenced in a `RESTORE` statement must be a `DATA` statement.

`Real numbers are not supported`

Sorry! TurboBASIC does not allow floating point numbers in the programs it is to compile.

`Statement not supported`

TurboBASIC does not support the BASIC statement used.

`String too long`

A string has been generated, accessed or otherwise manipul ted which is longer than 255 characters.

`Syntax error`

Whenever something is encountered which TurboBASIC cannot understand, it produces this error message.

`Term missing`

Part of a statement containing an expression has been omitted. For example, `WHILE` alone in a statement will cause the error but `WHILE 1` won't.

`Too many nested FOR-NEXT loops`

This error is generated when the compiler's internal limit on `FOR .. NEXT` loop nesting is encountered.

`Type mismatch`

A variable has been assigned a value of another type; for example, a string has been assigned to an integer variable. This error also may be caused if an operation or function call is not valid for the type in question.

`Unexpected ELSE`

An `ELSE` statement has been encountered in an inappropriate place; there is no matching `IF`.

`Unexpected NEXT`

A `NEXT` statement has been encountered outside of a `FOR .. NEXT` loop, or a `NEXT` occurs in an inappropriate place.

`Unexpected WEND`

If a `WEND` is encountered by the compiler before a `WHILE`, or in a context in which a `WEND` statement is invalid, this error is generated.

`Wrong number of subscripts`

An array has been referenced with a different number of dimensions to that used in the array's `DIM` statement.

Runtime errors

The compiled code generated by the compiler is less likely to encounter errors than the interpreter because most of the programming errors and all of the syntax errors will be trapped at compile time. However, there are a couple of error messages which can occur:

`Division by zero`

An expression has caused a division by zero error to occur.

`Can't find external command`

A program has used a '|' external command but the system has returned an error saying that it cannot locate the command

BASIC keywords, functions and statements accepted by TurboBASIC

ABS	AFTER	ASC	BIN$	BORDER	CALL
CHR$	CLG	CLS	DATA	DEF FN	DI
DIM	DRAW	DRAWR	EI	END	ENT
ENV	EVERY	FOR	FRAME	GOSUB	GOTO
HEX$	IF	INK	INKEY	INKEY$	INP
INPUT	INSTR	JOY	KEY	KEY DEF	LEFT$
LEN	LET	LINE	INPUT	LOCATE	LOWER$
MAX	MID$	MIN	MODE	MOVE	MOVER
NEXT	ON GOTO	ON GOSUB	ON BREAK GOSUB		
ON BREAK STOP		ON SQ	ORIGIN	OUT	PAPER
PEEK	PEN	PLOT	PLOTR	POKE	POS
PRINT	RANDOMIZE	READ	RELEASE	REM	REMAIN
RESTORE	RETURN	RIGHT$	RND	RUN	SGN
SOUND	SPACE$	SPC	SPEED INK	TAB	TAG
TAG OFF	TEST	TESTR	TIME	UPPER$	VAL
VPOS	WEND	WHILE	WINDOW	WINDOW SWAP	
XPOS	YPOS	ZONE			

As we have said in various places throughout this manual, TurboBASIC does NOT support floating point numbers; it deals with integers, which here are whole numbers between -32768 and 32767. Also, there are some functions and statements which the TurboBASIC compiler does not support. These are:

Tape, disc and printer handling in compiled code
Mainly because they wouldn't be any faster; however the printer may be accessed using the requisite firmware calls - see below for an example.

HIMEM, MEMORY and FRE
Not really useful in a compiler system.

ON ERROR
No non-fatal runtime errors. Extra command added in BASIC 1.1. Not universally applicable.

Also there are a few of differences and restrictions on some aspects of the system:

`DIM` must be used for each array and must textually precede the first use of the array.

`GOTO, GOSUB` etc. must not reference `DATA` statements.

`INPUT LINE` can have only one variable per line.

`PRINT` supports the usual separators but does not allow `USING`.

`RND` returns a 15-bit integer between 0 and 32767; see the example `INVADERS` program as an example of its use.

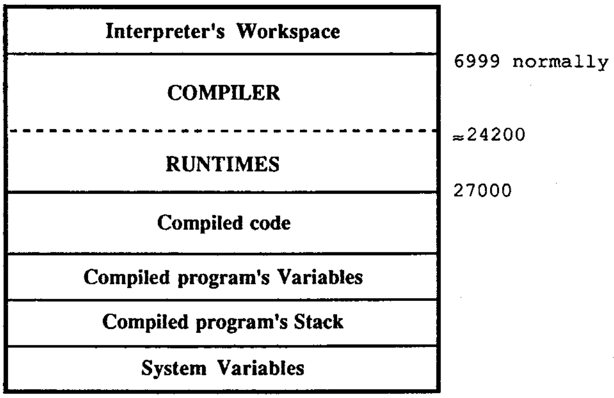
`TIME` also returns a 15-bit integer between 0 and 32767. When the time reaches 32767 it goes back to 0 again.

`RANDOMIZE` must be used in conjunction with a parameter.

Strings are not dynamic; the length defaults to 255 characters but this may be altered at compile time when using the `|COMPILE` command.

Technical Information

Memory Map



Interpreted & Compiled Code

To use interpreted and compiled code together compile the subroutines to be used together as one program, preceded by a line such as

```
ON PEEK(addr) GOTO a,b,c ...
```

where a, b, c and so on are the line numbers of each subroutine in the compiled section. By poking a value in to the `addr` address and using `|RUN`, the relevant compiled subroutine will be executed. `addr` is an arbitrary address which is decided upon by the programmer. Ensure that it does not conflict with the system, the compiler and the interpreter.

Variable values may be passed to and from the compiled code using `PEEK` and `POKE`. The technique is demonstrated in a simple example program in the next-but-one section.

Using the Printer from Compiled Code

The following assembly language program calls the firmware to output a string to the printer

```
          ; print a string to the printer
DD6E00                  ld    l,(ix)      ;hl -string descriptor
DD6601                  ld    h,(ix+1)
46                      ld    b,(hl)      ;b - string length
23                      inc   hl
5E                      ld    e,(hl)      ;de - string address
23                      inc   hl
56                      ld    d,(hl)
04                      inc   b
05          loop        dec   b           ;finished?
C8                      ret   z
1A          loop2       ld    a,(de)      ;next character
CD2BBD                  call  #bd2b
30FA                    jr    nc,loop2    ;loop until not busy
23                      inc   de          ;bump to next
18F5                    jr    loop
```

This can then be loaded into memory incorporated into a BASIC program as shown over the page:

```
10 pr=6900
20 FOR I=pr TO pr+22
30 READ A: POKE I,A
50 NEXT I
55 a$="hello there printer"+CHR$(13)+CHR$(10)
60 CALL pr,@a$
70 DATA &DD,&6E,0,&DD,&66,1,&46,&23,&5E,&23
80 DATA &56,4,5,&C8,&1A,&CD,&2B,&BD,&30,&FA,&13,&18,&F5
```

before we run or compile this we need to lower `HIMEM` using

```
MEMORY 6899 [ENTER]
```

so that BASIC does not use the area that we are using to store our machine code. Now compile and run this program using:

```
|MAKE [ENTER]
|RUN [ENTER]
```

After we have run the BASIC program we can print `a$ by:`

```
CALL pr,@a$ [ENTER]
```

Note that you can use `CALL pr,"HELLO THERE"` on later machines but not on the CPC 464.

Passing Variables Between Compiled and Interpreted Code

The short programs below show how easy it is to transfer data between compiled and interpreted code modules. Although it requires a bit of `PEEK`ing and `POKE`ing the technique is straightforward and easy to follow.

To enable the demonstration to be most effective we need to generate a file on cassette or disc containing some data. The short program on the next page will do this for us:

```
10 OPENOUT "DATAFILE"
20 FOR I=1 TO 100
30 WRITE #9,I*I
40 NEXT I
50 CLOSEOUT
```

As you can see, it writes the squares of the numbers between 1 and 100 to a file called 'DATAFILE', which it creates. This program must be run from the interpreter as it incorporates file handling commands.

Now we must write an interpreted program which opens this file and reads the data in, sending it to a compiled program. Here we go:

```
 10 MEMORY 6990
 20 LOAD"compile.bin"
 30 DIM A%(100)
 40 OPENIN"DATAFILE"
 50 FOR I=1 TO 100
 60 INPUT #9,A%(I)
 70 NEXT I
 80 CLOSEIN
 90 POKE 6998,(@a%(1)) MOD 256
100 POKE 6999,(@a%(1)) \ 256
110 CALL 24133
```

This program uses the two bytes at addresses 6998 and 6999 to pass the information. The addresses are protected from corruption by the system using the interpreter's `MEMORY` command. Two bytes are used because a Z80 memory address occupies 16 bits. The file is read in to an array called `A%` - an integer array.

When the entire data file has been read in to the array, the compiled code is called. As the compiler does not need to be in memory when we run this program, we can't use the `|RUN` command; if the compiler isn't there it won't be recognised. So we use the `CALL` command, calling the address which experience tells us is the address at which the compiler places the code by default. This may change in subsequent releases of the compiler.

If we relocated the compiler to anywhere but the default location, ***this address will be wrong.***

The compiled code reads the data from the array using this program:

```
10 DIM a%(100)
20 b=PEEK(6998)+PEEK(6999)*256-2
30 FOR i=1 TO 100
40 a%(i)=PEEK(b+i*2)+PEEK(b+i*2+1)*256
50 NEXT i
60 FOR i=1 TO 100:PRINT a%(i);:NEXT i
70 STOP
```

As you can see the program grabs the address of the array from 6998 and 6999 and then uses this address to get each integer element from the array.

The values read in are printed out as confirmation that the entire process has worked correctly.

Naturally this technique may be extended for complicated programs which need to pass large amounts of data or large data structures.

We hope that you enjoy using the TurboBASIC compiler system and welcome any comments you may have for further improvements and modifications.

Example Program - INVADERS.BAS

Supplied on the cassette tape or disc along with the compiler is a program called `INVADERS.BAS` which is a Space Invaders type game. Like many games type programs it uses the `RND` function to give a random response. As this gives a random number between 0.0 and 1.0 with the interpreter and as the compiler does not support fractions RND is different when compiled; instead it gives a random number between 0 and 32767. Clearly we have to change some of our program if we switch from using the interpreter to the compiler and vice versa. However by using a user-defined function we can arrange things so that we only need to modify one line.

In the INVADERS program listed below this is line 20. As listed

```
20 DEF FNR=RND
```

this works with the compiler. To make it work with the interpreter use

```
20 DEF FNR=RND * 32768
```

To produce the compiled version, load and invoke TurboBASIC in the normal way. Use the |COMPILE command and in response to the 'Input file' prompt type

```
 INVADERS.BAS
```

and press RETURN or ENTER. When asked for an output file name, type RETURN or ENTER. When asked about events hit RETURN but when asked for the string size give the reply 32. After a few seconds, if using disc, the compiled code will be generated. Then use |RUN to run it. Obviously compiling from cassette takes a little longer but the finished program will run just as fast. To play the game use Z and X to move left and right respectively and RETURN to fire.

If you want to see how slowly the program runs using the interpreter, load it with LOAD"INVADERS" and change line 20 as described above. We find it just about unplayable.

For reference purposes, there is a listing of INVADERS.BAS on the following pages.

Listing of INVADERS.BAS

```
10 RANDOMIZE 100
20 DEF FNR=RND
30 MODE 2
40 SYMBOL AFTER 240
50 EVERY 5 GOSUB 1690
60 DIM x(5,10),y(5,10),screen(80,25),bombx(20),bomby(20)
70 DIM gun$(2)
80 DEF
FNlook$(x)=CHR$(screen(x-1,25))+CHR$(screen(x,25))+CHR$(sc
reen(x+1,25))
90 DEF FNgun$(p)=gun$(2+(p<>0))
100 ship1=242:ship2=243:sx=0:sdir=1
110 near=2:xdir=1:gx=40:fx=0:dum=500:alive=3:top=1:score=0
120 invader=240:bomb=252:missile=239
130 gun$(1)=CHR$(244)+CHR$(246)+CHR$(247)
140 gun$(2)=CHR$(244)+CHR$(245)+CHR$(247)
150 SYMBOL 252,0,60,24,60,60,60,24,0
160 SYMBOL 240,66,36,60,90,126,60,36,36
170 SYMBOL 241,36,36,60,90,126,60,36,66
180 SYMBOL 242,15,127,255,204,204,255,127,15
190 SYMBOL 243,240,254,255,51,51,255,254,240
200 SYMBOL 244,0,0,0,0,15,8,8,255
210 SYMBOL 245,24,255,129,129,255,255,255,255
220 SYMBOL 246,0,0,24,255,255,255,255,255
230 SYMBOL 247,0,0,0,0,240,16,16,255
240 ENT 1,10,5,1,10,10,1,10,5,1
250 ENT -2,10,5,2,10,-5,2
260 GOSUB 990:newal=0
270 add=0:GOSUB 1390
280 WHILE INKEY(66) AND alive>0
290 IF aliens=0 THEN GOSUB 990
300 GOSUB 360
310 dum=dum XOR 100:SOUND 130,dum,10
320 invader=invader XOR 1
330 WEND
340 WHILE INKEY$<>"":WEND
350 STOP
360 'Move invaders
370 near=near+xdir:far=far+xdir
380 IF far>78 OR near<3 THEN md=1:xdir=-xdir ELSE md=0
390 near=80:far=0
400 FOR a=1 TO 5
```

```
410 GOSUB 590:GOSUB 730:GOSUB 1280
420 GOSUB 1470
430 FOR b=1 TO 10
440 IF x(a,b)=-1 THEN GOTO 560
450 xp=x(a,b):yp=y(a,b):ch=32:GOSUB 1180
460 x(a,b)=x(a,b)+xdir:y(a,b)=y(a,b)+md
470 xp=x(a,b):yp=y(a,b):ch=invader:GOSUB 1180
480 IF x(a,b)<near THEN near=x(a,b) ELSE IF x(a,b)>far
THEN far=x(a,b)
490 high=0
500 FOR v=1 TO 5
510 IF x(v,b)<>-1 AND y(v,b)>y(high,b) THEN high=v
520 NEXT
530 IF high=a AND FNR>25600 THEN GOSUB 1220
540 r=a:c=b:GOSUB 910
550 IF y(a,b)>24 THEN LOCATE 37,1:PRINT "INVADED":alive=0
560 NEXT b
570 NEXT a
580 RETURN
590 'Move gun
600 ogx=gx
610 IF FNlook$(gx)<>FNgun$(fx) THEN GOSUB 1720:RETURN
620 IF NOT INKEY(71) AND gx>3 THEN gx=gx-1
630 IF NOT INKEY(63) AND gx<77 THEN gx=gx+1
640 IF gx=ogx THEN GOTO 680
650 xp=ogx-1:yp=25:p$="    ":GOSUB 1810
660 IF FNlook$(gx)<>"    " THEN GOSUB 1720:RETURN
670 GOSUB 1880
680 IF INKEY(18) OR fx<>0 THEN GOTO 720
690 fx=gx:fy=24:xp=fx:yp=fy:ch=missile:GOSUB 1180
700 SOUND 1,400,15,,,1
710 GOSUB 1880
720 RETURN
730 'Move missile
740 IF fx=0 THEN RETURN
750 xp=fx:yp=fy:ch=32:GOSUB 1180
760 fy=fy-1
770 IF fy=1 THEN fx=0:GOSUB 1880:RETURN
780 IF screen(fx,fy)=32 THEN GOTO 890
790 IF screen(fx,fy)=ship1 OR screen(fx,fy)=ship2 THEN
GOSUB 1880:GOSUB 1550:RETURN
800 FOR row=1 TO 5
810 FOR col=1 TO 10
820 IF x(row,col)<>fx OR y(row,col)<>fy THEN GOTO 870
830 x(row,col)=-1
840 xp=fx:yp=fy:ch=32:GOSUB 1180
850 aliens=aliens-1:add=10:GOSUB 1370
```

```
860 row=5:col=10:fx=0
870 NEXT col,row
880 IF fx=0 THEN GOSUB 1880:RETURN
890 xp=fx:yp=fy:ch=missile:GOSUB 1180
900 RETURN
910 'Missile Hit check. indices of array in r and c
920 IF fx=0 THEN RETURN
930 IF screen(fx,fy)<>invader THEN RETURN
940 xp=fx:yp=fy:ch=32:GOSUB 1180
950 fx=0:x(r,c)=-1
960 GOSUB 1880
970 aliens=aliens-1:add=10:GOSUB 1370
980 RETURN
990 'new aliens
1000 FOR r1=1 TO 5
1010 FOR r2=1 TO 10
1020 x(r1,r2)=r2+r2:y(r1,r2)=r1+r1+top
1030 NEXT r2,r1
1040 aliens=50
1050 FOR r1=1 TO 80
1060 FOR r2=1 TO 25
1070 screen(r1,r2)=32
1080 NEXT r2,r1
1090 xdir=1:near=2:far=2
1100 top=top+1
1110 GOSUB 1880
1120 newal=-1
1130 FOR bc=1 TO 20
1140 IF bombx(bc) THEN
xp=bombx(bc):yp=bomby(bc):ch=32:GOSUB 1180:bombx(bc)=0
1150 NEXT
1160 IF sx<>0 THEN GOSUB 1650
1170 RETURN
1180 'Print to array and screen
1190 screen(xp,yp)=ch
1200 LOCATE xp,yp:PRINT CHR$(ch);
1210 RETURN
1220 'Drop bomb
1230 FOR z=1 TO 20
1240 IF bombx(z)=0 THEN empty=z:z=20
1250 NEXT
1260 bombx(empty)=x(a,b):bomby(empty)=y(a,b)+1
1270 RETURN
1280 'Move bombs
1290 FOR z=1 TO 20
1300 IF bombx(z)=0 THEN GOTO 1350
1310 xp=bombx(z):yp=bomby(z):ch=32:GOSUB 1180
```

```
1320 bomby(z)=bomby(z)+1
1330 IF bomby(z)=26 THEN bombx(z)=0:GOTO 1350
1340 xp=bombx(z):yp=bomby(z):ch=bomb:GOSUB 1180
1350 NEXT
1360 RETURN
1370 'Score
1380 SOUND 1,200-add,5
1390 score=score+add
1400 sc$=STR$(score)
1410 l=LEN(sc$)
1420 FOR zero=l TO 5
1430 sc$="0"+sc$
1440 NEXT
1450 LOCATE 38,1:PRINT sc$
1460 RETURN
1470 'Move ship
1480 IF NOT se THEN RETURN ELSE se=0
1490 IF sx=0 THEN IF FNR>250 THEN sx=1:GOSUB 1650:SOUND
132,200,10000,8,,2 ELSE RETURN
1500 IF screen(sx,2)<>ship1 OR screen(sx+1,2)<>ship2 THEN
GOTO 1550
1510 xp=sx:yp=2:ch=32:GOSUB 1180
1520 sx=sx+sdir:IF sx>77 THEN GOSUB 1610:sx=0:SOUND
132,100,0,0:RETURN
1530 GOSUB 1650
1540 RETURN
1550 'Ship hit by missile
1560 add=100:GOSUB 1370
1570 GOSUB 1610:sx=0
1580 fx=0:SOUND 132,100,0,0
1590 GOSUB 1880
1600 RETURN
1610 'Remove ship
1620 xp=sx:yp=2:ch=32:GOSUB 1180
1630 xp=sx+1:yp=2:ch=32:GOSUB 1180
1640 RETURN
1650 'Print ship
1660 xp=sx:yp=2:ch=ship1:GOSUB 1180
1670 xp=sx+1:yp=2:ch=ship2:GOSUB 1180
1680 RETURN
1690 'Enable ship movement
1700 se=-1
1710 RETURN
1720 'Kill the gun
1730 alive=alive-1
1740 FOR bc=1 TO 20
1750 IF bombx(bc)=0 THEN GOTO 1770
```

```
1760 IF bombx(bc)=gx AND bomby(bc)=25 THEN bombx(bc)=0
1770 NEXT
1780 SOUND 129,100,20
1790 xp=gx-1:yp=25:p$=FNgun$(fx):GOSUB 1820
1800 RETURN
1810 'Print a string to array at xp,yp
1820 LOCATE xp,yp
1830 PRINT p$
1840 FOR zz=xp TO LEN(p$)+xp-1
1850 screen(zz,yp)=ASC(MID$(p$,zz-xp+1))
1860 NEXT
1870 RETURN
1880 'Print the gun base
1890 xp=gx-1:yp=25:p$=FNgun$(fx):GOTO 1810
```

Printed by Jiffy Print Limited, Luton, Beds.