# DOCUMENTATION FOR EXTENDED BASIC COMPILER 2.12

By Harry Wilhelm – 2012-2014

The BASIC and Extended BASIC languages are arguably the most versatile of the languages available for the TI99/4A. Programs are easy to write, relatively understandable, and simple to modify and edit, with lots of error checking to facilitate program development. The main drawback is that the double interpreted nature of these languages make them extremely slow.

The intent in writing my Extended BASIC compiler was to make it possible to take full advantage of the simple program development offered by Extended BASIC, then make an end run around the speed limitations. The goal was to implement Extended BASIC as fully as possible within the time limits of the programmer and the memory limits of the machine. There *are* limitations and you may need to adjust your programming style a bit, but in general, all the major features of XB run the same when compiled. This means that you can concentrate on writing the XB code and testing it in the XB environment. After the program has been perfected in Extended BASIC it can then be compiled into an equivalent code that functions at a speed approaching that of assembly language. The average Extended BASIC program will run about 20 times faster after being compiled, and certain operations will run up to 70 times faster.

There are several methods by which the compiler achieves this speed increase. First, Extended BASIC must perform a lengthy prescan operation before a program can even start. This is done in advance by the compiler and becomes part of the compiled code. Second, a TI BASIC or XB program is interpreted twice by the computer; once by the BASIC interpreter, and a second time by the GPL interpreter. The compiler generates "threaded code" which needs its own interpreter (called the runtime routines), but at least only one interpreter is involved, and it's a fast one! Third, integer arithmetic is used throughout instead of floating point arithmetic. This alone makes the code run at least 5 times faster, albeit without the versatility of 13 digit floating point accuracy. Fourth, to increase the speed even more, virtually no error trapping is done. Any error reports that are given are not very helpful anyway because you won't know the line number where the error happened. Therefore it is *imperative* that the BASIC program be thoroughly debugged before you attempt to compile it!

The Extended BASIC compiler has been tested with a genuine TI-99/4a, with Classic 99, and with Win994a and is compatible with all of them.

The compiler is designed to be on a disk placed in drive #1.

## Compiling an Extended BASIC program

Following is a brief outline of the steps used to compile a program.

1 – Save an Extended BASIC program to disk in merge format.
2 – Run the compiler. This reads the merge format BASIC program and creates an assembly language source code file.
3 – Run the assembler. This reads the assembly language source code file created by the compiler and creates an assembly language object code file.
4 – Run the compiler loader. This loads the assembly language object code file into memory in a form that can be saved and run via Extended BASIC or in EA5 format.

You can see that five files are used in the process of compiling a program. To help keep track of them, I recommend using the following conventions. If the original XB program is called TEST then:
TEST = original XB program
TEST-M = merge format file
TEST-S = assembly source code file
TEST-O = assembly object code file
TEST-C = compiled code ready to run from Extended BASIC
TEST-A – compiled code ready to run from EA5
(The above are just suggestions. Feel free to use whatever file names you wish.)

The following steps explain how to compile a program in detail:

Write or load the BASIC or Extended BASIC program. This can be done in either BASIC or XB. Using TI BASIC gives you the use of character sets 15 and 16, which is about the only reason for using it. Extended BASIC offers sprites, DISPLAY AT and ACCEPT AT and some additional assembly language subroutines written specifically for the compiler. Debug the program; then check the listing to be sure that only integers are used in the variables.

Save a BASIC or Extended BASIC copy for future development (let's call it VMBWTEST):
SAVE DSKn.VMBWTEST<enter>

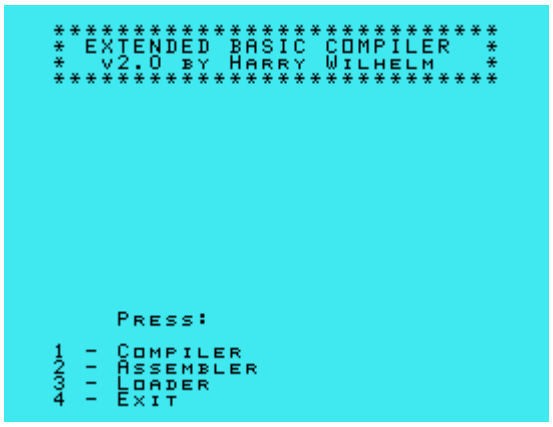
(All the operations below are done in Extended BASIC.)

To create the merge format file required by the compiler:
SAVE DSKn.VMBWTEST-M,MERGE<enter>
If you are compiling a TI BASIC program you should load it into XB. Even if the program won't run in XB, you can save it in merge format.
It's easy to forget to add the MERGE, which confuses the compiler later on. When you do this using a Horizon Ram disk you will find that you cannot then SAVE DSK3.VMBWTEST-M,MERGE without first doing DELETE "DSK3.VMBWTEST-M"

Put the compiler disk into drive #1

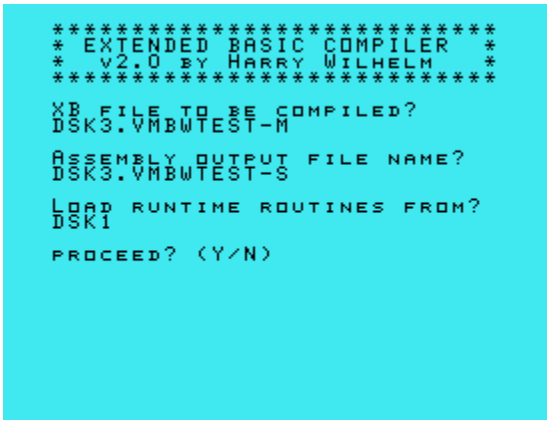
Quit with <Fctn => and then select Extended BASIC or type: RUN "DSK1.LOAD"<enter>

You will get the following screen:




Press 1 for the compiler.

As you go through the next menu you will be prompted for three things:
XB file to be compiled, the assembly output file name, and the location of the runtime routines. The screen-shot below shows how it should look after you have entered the file names:




Press Y or <enter> if everything is as you want it, otherwise press N to start over.

The compiler will compile the program. When finished you are presented with the main compiler menu. If you press 2 the assembler will load via Funnelweb. There are two advantages to using Funnelweb. First, it runs out of Extended BASIC so you don't have to insert the E/A cartridge. Second, the assembly source code file name that you just made appears on the screen when you get to the assembler. Please note that this is not the complete Funnelweb package. Nothing loads except the assembler and a disk catalog utility. The disk catalog utility can be accessed by pressing Fctn 7. You can exit the disk catalog by pressing Ctrl =. There are other assemblers that can be used. Classic 99 has one built into the E/A cartridge. Win994a has an assembler as part of the package that might run quite a bit faster than the standard TI assembler. (Actually, the standard assembler seems to run faster in Win994a, probably due to faster disk access.)

If you are running the assembler out of the compiler, press 2, then press 2 three more times which should take you to the assembler.(If the word processor menu appears just press the space bar to bring up the assembly language menu)

You will be prompted for the source file name, the object file name that the assembler will create and then some options. Use the space bar to erase the “C” under options but be sure to leave the “R”

When done you should see a screen something like this:


Press <Fctn 6> to proceed or <Fctn 8> to redo.

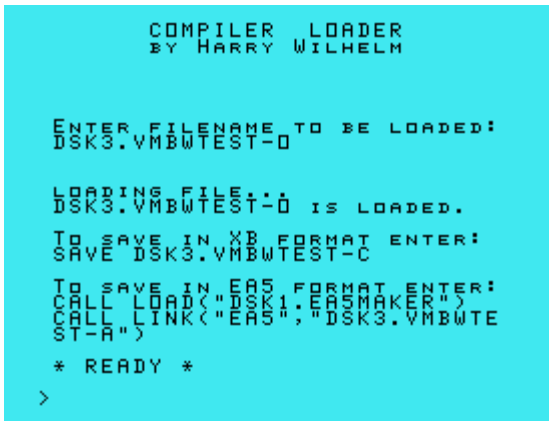
The assembler will assemble the program. Be sure the runtime routines (i.e. the compiler disk with the files RUNTIME1 and RUNTIME2) are in the drive you said they would be! Make sure your disk has room on it for the assembly source file. When assembling I've received a number of DSR error messages which have perplexed me until I realized that the disk was full.

When the assembler has finished, you should get a message saying:
"0000errors, Press ENTER to continue"
At this point press <Enter> and then "quit". From the TI title screen select XB.

With the compiler disk in drive 1, wait for the compiler menu, then press 3 to run the compiler loader.

You will receive the following prompt:
Enter filename to be loaded:
Type the name of the object code file the assembler just created and press enter
DSKn.FILENAME-O<enter>

The compiler loader will load the object file into memory and create a program that will run via XB.
When the loader is done you will see a screen like this:




Follow the prompts to save the program in an XB compatible format or to save it into an EA5 compatible format. Change the disk numbers if you want to save to a different drive number.

Now you should be able to RUN the program.

## Differences from Extended BASIC

An ideal compiler would be able to take any Extended BASIC program and compile it with no changes necessary so that it would run exactly the same only faster. This compiler falls short of that ideal, but does come close.

Following is a summary of the major differences between the compiler and Extended BASIC.

The biggest difference that you will have to deal with is that all numbers are integers from -32768 to 32767.
Here are some examples showing how the compiled code differs from the XB code:
32767+1=32768 in BASIC
32767+1=-32768 in the compiled code
200*200=40000 in BASIC; -25536 in compiled code because of the integer arithmetic.
If an operation such as dividing or SQR can give a non integer result, then you should use INT in the BASIC program to be sure that the BASIC and compiled programs function the same.

Because RND returns a number between 0 and 1, the INT of RND is always 0. Because of this, the following line of code won't work properly in the compiled code:.10 IF RND>.5 THEN 100 ELSE 200
There is a work around built into the compiler that deals with this problem. You have to multiply the RND by some number and then INT the result. Instead of the example above you should use:

```
10 IF INT(RND*2)=1 THEN 100 ELSE 200
```

This gives either a 0 or a 1 in both Extended BASIC and the compiled code.

The timing of delays loops has to be modified. FOR I=1 TO 500::NEXT I gives a delay of several seconds in XB or BASIC; a fraction of a second in the compiled code. The best way to do a delay is to use CALL SOUND. For a 2 second delay you would use CALL SOUND(2000,110,30)::CALL SOUND(1,110,30). Neither XB nor the compiler can process the second call sound until the first has finished, so you get the full 2 second delay. This method makes it possible to create delays that work the same in XB or compiled code.

IF-THEN-ELSE will only work with line numbers. See the discussion below for ways to partially work around this limitation.

Nested arrays cannot be used. See discussion below.

User defined subprograms are not supported.

Trig functions, LOG and DEF not supported.

Speech is not supported.

Assembly language subroutines cannot be used except for the four included with the compiler.

Disk and other peripheral access is not supported at the present time.

Only one variable can be assigned at a time in a LET statement. A line like:
10 A$,B$,C$=”B” will crash the compiler.

## Supported Instructions

Following is a list of the TI Extended BASIC operations supported by the compiler:

Multiple statement lines can be used, with the statements separated with a double colon.
Parentheses can be used to change the mathematical hierarchy used to evaluate expressions.

The arithmetic operators + - * / ^ work as they do in XB within the limits of integer arithmetic. Remember that because of the integer arithmetic, dividing 5/2 will give 2, not 2.5. You can use INT in the XB program when dividing (for example INT(5/2) to be certain that XB and the compiler give the same results.

The logic operators NOT, AND, XOR, OR work the same as in XB.

The relational operators < > = <> <= >= work the same as in XB.

GOTO and GO TO
GOSUB and GO SUB
ON-GOTO and ON-GO TO
ON-GOSUB and ON-GO SUB
RETURN
END
STOP
FOR-TO-STEP – The step is optional; +1 is assumed if no step is specified.
NEXT
READ
DATA – Do not GOTO a DATA statement!
RESTORE

ABS
MAX
MIN
INT
SGN
SQR – gives same number as INT(SQR(N)) in XB
ASC
LEN
POS
VAL
CHR$
SEG$
STR$
RPT$ – the string is truncated if over 255 characters and no warning is given.

RANDOMIZE can be used, but has no effect; it is done automatically
RND returns a value of 0. RND is only useful when it is multiplied by another number. i.e. INT(RND*6) gives the same results (0,1,2,3,4,5) when compiled as it does when used in XB. The order is not important – it can be (RND*6) or (6*RND)

String concatenation (i.e. A$&B$) works the same as in XB. The string is truncated if over 255 characters but no warning is given.

IF-THEN-ELSE will only work with line numbers, like in TI BASIC. The more advanced XB style of IF-THEN-ELSE is not supported. Being able to use multiple statements in a line provides ways to partially work around this limitation:

```
10 IF X>3 THEN Y=7::Z=19
```
can be changed to:
```
10 IF X<=3 THEN 20::Y=7::Z=19
```
Another example:
```
10 IF Q=5 THEN X=8::Y=14::Z=23 ELSE X=11::Y=4::Z=9
```
can be changed to:
```
10 IF Q=5 THEN 20::X=11::Y=4::Z=9::GOTO 30
```
(Note that XB is happy with no ELSE here)
```
20 X=8::Y=14::Z=23
30 !go on
```

INPUT works almost exactly like in XB, with the following differences. You can use the optional prompt. You can input more than one variable, but you must use the optional prompt to do this, even if it is just a question mark.. If inputting more than one variable, data being inputted is separated by the first comma the compiler comes to. Quotation marks will not behave as they do in XB. Rather, they are simply input as part of the string. You cannot use quotation marks to input leading or trailing spaces.
LINPUT works exactly like in XB.
ACCEPT works almost exactly like it does in XB. AT, BEEP, ERASE ALL, SIZE and VALIDATE are all supported with one difference: VALIDATE requires that you provide the string expression. UALPHA, DIGIT, NUMERIC are not supported.

PRINT works like TI Extended BASIC. You can use TAB, commas, semicolons and colons.
DISPLAY works just like in XB. You can use AT(row,col), BEEP, ERASE ALL, and SIZE(length) as well as TAB, commas, semicolons and colons. DISPLAY USING is not supported.

DIM is optional, as it is in XB, but using it can reduce size of the compiled program.
OPTION BASE
ARRAY LIMITATION – Important!! The program being compiled cannot use nested arrays. For example, if you have the two arrays DIM A(10),DIM B(10); you can use Q=A(X+Y-Z) but you can't nest the arrays like this: Q=A(B(7)). Use of nested arrays will cause the compiled program to crash!!! For the above example you have to split up the statement something like this:
X=B(7)::Q=A(X)

The following CALL subprograms function just like in Extended BASIC except as noted:

CALL COLOR
CALL CLEAR
CALL SCREEN
CALL CHAR
CALL HCHAR
CALL VCHAR
CALL SOUND cannot handle frequencies greater than 32767. (Neither can my ears!)
CALL GCHAR
CALL KEY
CALL JOYST
CALL CHARPAT
CALL CHARSET
CALL SPRITE
CALL MAGNIFY

CALL DISTANCE
CALL COINC
CALL LOCATE
CALL DELSPRITE
CALL POSITION
CALL PATTERN
CALL MOTION
CALL PEEK
CALL LOAD – can only be used to load values in RAM. Will not load assembly language subroutines.
CALL LINK – only works with the 4 additional assembly language subroutines included with the compiler and described in the section below called "Assembly language extras for the compiler".

LET is optional
REM – All remarks are removed from the compiled program, but you can GOTO a REM statement just like in XB. Use of REM will not increase the size of the compiled program.
! – the exclamation point for "remark" works just like the REM statement.

From the command mode in Extended BASIC:
CALL LINK("RUN") functions the same as RUN in XB. You cannot use RUN or RUN line # within a program.
CALL LINK("CON") functions the same as CON in XB
<FCTN 4> breaks the program as in XB except during INPUT or ACCEPT, or when running in EA5.

## NOT SUPPORTED:

RUN or RUN line #.
DEF
ATN
COS
EXP
LOG
SIN
TAN
DISPLAY USING

No File processing capabilities have been implemented at this time.

The following have no meaning in a compiled program:
LIST
NUM
RES
BREAK
UNBREAK
CON – use CALL LINK("CON") if running the compiled program from XB.
TRACE
UNTRACE
EDIT

## Assembly language extras for the compiler

One of the main concepts behind the compiler is to be able to thoroughly test the program in the Extended BASIC environment and only then proceed to compile the program. There are four assembly language extensions that have been written to address certain limitations in XB. These can be loaded into XB for testing. If you include CALLs to these subroutines in your program they will execute the same in the compiled program as they did in XB. Load these in the XB command mode by typing:
CALL INIT
CALL LOAD("DSK1.C-XTRAS)

The assembly routines will be loaded into low memory. When your XB program tries to LINK to them and you have forgotten to load them, XB will issue a syntax error message. (Use the command mode – do not add the above CALL INIT and CALL LOAD to your program – they will confuse the compiler.)

The four routines are:

**CALL LINK("EARLYC",NUMBER[...])**
This routine sets the sprite early clock. With the early clock on, the sprite's location is shifted 32 pixels to the left, allowing it to fade in and out on the left edge of the screen. The number is a combination of the sprite number and the sprite color. The last 2 digits are the sprite color which must be 2 characters long. The first one or two digits are the sprite number. To set the early clock on sprite #7 with the color as magenta the number should be 714. To set the early clock of sprite #13 with a color of black the number should be 1302. One call to EARLYC can set the early clock for up to 16 sprites. The early clock is not supported by Win994a.

**CALL LINK("DISPLY",row,col,string[,direction,repeat])**
One of the limitations of Extended BASIC is that you are restricted to only 28 columns on the screen. The two columns on both sides of the screen cannot be printed to directly. You have to use HCHAR or VCHAR and even then it's not simple.. DISPLY addresses this limitation. It will print a string starting at screen location row and col, and can use all 32 columns when it prints. By default it will print to the right. If it reaches the right hand edge of the screen it will drop down one row, go to the left side of the screen and continue printing. If it reaches the lower right corner of the screen it will continue printing at the upper left corner of the screen. If you want DISPLY to print in a different direction you can include the optional fourth parameter. A 1 will print each character one space to the right; a 2 will print each character 2 spaces to the right. A 32 will print text vertically down, 33 will print text diagonally down and to the right. A -32 will print text vertically upwards, and so on. If you want DISPLY to print the string repeatedly (somewhat like HCHAR or VCHAR) include the optional fifth parameter to tell DISPLY how many times you want the text to be repeated. If you want to use the repeat function, you must include the direction. Keep in mind that with DISPLY column 1 is the first column, while with DISPLAY AT column 1 is the third column on the screen.

**CALL LINK("VREAD", memory address, # bytes, string[ , . . .])**
VREAD reads the specified number of bytes from the VDP ram into a string variable. One call to VREAD can read up to 5 strings.

**CALL LINK ("VWRITE",memory address, string[ , . . .])**
VWRITE writes the string to the VDP ram starting at the specified memory address. One call to VWRITE can write up to 8 strings.

VREAD and VWRITE are extremely versatile subroutines. The programmer can use them to read a row from the screen, erase the row, input text using ACCEPT AT, then restore the screen to its original

state. You can open a small window, input text in the window, and scroll just the window. If you want to dig more deeply into the TI than XB normally allows, you can prepare strings ahead of time with a short utility that saves them in MERGE format, then include them in your program to write character definitions, to load sprite definitions and motions, to load a sound list to be played automatically, etc. It is beyond the scope of this manual to go into detail on these more advanced uses although the most important VDP addresses are listed below. The editor/assembler manual has information on memory locations in the VDP and how to set up sprites and sound lists.

Here is a short demo program that changes the lower case characters to white on blue, uses DISPLY to print a background on the screen, clears a small window, inputs text in the window, and scrolls the window using VREAD and VWRITE when you press enter.

| | |
|---|---|
| `100 FOR I=9 TO 12 :: CALL CO LOR(I,16,5):: NEXT I` | Changes lower case character set to white on blue |
| `110 CALL LINK("DISPLY",1,1," background",32,77)` | Uses DISPLY to print the word "background" vertically down and repeat 77 times |
| `120 FOR R=16 TO 20 :: CALL H CHAR(R,9,32,16):: NEXT R` | Clears a window on the screen |
| `130 ACCEPT AT(20,7)SIZE(16): Z$` | Lets you input text at the bottom of the window |
| `140 CALL LINK("VREAD",520,16 ,A$,552,16,B$,584,16,C$,616, 16,D$)` | Reads the lower 4 lines of the window |
| `150 CALL LINK("VWRITE",488,A $,520,B$,552,C$,584,D$)` | Scrolls the window by writing the four lines one line higher |
| `160 GOTO 130` | Input more text |

Remember that these assembly language subroutines are meant to be loaded into XB where they can be easily tested before compiling. They can also be used in XB programs that you don't want to compile.

## Some useful VDP addresses for VWRITE and VREAD

| | | |
|---|---|---|
| 0 – 767 | Screen Image Table | VDP address is given by (Row-1)*32+Column-1 |
| 768 – 879 | Sprite Attribute Table | Room for 28 sprites – each sprite needs 4 bytes<br>vertical position-1,horizontal position,<br>char #+96,color-1(+128 for early clock) |
| 1008 – 1919 | Pattern Descriptor Table | 8 bytes per character – char 30 starts at 1008 |
| 1920 – 2047 | Sprite Motion Table | 4 bytes per sprite.<br>Vertical velocity, horizontal velocity; sys use; sys use<br>(you can use 0 for the sys use value) |
| 2048 – 2079 | Color Table | 1 byte per character set – char set 0 = 2063<br>(foreground-1)*16+background-1 |
| 14301 – 16106 | Disk buffering area | Can safely be used for sound lists, etc. |

## Embedding SINE values in a string:

Due to the integer arithmetic, trig functions are not supported by the compiler. However, there is a way to use them in a program. You can produce a 91 byte long MERGE format program line that contains a string with the values for sine from 0 to 90 degrees multiplied by 255, then use SEG$ to extract the sine value for any degree from 0 to 90 and convert it to a number with ASC. Such a string would contain characters that cannot be input from the keyboard, so we have to use a program to generate it.

The following program can be used to generate a merge format file that contains the following program line:

```
10000 S$="a string containing 91 values for sine from 0 to 90,
multiplied by 255"
```

```
10 OPEN #1:"DSK3.SINE255",DI    100
SPLAY ,VARIABLE 163,OUTPUT
19 A$=CHR$(39)&CHR$(16)&CHR$          Line number - 39*256+16=10000
(83)&CHR$(36)&CHR$(190)&CHR$          S$ and =
(199)&CHR$(91)                        string constant; length of string
20 FOR ANGLE =0 TO 90
40 SINE=INT(255*SIN(ANGLE*PI          convert from radians to degrees and multiply
/180)+.5)                             by 255
50 A$=A$&CHR$(SINE)                   keep building string
80 NEXT ANGLE
90 A$=A$&CHR$(0)                      a zero at the end of the string
100 PRINT #1:A$
105 A$=CHR$(255)&CHR$(255)::
 PRINT #1:A$::PRINT #1:A$             Write >FFFF twice to write EOF
110 CLOSE #1
```

Let's say you wanted to launch a sprite with a velocity (VEL) and at an angle(ANG) between 0 and 90 degrees . (0 degrees is to the right, 90 degrees is straight up)
The column velocity (CVEL) is given by: VEL*COS(30) and the row velocity (RVEL) is given by: -VEL*SIN(30). But what do we do about the missing COS functions? Well, it turns out that COS(ang) is the same as SIN(90-ang), so....
You could run the above program, type NEW, then merge SINE255. Then add line 10010 to get the following subroutine:

```
10000 S$="a string containin
g 91 values for sine from 0 t
o 90, multiplied by 255"
10010 RVEL=INT(-VEL*ASC(SEG$
(S$,ANG+1,1))/255):: CVEL=IN
T(VEL*ASC(SEG$(S$,91-ANG,1))
/255):: RETURN
```

Save this in MERGE format for future use. You would call this from an XB program something like this:

```
10 VEL=50::ANG=53::GOSUB 10000::CALL MOTION(#1,RVEL,CVEL)
```

The above subroutine is included on the compiler disk under the file name “SINE255”

The program above beginning with 10 OPEN #1 should have enough comments to give you ideas on how to write something similar that can generate strings containing character definitions, sprite data, or sound lists. You should know that the strings generated contain characters that cannot be input from the keyboard. This means that XB will complain if you try to edit the line. Besides speed, one advantage to using a string like this for defining characters is that the string is more compact. It uses 8 bytes per character and the normal CALL CHAR uses 16 bytes per character. But you lose the ability to easily edit the line or even to understand what is in it.

## Disk Access

At present there is no provision for accessing peripherals. However, there is a way to get a limited degree of disk access, which might be useful for saving simple things like the high score in a game. Unless you choose the E/A5 option the compiler will create a program that loads and runs in the Extended BASIC environment. If you list the program you will see one line something like this:
10 CALL INIT :: CALL LOAD(8192,235,002) :: CALL LINK("RUN")

You could write a few lines of code before line 10 that read the name and high score from a disk file and display them on the screen. The screen is not cleared when the compiled program starts, so the compiled program could can the name and high score with GCHAR, then clear the screen and start up the game. If a new high score is reached during the game, then the name and score can be printed on the screen when exiting the program. Then a few lines of code after line 10 can read the name and score and save them in the disk file to be retrieved the next time the game is run.

This method will not work if the compiled program is running from EA5.

## In case of trouble

Here are some steps that you can take to try to sort things out if there is a problem with the compiler.

Sometimes the compiler does not like one or more of the statements in the XB program. Normally it will say "compiling Line 10" (or whatever the first line number is). If successful in compiling that line it will then say "compiling Line 20" and so on until it is done. If it gets stuck on a line number then there is something in that line that it doesn't like. Check the XB program and try to see which statement is unsupported. If you want to see this in action (actually in inaction) try to compile this program:
`10 OPEN #1:"DSK3.TEST"` and you will see it get stuck on line 10

The compiler will report if it was able able to successfully compile your XB program. If so, choose option 2 to assemble the code. The assembler might issue an error message during the assembly process. If so then the error is probably in the source code file the compiler just made, not in the runtime routines. The message will be something like this: `undefined symbol 0141.` This tells you that there is something wrong in line 141 of the compiled source code. Examine it to see if you have used an unsupported statement or if there is something that doesn't look right. You can use a text editor, the editor for the E/A cart, or TI99dir which lets you view a file. Except for B @RUNEA5 there should be nothing but DATA statements, something like the following compiled code:

```
        DEF RUN,CON
RUNEA   B @RUNEA5
FRSTLN
L100
FOR1
        DATA FOR,NV1,NC1,NC2,ONE,0,0
        DATA COLOR,NV1,NC3,NC4
        DATA NEXT,FOR1+2
L110
        DATA DISPLY,NC1,NC5,SC1,NC6,NC7
L130
        DATA AT,NC8,NC9
        DATA SIZE,NC3
        DATA ACCEPT,SV1

LASTLN DATA STOP
- - - - (lines are omitted)- - - -
SC0
SC1     DATA SC1+2
        BYTE 9,98,97,99,107,103,114,111,117,110
        EVEN
SV0
SV1     DATA 0 Z$
- - - - (lines are omitted)- - - -
        COPY "DSK1.RUNTIME"
        END
```

The code the compiler creates should be understandable when compared to the original XB program. Look for a missing DATA statement or something that doesn't look right. If the assembler gives a line number you should be able to find the error easily. I like using TI99dir to view files, but when you look at a file there are no line numbers and you have to count the lines by hand.