

Start with the *README* file in the docs, then study the file *Using XB GDP*, which has directions on how to set up and use the Game Developer's Package on your equipment. Be sure you understand the steps to follow in developing, compiling, assembling, and loading an XB or XB256 program. Then come back here for detailed information about the compiler.

EXTENDED BASIC COMPILER

By Harry Wilhelm – 2012-2024

09/27/24

The Extended BASIC language is 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 Extended BASIC makes it 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 XB, 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 will probably 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 or XB256 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 at least 30 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, an XB program is interpreted twice by the computer; once by the Extended BASIC interpreter, and a second time by the GPL interpreter. The compiler generates "threaded code" which needs its own interpreter (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 Extended BASIC program be thoroughly debugged before you attempt to compile it!

The compiler has been expanded to include all the XB256 assembly language extensions except for CAT and SL2VDP. XB256 removes most of the graphics restrictions imposed by Extended BASIC. It lets you toggle between two independent screens. Screen1 is the graphics mode normally used by Extended BASIC; Screen2 lets you define 256 characters, more than double the number normally usable in XB. When in screen2, you can use up to 28 double sized sprites using the patterns available to Screen1. You can scroll screen characters left, right, up, or down and specify a window area for scrolling, leaving the rest of the screen unchanged. Other routines let you scroll smoothly one pixel at a time to the left, right, up or down. There are miscellaneous subroutines that let you highlight text, set the sprite early clock, print in any direction on the screen using all 32 columns, read from or write to the VDP RAM, write compressed strings to VDP, move sound tables into VDP, and more.

With XB256 you can test your program in the XB environment, then use the compiler to get a huge performance increase. Once compiled an XB program that required XB256 becomes a stand alone program. It does not need XB256 to run.

The compiler is part of the XB Game Developer's Package. This is designed to work with Mike Brent's Classic99 emulator which is an easy, fast and powerful development tool. It eliminates many of the annoyances that come when you are running on a real TI99, such as slowness, limited room in the disk drives, difficulty in reading text files, etc. Follow the directions in *Using XB GDP* to set up the Game Developer's Package on your equipment.

The steps you need to follow in developing, compiling, assembling, and loading an XB or XB256 program are described in *Using XB GDP*.

The rest of this manual will describe the compiler, what instructions are supported, etc.

Starting at page 14, there is a section that describes how to use Asm994a, which is one of the cross assemblers available for the TI99.

Starting at page 17, there is information on how to combine assembly subroutines with a compiled XB program. This lets you compile XB programs that use XXB, T40XB, T80XB, The Missing Link, The Missing Link Graphics adventure. Or you can use your own assembly support subroutines.

If the program you are compiling was written in TI BASIC, it is possible to use the runtime routines from the original TI BASIC compiler. This is limited to BASIC only, but it generates programs that are more compact and a wee bit faster than those created by the newer compiler. This is described starting at page 28.

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 short overview of the 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.

In Extended BASIC, RND returns a number between 0 and 1, so 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. One way to have the same delay in both compiled and XB programs 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 whether in XB or compiled code. Another way is to use CALL LINK("DELAY",2000) in XB256.

IF-THEN-ELSE now can use the more versatile Extended BASIC format, Earlier there were some limitations when using complex IF-THEN-ELSE statements. Those restrictions no longer apply.

User subprograms are fully supported with this difference: when using subprograms, the compiler will shorten the name to the first 6 letters. You can use longer names as long as the first six letters do not duplicate another subprogram. CALL UPDATEWHITE and CALL UPDATEBLACK would not compile properly. CALL UPDATWHITE and CALL UPDATBLACK would be fine, as the compiler sees them as UPDATW and UPDATB

Trig functions, LOG and DEF are not supported.

The latest compiler, part of the "Jewel" package, lets you use assembly language subroutines. XXB, T40XB, T80XB, and The Missing Link are part of the Jewel package. Or you can use your own custom assembly routines to support a compiled XB program. See page 17 for more information.

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.
Do not put a double colon at the end of a program line. XB ignores this, but the compiler crashes.

The arithmetic operators + - * / ^ work as they do in XB within the limits of integer arithmetic.
Parentheses can be used to change the mathematical hierarchy used to evaluate expressions. 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.

ABS

ASC

CHR\$

DATA

But you cannot GOTO a DATA statement

END

FOR-TO-STEP

As in XB, the step is optional; +1 is assumed if no step is specified.

GOSUB and GO SUB

GOTO and GO TO

But do not GOTO a DATA statement

INT

LEN

LET – is optional just like in XB

MAX

MIN

NEXT

ON-GOSUB and ON-GO SUB (this may cause trouble if used in an IF/THEN/ELSE statement)

ON-GOTO and ON-GO TO (this may cause trouble if used in an IF/THEN/ELSE statement)

POS

READ

RESTORE *But RESTORE cannot point to a comment; it must point to a DATA statement*

RETURN

RPT\$ – the string is truncated if over 255 characters and no warning is given.

SEG\$

SGN

SQR – gives same number as INT(SQR(N)) in XB

STOP

STR\$

SUB

only the first 6 letters of the subprogram name are used.

SUBEND

SUBEXIT

VAL

RANDOMIZE can be used, but has no effect; it is done automatically. The random number seed is the 2 bytes at >83C0 or -31808. The TI is constantly scrambling this value until the program starts up. You can CALL LOAD(-31808,N1,N2) to get the same random number sequence, or CALL PEEK(-31808,N1,N2) to store the seed so it can be restored later.

Because of the integer arithmetic, RND always returns a value of 0. The workaround is to multiply RND by another number. i.e. INT(RND*6), which 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 now can use the more versatile Extended BASIC format.

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.

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 a string expression., which can be numbers, upper case characters, etc. UALPHA, DIGIT, NUMERIC are not supported. If you are using SIZE the computer will give a “honk” instead of a “beep” when it reaches the right hand limit.

PRINT works like TI Extended BASIC. You can use TAB, commas, semicolons and colons. *Do not print more than 20 variables in a print statement.* See page 7 for more information.

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. (An XB trick to save memory is to use DISPLAY ERASE ALL to clear the screen. This crashes the compiler which expects DISPLAY to actually display something. Use CALL CLEAR if you just want to clear the screen.) With SIZE, using a print list with more than one element will cause the line to be cleared to column 28. If that is a problem, you can avoid it by concatenating and displaying a single string.

DIM and OPTION BASE are optional, as is in XB, but using them can reduce the size of the compiled program.

ARRAYS: Nexted arrays can now be used. If you have the two arrays DIM A(10),DIM B(10); you can now nest the arrays like this: Q=A(B(7))

Multiple variables can be assigned in a LET statement. Lines like these are now permissible:

```
10 A$,B$,C$="Hello World"
```

```
10 IF Z=7 THEN A,B,C=3
```

You cannot use ON GOTO or ON GOSUB in an IF/THEN/ELSE statement such as:

```
570 IF V=-10 THEN GOSUB 1420 :: ON FL+1 GOTO 500,730.
```

Error Messages

Although virtually no error checking is done, there are three conditions that can cause an error message to be issued. This can only happen when running the compiled program in XB with the runtime routines in high memory. Running in EA5 or in XB using all 32K of memory (runtime routines in low memory) will simply “quit” if the compiled program encounters these errors.

“BAD ARGUMENT IN 10” - issued if you take the SQR of a negative number.

“DATA ERROR IN 10” - issued if you read past the last DATA statement.

“MEMORY FULL IN 10” - issued if you run out of memory.

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

CALL CHAR
CALL CHARPAT
CALL CHARSET
CALL CLEAR
CALL COINC
CALL COLOR
CALL DELSPRITE
CALL DISTANCE

CALL LINK("FLICK") and CALL LINK("FLICKX") flicker routine used when more than four sprites are on a line. The folder *FLICKERROUTINE* has the information and files you need to use this.

CALL GCHAR – can do multiple GCHARs in a single statement

CALL HCHAR – can do multiple HCHARs in a single statement

CALL JOYST – Both JOYST and KEY use the same internal keyscan routine.

CALL KEY – If KEY immediately follows JOYST, they will share one keyscan which is a bit faster. The key units must match. e.g 10 CALL JOYST(1,X,Y)::CALL KEY(1,K,S)
If KEY *does not* immediately follow JOYST then each does its own keyscan as in XB.

CALL LINK – works with the assembly subroutines provided by XB256. Other assembly subroutines can be used, such as T40XB, TML, or your own, as described on page 17.

The name of the assembly subroutine cannot be a string variable like CALL LINK(A\$)

CALL LDCR (this is part of XB 2.9 G.E.M.)

CALL LOAD – can only poke values to RAM. Assembly subs can be used as described on page 17

CALL LOCATE

CALL MAGNIFY

CALL MOTION

CALL MOVE (this is part of XB 2.9 G.E.M.)

CALL PATTERN

CALL PEEK

CALL PEEKV

CALL POKEV

CALL POSITION

CALL SAY – some minor limitations. See page 11 for more information.

CALL SCREEN – saves the screen color like CALL LINK("SCREEN") in XB256

CALL SOUND – cannot handle frequencies greater than 32767. (Neither can my ears!)

CALL SPGET

CALL SPRITE

CALL STCR (this is part of XB 2.9 G.E.M.)

CALL VCHAR – can do multiple VCHARs in a single statement

CALL (user defined subprogram) Only the first six letters of the subprogram name are used.

Some names are reserved for the compiler. The table on page 12 has a list of these.

All the assembly language subroutines in XB256 are supported except for CAT and the IV254 utilities RUN, RUNL1, and SAVEIV.

REM and ! – 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. (But remember that RESTORE cannot point to a remark; it must point to a DATA statement.)

Peripheral access is now supported for DISPLAY, VARIABLE files. Page 13 has more information.

RUN or RUN line #. Usually you cannot use RUN or RUN line # within a compiled program, The one exception is if you want to chain cartridges together. See *Using XB GDP.pdf* for more information. See page 11 below if you want to chain compiled programs running from XB.

When running from XB, <FCTN 4> breaks the program except during INPUT or ACCEPT. It has no effect when running in EA5.

Use CALL LINK("RUN") and CALL LINK("CON") instead of RUN and CON.

NOT SUPPORTED – most of these will cause an error when assembling:

A statement separator at the end of a line will crash the compiler:

```
10 FOR I=1 TO 10 :: 10 FOR I=1 TO 10 :: !comment 10 FOR I=1 TO 10 :: REM comment
```

ATN

COS

DEF a line with DEF will be omitted by the compiler

DISPLAY USING will compile without error, but will not use image string.

CALL ERR

EXP

IMAGE a line with IMAGE will be omitted by the compiler

LOG

SIN

TAN

The following have no meaning in a compiled program:

BREAK

CON – use CALL LINK("CON") if running the compiled program from XB.

EDIT

LIST

NUM

RES

TRACE

UNBREAK

UNTRACE

The compiler uses a string that can be up to 255 bytes long for processing lines of code. This is almost always large enough. However, too many semicolons, commas or colons in a PRINT statement can cause the compiler to generate a string longer than 255 bytes. Although the compiler does not crash, the line is truncated and the code generated will not run properly.

```
10 PRINT A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P,Q,R,S,T,U,V,W
```

This compiles properly, but adding one more variable will be too long. You should be safe as long as there are no more than 20 variables in a print statement.

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.

A program can be used to generate a merge format file consisting of just one line:
10000 S\$="a string containing 91 values for sine from 0 to 90, multiplied by 255"

Here is the program:

10 OPEN #1:"DSK3.SINE255",DI	
SPLAY ,VARIABLE 163,OUTPUT	
19 A\$=CHR\$(39)&CHR\$(16)&CHR\$(83)&CHR\$(36)&CHR\$(190)&CHR\$(199)&CHR\$(91)	Line number - 39*256+16=10000
20 FOR ANGLE =0 TO 90	S\$ and =
40 SINE=INT(255*SIN(ANGLE*PI/180)+.5)	string constant; length of string
50 A\$=A\$&CHR\$(SINE)	convert from radians to degrees and multiply by 255
80 NEXT ANGLE	keep building string
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 cosine functions? Well, it turns out that $\cos(\text{angle})$ is the same as $\sin(90-\text{angle})$, which gives us a solution:

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 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. These will run fine, but 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 while 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. The COMPRESS utility in XB256 automates the creation of this type of DATA line.

Disk Access

Disk and other peripheral access is now supported with some limitations:

DISPLAY, VARIABLE is the only file type recognized, but you can use any length desired from DV1 to DV254.

Up to three files can be open at a time. You must use #1, #2, or #3 – do not use other file numbers.

You can only use colons in a print statement. Commas and semicolons will not save as in XB.

10 PRINT #1:"Now, is, the, time " will print the entire string contained in the quotes.

20 PRINT #2:"Hello":"World" or 20 PRINT #2:"Hello":PRINT #2:"World" are equivalent.

Use LINPUT for reading strings – INPUT will be treated as LINPUT if used

LINPUT will read the entire entry including any ASCII characters (like in XB)

Use INPUT for reading numbers (like in XB)

You must specify INPUT or OUTPUT when opening a peripheral for reading or writing files.

DELETE device-filename is supported but does not work with FIAD on Classic99

Error checking with peripherals

Error checking should be set up just like in XB with the following limitations:

ON ERROR line number - transfers control to the desired line number

If you are not using ON ERROR and an error is encountered:

-If running from an XB loader, the program will end and return to the line editor. No disk error message is printed.

-If running as an EA5 program the program will return to the master title screen.

RETURN line number – this only works to return to a specific line number. Do not use RETURN or RETURN NEXT

Other peripheral devices should work if they can use DISPLAY VARIABLE format.

MODIFYING THE XB LOADER

EA5 programs cannot be changed, but there are modifications you can make to the XB program created by the loader. It consists of one XB line followed by the compiled program embedded in a way that is invisible to the user. Here is the line of XB code:

10 CALL INIT :: CALL LOAD(8192,255,158):: CALL LINK("RUN")

This is a legal XB line which can be modified as desired by adding a comment or any legal XB command. You can add additional lines of code if desired, as long as you do not resequence the program..

If you want to pass a value, such as the timing for a loop, you can add to line 10 **CALL LOAD(16383,VALUE)::CALL INIT** etc. When the compiled program runs the first thing it does should be CALL PEEK(16383,VALUE) and now VALUE is available to the compiled program.

When running from XB, the compiled program is treated as a giant assembly language subroutine, invoked by CALL LINK("RUN"). When the compiled program ends or F4 is pressed, control is returned to XB. To pass a value back to XB the compiled program can CALL LOAD(16383,VALUE). You would add to the loader: 20 CALL PEEK(16383,VALUE). When the compiled program ends, control returns to XB which executes line 20 and retrieves the value placed there by the compiled program.

With CALL LOAD and CALL PEEK you can easily pass values from XB to a compiled program, from a compiled program back to XB, or from a compiled program to a chained compiled program. When the entire compiled program is in high memory, addresses from 9728 to 16383 are available. When the runtime routines are in low memory, the amount of free memory depends on how many extras (XB256, Star Wars text crawl, lower case with descenders, disk access, speech.) you use. The loader reports what addresses are safe to use.

You can also use CALL HCHAR to store a value and CALL GCHAR to retrieve it.

Besides RUN there are two other options for starting the compiled program.

CALL LINK("RUNEA") - The compiled program behaves exactly the same as if you were running from EA5. The character sets are loaded and the colors are set. The only real difference is that no F4 scan is performed, so you can't accidentally break the program, and it will run a *very* tiny bit faster.

When RUN is performed, one of the things it does is to initialize the XB256 screen2 by loading the standard character patterns and colors, and then it starts the compiled program. If you are chaining compiled programs and wish to preserve the Screen2 graphics you can:

CALL LINK("RUNV") - This starts the compiled program just like RUN without initializing the Screen2 graphics.

HOW TO CHAIN COMPILED PROGRAMS

Here's a ridiculously simple program that chains to another equally simple program:

```
10 PRINT "Program One"::RUN "DSK1.PROGRAM2"           (saved as PROGRAM1)
10 PRINT "Program Two"                                (saved as PROGRAM2)
```

Is there any way to do the same thing in a compiled program? Not directly, because RUN cannot be used within the compiled code. But there *is* a way to do it. Here is a quick demonstration of how to do this. Compile these two programs:

```
10 PRINT "Program One"                                (compiled and saved as PROGRAM1-X)
10 PRINT "Program Two"                                (compiled and saved as PROGRAM2-X)
```

Now add line 20 to the XB portion of PROGRAM1-X

```
10 CALL INIT :: CALL LOAD(8192,255,158):: CALL LINK("RUN")
```

20 RUN "DSK1.PROGRAM2-X"

When the compiled PROGRAM1 ends, it returns to XB. Since the XB program is still running, it goes on to the next instruction which is RUN "DSK1.PROGRAM2-X"

If PROGRAM1 modifies the screen2 screen, character patterns, or colors and you want to preserve them in PROGRAM2, then you should change line 10 of PROGRAM2-X from CALL LINK("RUN") to CALL LINK("RUNV"). Also, to avoid scrambling screen2, PROGRAM2-X must be saved in IV254 format. Long programs do this by default, but if it is shorter than about 13K, first save PROGRAM2-X normally, then start up XB256 and type:

```
OLD DSK1.PROGRAM2-X
CALL LINK("SAVEIV","DSK1.PROGRAM2-X")
```

ADJUSTING THE TIMING IN A GAME PROGRAM

One frustration in developing an XB program intended for compilation is that it can be rather tedious to adjust the speed of the gameplay. You try a value in a FOR/NEXT loop, save the program, compile, assemble, load, only to find that it is too fast. Then you go back to XB, try a larger value, repeat the process; find that it is still too fast, try another value, etc, etc.

If you are using XB256 to develop the game there is an easy way to streamline the process. Let's say you are working in screen2. All you have to do is set up a "hot key" to go to a diagnostic menu in screen1, where variables can be modified without disturbing screen2. When done simply return to screen2 and resume where you left off.

In the simple demo program below, lines 100-200 define a ball and put it on the screen. The ball can be moved with the ESDX keys. If you press <Fctn 1> line 160 will go to line 210 where the delay value can be modified. After pressing <Enter> control returns to the main program loop with the modified delay value.

```
100 CALL LINK("CHAR2",65,"3C7EFFFFFFF7E3C"):: R=12 :: C=16 :: DLY=1
110 CALL LINK("SCRN2")
120 CALL HCHAR(R,C,65)
130 FOR I=1 TO DLY
140 CALL KEY(0,K,S):: IF S=1 THEN 160
150 NEXT I
160 IF K=3 THEN 210
170 RN=R-(K=69)*(R>1)+(K=88)*(R<24):: CN=C-(K=83)*(C>1)+(K=68)*(C<32)
190 IF RN=R AND CN=C THEN 130
200 CALL HCHAR(R,C,32):: R=RN :: C=CN :: GOTO 120
210 CALL LINK("SCRN1"):: CALL CLEAR :: INPUT "DELAY VALUE? ":DLY :: GOTO 110
```

SPEECH

Speech has been added to the compiler. CALL SPGET works exactly like it does in XB. There are some minor differences in CALL SAY. The syntax is a bit more restrictive. Do not use leading spaces; only use one space between words; and do not append punctuation to words.

Unlike in XB, .(period) +(positive) and -(negative) are pronounced.

Commas can be used for a short pause both in XB and compiled like this: CALL SAY("HELLO , , THERE"). More than one comma can be used for a longer pause.

If a word is not found in the speech synthesizer's vocabulary, Extended BASIC will sound out the letters of the word. The compiler simply skips the word. If you want to say the letters "A B C" you should put spaces between the letters. CALL SAY("A B C") works the same in XB and compiled.

An undocumented feature of XB is the use of # to consider a phrase as one word. Neither TEXAS nor INSTRUMENTS is in the speech synthesizer's vocabulary, but TEXAS INSTRUMENTS is. It can be spoken with CALL SAY("#TEXAS INSTRUMENTS")

This also works with CALL SPGET("#TEXAS INSTRUMENTS",A\$).

Do not use any of the following as a name for a user subprogram:

The letters NC, NV, NA, SC, SV, SA, L followed by a number, or any of the names below:

ABS	CLLADR	CWRIT1	EA5WS	GPBUFF	INPUT3	NEXT	PRNTN8	SCPXS8	SIZLTH	STAR3	VREAD1
ACCEP1	CLOSE	CWRIT2	EAINTE	GPLCHR	INPUT4	NEXT1	PRNTN9	SCPXSB	SLIST1	STAR4	VSBB4
ACCEP2	CLOSE2	CWRIT3	EARLRT	GPLLNK	INPUT5	NEXT10	PUTSTK	SCPXU2	SLIST2	STAR5	VSBR
ACCEPT	CLRLN	CWRIT8	EARLYC	GPLWS	INPUT6	NEXT11	QMARK	SCPXU3	SLIST3	STAR6	VSBR1
ACCSCP	CLRLN1	CWRITE	ELSS	GR4	INPUT7	NEXT12	RAND1	SCPXU4	SLIST4	STAR7	VSBR2
ADD	CLRLN2	CYAN	ELSS1	GR4LB	INPUT8	NEXT13	RAND2	SCPXU5	SLIST5	STAR8	VSBBW
AMATCH	CLRSC1	DATPNT	ENDCC	GR6	INPUT9	NEXT2	RAND3	SCPXUP	SLOFF	STAR9	VSBBW2
AND	CLRSCN	DELAY	ENDIF	GSTAT	INPUTN	NEXTSP	RAND4	SCREE1	SLOFF1	STKPNT	VSBBW96
ASC	CLT	DELAY1	EOF	GTAR1A	INT	NOCO11	RAND5	SCREE2	SLOFF2	STOP	VSCR1A
ASTRN1	CLT1	DELAY2	EOF1	GTAR1B	INVID	NOCOIN	RANDBK	SCREEN	SLP2	STRN	VSCR1Z
ASTRN2	CMPAR1	DELS1A	EOF2	GTPABA	INVID1	NOPLAY	RDSCR1	SCRENE	SLP2A	STRPAD	VSCR2A
ASTRNG	CMPAR2	DELS1P	EOF3	GTSPNO	INVTX1	NOT	RDSCR2	SCRLA2	SNDOFF	STRS	VSCR2X
AT	CMPAR3	DELS1P2	EORT	GXMLAD	INVTXS	NOWNDW	READ	SCRLAT	SOUND	STRST1	VSCR3A
AT1	CMPAR4	DELS1P	ERRLN	H0360	IRND	NULLST	READ2	SCRLBK	SOUND1	STRST2	VSCR1L
AT1A	CMPAR5	DERRLN	ERRXB	H10	JOYST	NXTPHR	READBK	SCRLDN	SOUND2	STRST3	VSCR1L2
AT2	CMPAR6	DFWND1	ERROR	H2320	JSTADR	NXTSTR	READER	SCRLF1	SOUND3	STRST4	VSCR1LB
AT3	CMPAR7	DFWNDW	ERROR1	H2C00	KEY	OLDCHR	READSP	SCRLF2	SOUND4	STRSTR	VSCR1M
AT4	CMPAR8	DIRECT	ERROR5	H4000	KEY1	OLDINT	RESTO1	SCRLF4	SOUND5	SUBEN1	VSCR1M2
ATPNTR	CMPARE	DISP3B	ERRRPT	H8000	KEY2	ONE	RESTO2	SCRLFS	SOUND6	SUBEN2	VSCR1M4
BACK	CNE	DISP3C	FAC	HCHAR	KEY3	ONGOS1	RESTOR	SCRLLF	SOUND7	SUBEN3	VSCR1MU
BEEP	CNS	DISP3E	FILERR	HCHAR1	KEYBP	ONGOSU	RETUR1	SCRLP	SOUND8	SUBEND	VSCR1OL
BEEP1	CNS1	DISP3F	FOR	HCHAR2	KSC1	ONGOTO	RETURN	SCRLRT	SPACE1	SUBEX1	VSCR1LAG
BKINT	CNS1A	DISP4D	FORX1	HCHAR3	KSCAN	OPEN	RGSTRS	SCRLS1	SPACES	SWPPA1	VWA
BKPDSP	CNS2	DISP4E	FORX2	HCHAR4	LASTDT	OPEN1	RND	SCRLUP	SPCHRD	SWPPAD	VWRITE
BLWPWS	CNS3	DISPL1	FORX3	HCHARX	LASTLN	OPENBK	RPTER1	SCRN1	SPCHWT	SWPSC1	VWTR
CALL	CNS7	DISPL2	FORX4	HCHARY	LCDEFS	OPTBAS	RPTER2	SCRN1A	SPCOL	SWPSC3	VWTR1
CALL1	CNS8	DISPLA	FORX5	HCHGAD	LDCLR	OR	RPTERR	SCRN1B	SPDIS1	SWPSC4	WAIT
CALL2	CNS9	DISPLY	FREEZE	HEADER	LDCLR1	OUT	RPTS	SCRN2	SPDIS2	SWPSCR	WAIT1
CALLS1	CODE	DISTA1	FRSTDT	HEXDE2	LDGADD	PAB	RPTS1	SCRN2A	SPDIST	SYNC	WAIT2
CALLS2	CODEND	DISTAN	FRSTLN	HEXDE3	LDRGST	PABADR	RPTS2	SCRN2Z	SPDOVR	SYNC1	WFRSTR
CALLSB	COIALL	DIVID1	FRSTST	HEXDEC	LEN	PATTER	RPTS5	SCRNPT	SPEAK	SYNTH	WHIGHT
CEQ	COINC	DIVID2	GARBA1	HIGH	LEN1	PEEK	RTN	SCRNRT	SPEAK1	TAB	WINDO1
CEQ1	COLON	DIVID3	GARBA2	HILIT1	LET	PEEK1	RTNAD	SCROB	SPGET	TAB1	WINDO2
CGE	COLOR	DIVID4	GARBA3	HILITE	LET1	PI	RUN	SCROB4	SPGET1	TAB2	WINDO3
CGT	COLOR1	DIVID5	GARBA4	HX0010	LET2	PLAY	RUN1	SCROB5	SPGET2	THAW	WINDOW
CHAR	COLOR2	DIVID6	GARBA5	HX0018	LIMZRO	PLYR1	RUN10	SCROLL	SPGET3	TYPE	WKSP
CHAR2	COLORA	DIVIDE	GARBA6	HX001E	LINPTN	PLYR1A	RUN2Q	SCRUP1	SPGET5	VAL	WKSP1
CHAR2A	COLORC	DLINK1	GARBAG	HX0051	LINPUT	PLYR1B	RUNEA	SCRUP2	SPGFLG	VALID	WLASTR
CHAR2B	COLORD	DLINK2	GASIZ1	HX0300	LOAD	PLYR2	RUNEA5	SCRUP3	SPINI1	VCHAR	WLCOL
CHAR2C	COMDLY	DLINK3	GASIZ2	HX6080	LOAD1	POS	RUNV	SCRUP4	SPINI2	VCHAR1	WRCOL
CHAR2E	COMMA	DLY12	GASIZE	HX8000	LOADLP	POS0	SAY	SCRUP5	SPINI3	VLDRO1	WWIDTH
CHARP1	COMMA1	DLY42	GCHAR	HX8080	LOADSP	POS1	SAY1	SEARC1	SPINIT	VLDRO2	XB255A
CHARP2	COMMA2	DONE	GET0	HXFFF0	LOCATE	POS2	SAY1A	SEARCH	SPLOC	VLDRO3	XBEA5
CHARPA	COMMA5	DONEX	GET1	IF	LOW	POS3	SAY1T	SEGS	SPLOC1	VLDRO4	XBRTN
CHARPB	COMMA6	DR3LB	GET2	IF2	MAGNIF	POSITI	SAY2	SEGS1	SPPAT	VLDROU	XBRTN1
CHARSE	COMMA7	DRCTL2	GET3	IF3	MATCH	PRIN7B	SAY3T	SEGS2	SPPAT1	VLDSTR	XBRTN2
CHARTB	COMMA8	DRCTL3	GET4	IF4	MAX	PRIN9A	SAY4T	SEGS3	SPPAT2	VMBR	XBRTN3
CHPAT2	CON	DRCTL4	GETAR	INP13A	MAX0	PRINB3	SAY5T	SEMI	SPRIT1	VMBR1	XMLRTN
CHRS	CONCA1	DRCTL5	GETAR1	INPTN1	MAX2	PRINB4	SAY6T	SETADR	SPRIT2	VMBR1A	XOR
CHRSE1	CONCA2	DSKBUF	GETAR2	INPTN2	MAX3	PRINBK	SBTRCT	SETEQ	SPRIT3	VMBR1B	XPONE1
CHRSE3	CONCA3	DSRAD1	GETARR	INPTN5	MIN	PRINT	SC1CLR	SETSI1	SPRITE	VMBR2	XPONE2
CHRSED	CONCA4	DSRADD	GETSTK	INPTNL	MINUS	PRINT2	SC1DC	SETSI2	SPRMO	VMBR5	XPONE6
CHRSRT	CONCA5	DSREND	GLINK1	INPU10	MLTPLY	PRINT3	SC2CLR	SETSI4	SPRMO1	VMBW	XPONE8
CHSET2	CONCAT	DSRLNK	GLNKWS	INPU11	MONIT3	PRINT4	SC2DC	SETSI6	SQR	VMBW1	XPONE9
CHSET3	CRAWL	DSRWS	GODSR	INPU12	MONIT4	PRINT5	SCPXD2	SETSI7	SQR1	VMBW1A	XPONEN
CHSETD	CRSPOS	DWIND	GODSR1	INPU13	MONIT5	PRINT6	SCPXDN	SETSI8	SQR2	VMBW1B	XPONEX
CHSETL	CSN	DWNROW	GODSR2	INPU14	MONITG	PRINT7	SCPXL2	SETSI9	SQR5	VMBW2	XPONEY
CHSETZ	CSN1	EA5	GODSRE	INPU4A	MONITR	PRINT8	SCPXLF	SETSIZ	SQRERR	VMBW5	XPONEZ
CLE	CSN2	EA5B	GOSUB	INPU4B	MONWS	PRINT9	SCPXR2	SGN	STAR0	VMWLP1	XTAB27
CLEAR	CSN3	EA5B1	GOSUB1	INPU5A	MOTION	PRINTN	SCPXR2	SGN1	STAR1	VMWLP2	ZERO
CLEAR1	CSN4	EA5C	GOSUB2	INPUT	NAMLEN	PRN5A	SCPXS2	SGN2	STAR10	VRD	
CLEAR2	CSN5	EA5D	GOTO	INPUT2	NBR	PRNTN1	SCPXS4	SIZE	STAR2	VREAD	

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 display "L10" (or whatever the first line number is). If successful in compiling that line it will then display "L20" 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.

The compiler will report if it was able able to successfully compile your XB program. If so it will return to the menu where you can choose 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. This is another good reason to use Classic99, because the files are in windows format and can be opened and viewed with a text editor such as Notepad. 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.RUNTIME1"
      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.

USING ASM994A WITH CLASSIC99 AND XBGDP

Be sure your computer is set up so it will show file extensions. If you do not know how to do this, do a search for “How to show file extensions in Windows 10/8/7”

Set up the Game Developer's Package as described in *Using XBGDP*. DSK1 should be the folder called JUWEL6. The runtime routines and Asm994a.exe are already in this folder. Win994a is a nice emulator for the TI99 that comes with a huge amount of cartridge and disk software. If you want to try it out, the latest version can be found at www.99er.net on the home page. It is on the left under emulation.

Because Asm994a is a windows program it does not know anything about DSK1, DSK2, etc. The most foolproof way to use it is to have the source code created by the compiler, the runtime routines and Asm994a in the same folder. Since the runtime routines and Asm994a are already in DSK1 (JUWEL6), let's leave them there and do all our initial testing from that disk.

Start by opening the JUWEL6 folder, then right click on Asm994a.exe and create a shortcut. Drag and drop the shortcut to your desktop for easy access.

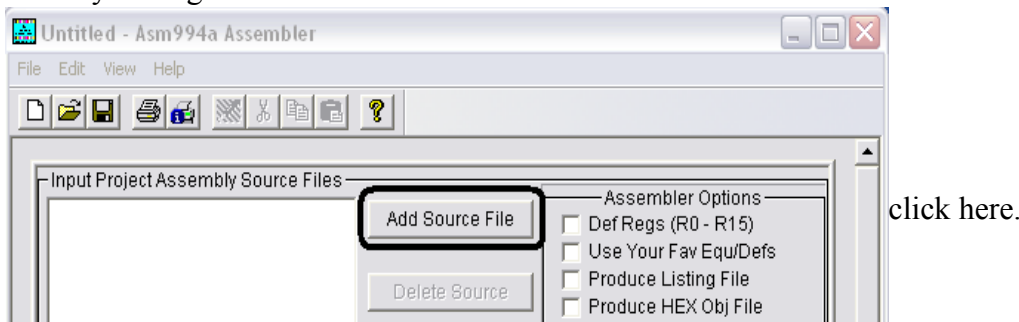
Let's test it by recompiling *HELLO*. The steps for compiling *HELLO* were described in *Using XBGDP*. Follow them up to the point in the compiler where you are asked:

Using Asm994a? Press Y and Enter, then Enter twice more to Proceed.

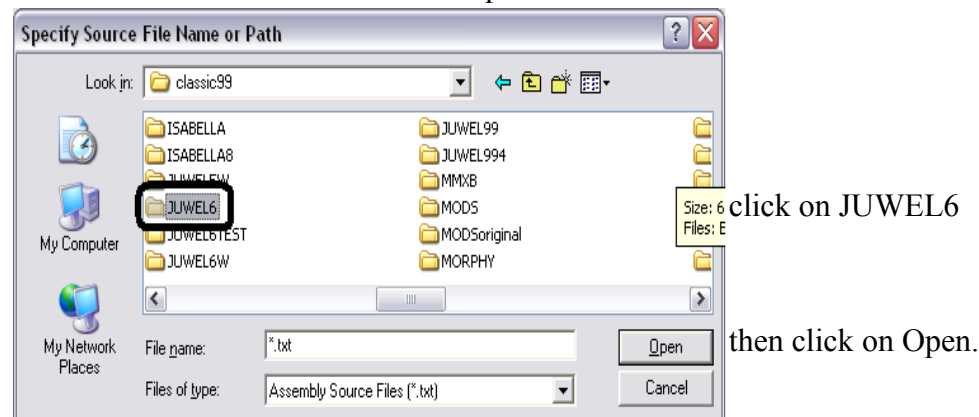
When the compiler is finished and returns to the main menu it knows you will be using Asm994a, so it bypasses the Assembler and points to Loader.

Now it's time to try out Asm994a. Windows 10 will look a little different from the XP screen shots shown here, but the steps are exactly the same.

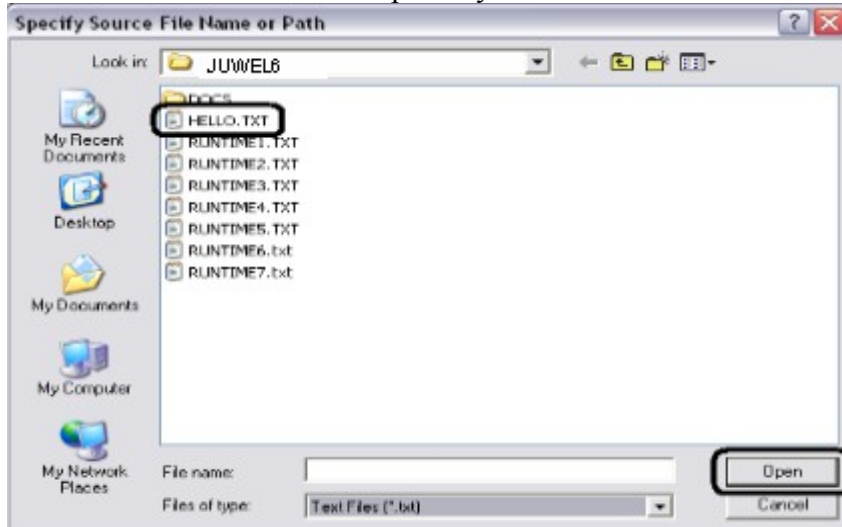
Start by adding a Source File.



Select JUWEL6 in the window that opens.



When the JUWEL6 folder is opened you will see a menu with all the text files in the folder.



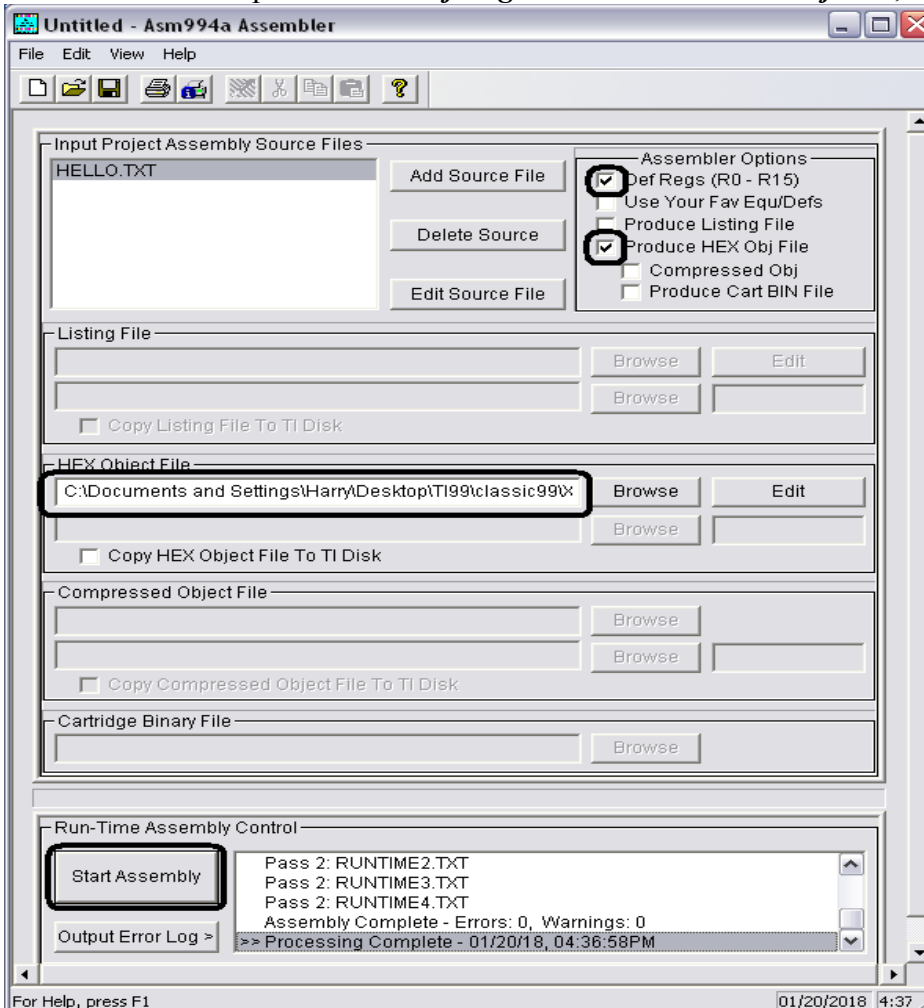
First click here.

The runtime files, compiled source files, and Asm994a should be on the same disk

then click here.

The menu closes when you choose a file name.

Under Assembler Options click **Def Regs** and **Produce HEX Obj File**, then **Start Assembly**.



Click here.

Click here.

The Object File is automatically filled in for you with the .obj extension

Click here to Start Assembly

If all goes well you see the message **Assembly Complete – Errors: 0, Warnings: 0**

Click on the Classic99 window and you are back to familiar territory.

Press Enter for the LOADER then press Enter at the prompts:

DSK1.HELLO.OBJ

CALL LINK("EA5,"DSK1.HELLO-E")

SAVE DSK1.HELLO-X

RUN Set CPU speed to normal and press Enter.

You can see there is some set up to use Asm994a the first time, and you may wonder whether it's worth it when a just few keystrokes will have the TI Assembler up and running.

Let's find out. In developing a program, usually you'd be making a number of changes to the XB program and then recompiling. Let's try changing HELLO. Break the compiled program with Fctn4, Quit, press a key, press 2 for XB. The menu comes up pointing to EXTENDED BASIC. Press Enter, and press Enter again to load HELLO. Change the text in line 10.

10 A\$=" Hello World! How are you doing???"

Type SAVE and follow the prompts to recompile. When the menu comes up pointing to LOADER don't forget that you need to assemble. Asm994a is already filled in for you and you just have to click **Start Assembly**. Then back to Classic99 to load, save and run the program.

See how much faster that is? Assembling the second time only took a few seconds.

As noted earlier, this is the easiest way to use Asm994a, but I do not particularly like the way DSK1 gets cluttered up when using this method. Once you are familiar with using Asm994a, I would suggest setting up a different disk for program development. I use DSK4 with the windows name *WorkingDisk*, but the disk number can be from 2 to 9 and the windows folder can be any name you want. You have to copy the runtime routines and Asm994a to DSK4. (WorkingDisk) This is where you will have the XB or XB256 source programs, as well as all the compiled files. Because it is in a new location, be sure to delete the old shortcut, then make a new shortcut to Asm994a and move it to your desktop.

COMBINING ASSEMBLY ROUTINES WITH COMPILED XB CODE

The latest version of the XB Game Developer's Package, *Juwel*, lets you develop hybrid programs that use assembly language support routines along with compiled BASIC code. Since the assembly routines are not part of the compiler, this offers great versatility for combining custom assembly routines with compiled XB code.

Normally in Extended BASIC, the XB program is located in the 24K “high memory” and assembly support routines are located in the 8K “low memory”. If it uses assembly support, a compiled program will use memory the same way: the compiled XB program is placed in high memory, and the assembly support is placed in low memory. The assembly routines remain virtually the same. They need some minor changes to the routines that pass string and numeric variables, and they must return to the compiled program, not to XB.

There are some minor limitations.

Since the compiled XB code can only be loaded to the 24K of high memory, you cannot use the compiler option to put the runtime routines in low memory. This means the maximum program size is smaller than when using just compiled XB.

Since all numbers in a compiled XB program must be integers, it follows that any numbers passed to or from the support routines must also be integers. Another difference is in how NUMASG or STRASG return variables to the XB program.

When running in XB, you can enclose a variable in parentheses to keep it from being returned to the calling program. That option will not work in compiled code. The parentheses are ignored and the variable will always be returned.

Finally, you must use a string constant for CALL LINK. You cannot CALL LINK(A\$,...) works properly (if there is an assembly sub named “CLS”), but 10 A\$=”CLS”::CALL LINK(A\$) will not work properly in the compiled program.

At this point, some explanation of how the compiler works is necessary. Remember that XB256 is built into the compiler. When the compiler encounters any of the XB256 CALL LINK routines, it strips out the LINK and treats it like a CALL. For example, the compiler would treat CALL LINK(“DISPLAY”,1,1,”Hello World”) as if it were CALL DISPLAY(1,1,”Hello World”). So it would seem it is not possible to use any CALL LINK that is not built into the compiler,

But there is a way around that. The subroutine names can be converted to lower case alphabetic characters. The lower case lets the compiler know it is a user CALL LINK that is not part of the compiler, and so it is treated differently. There is an assembly program (UC2LC) that converts all the assembly subprogram names in an XB program to lower case. A similar program (LC2UC) converts them back to upper case to rescue you when (not if!) you unintentionally convert them to lower case.

Another assembly program (FIXAL) will modify the assembly support routines for compatibility with the compiler. It converts the assembly subprogram names in the DEF table to lower case, modifies NUMASG, NUMREF, STRASG, STRREF, CSN, and CNS, and changes the return to XB so it returns to the compiler instead of returning to XB. It then embeds the modified assembly code into an XB loader.

As always, before trying to compile, you should always thoroughly test the XB program and its assembly support. There are a number of ways to load the assembly routines to low memory for testing. To load your own own custom assembly routines, you would use CALL LOAD(“DSKn.FILENAME”). If you are using XB 2.9 G.E.M., you can load XXB, T40XB, T80XB, the Missing Link, or MULTICOLOR XB directly from the menu. Or you can use the disk based versions of these or similar utilities. These use an XB loader with embedded code; RUN “DSKn.FILENAME” will copy the assembly routines to low memory. Once the assembly routines are loaded you can load or write the XB program.

When your hybrid XB/assembly program is thoroughly tested, it is time to create the compiled program.

Start by modifying the assembly support routines so they can work with the compiler. If you are using XXB, T40XB, T80XB, the Missing Link, or Multicolor XB, this has already been done and you can skip the next step. The files are XXBC, T40XBC, T80XBC, TMLC, and MCOLORC on the Jewel disk.

To modify your own assembly routines, with the JUWEL folder as DSK1, choose XB, then type:

NEW

CALL INIT

CALL LOAD("DSKn.ASSEMBLY.OBJ") or CALL LOAD("DSKn.ASSEMBLY-O") on a real TI99

CALL LOAD("DSK1.FIXAL")

CALL LINK("X") starts FIXAL to modify the assembly routines and creates an XB loader.

SAVE DSKn.ASSEMBLYC it is a good idea to append C so you know this is for the compiler.

Once the assembly routines are modified and saved, the compilation process is much like compiling an ordinary XB program. Here is a walk-through of how to write and compile a simple XB program for *The Missing Link*. This program computes a random angle, centers the turtle, draws a polyspiral, then repeats.

```
10 CALL LINK("CLEAR"):: ANG=RND*90+90 :: CALL LINK("PUTPEN",92,120,0)
```

```
20 FOR I=1 TO 130 :: CALL LINK("FWD",I,ANG):: NEXT I
```

```
30 GOTO 10
```

Load The Missing Link, then enter the above program and test it out. When you know it works:

SAVE DSK2.POLY

With the JUWEL disk in drive #1, quit or BYE to the color bar screen, and press 2 for Extended BASIC. When the Jewel menu is displayed, select Extended Basic.

OLD DSK2.POLY loads the polyspiral program to be compiled. Add this line.

1 CALL LINK("TML16") initializes The Missing Link for 16 colors

CALL LOAD("DSK1.UC2LC") this changes the upper case characters in the A/L

CALL LINK("X") subprogram names to lower case characters.

List the program to verify that the subprogram names are in lower case, then type SAVE.

You get the prompt "SAVE DSK2.POLY" It is a good idea to add "L" to show that it is lower case.

Keep pressing Enter at the prompts until you come to the loader. Remember that, when compiling, the RUNTIME routines cannot be in low memory.

At the prompt "Using Assembly Support? N" type Y, then Enter

LOAD COMPILED PROGRAM

Using Assembly Support? Y

Compiled file to load?

■SK2.POLYL.OBJ

You are prompted for the filename

Press Enter or input a different file

Compiled file to load?

DSK2.POLYL.OBJ

17874 bytes remaining

Assembly routines to load?

DSK1.TMLC■

You are prompted for the assembly routines

TMLC is a version of TML modified for the compiler

Assembly routines loading...

DSK1.TMLC

Press Enter:

CALL LOAD(-31868,0,0,0,0)::
RUN "DSK1.L24AL2" ■

Press Enter when the cursor appears

```

Compiled file loaded.
DSK2.POLYL.OBJ
17874 bytes remaining

Assembly routines loaded.
DSK1.TMLC

Options for saving program
Press Fctn 3+Enter to bypass

Saved as EA5
DSK2.POLYL-E

Save as XB
SAVE DSK2.POLYL-X

```

EA5 program was saved as POLY-E

Press Enter twice to save as XB program

Now you can type RUN to test it out.

One more step is needed if you want to load the program from XB. The XB program must be in two parts. You just saved the compiled XB part as POLY-X. Now let's create the other part.

OLD DSK1.TMLC loads the TML support utilities for the compiler.

Modify line 10:

```
10 CALL INIT :: CALL LOAD(8192,255,172):: CALL LINK("X"):: RUN "DSK2.POLY-X"
```

Then SAVE DSK2.POLY-C

When you run POLY-C it loads the TMLC routines to low memory, then loads and runs POLY-X.

INITIALIZING THE MISSING LINK, T40XB, AND T80XB

For The Missing Link, the first line executed should be CALL LINK("TML16") or CALL LINK("TML2"). This initializes the bit mapped screen, does CALL FILES(2), and enters the bitmapped mode. You can toggle between the graphics modes with CALL LINK("BITMAP") and CALL LINK("GRAFIX") Fctn 4 has been disabled to avoid returning to XB in the wrong graphics mode. If you want the program to return to XB when the program ends, the last line executed before end should be CALL LINK("GRAFIX") to set the normal XB graphics mode.

For T40XB or T80XB, the first line executed should be CALL LINK("T40XB") or CALL LINK("T80XB"), These routines initialize the screen but do not go into the text mode. Your program can then select the text mode with CALL LINK("T40") or CALL LINK("T80"), or select the graphics mode with CALL LINK("G32"). If the program breaks or ends, it will return to XB in the normal graphics mode.

MORE INFORMATION ABOUT FIXAL

FIXAL looks for the following byte sequences. If found they will be modified for compatibility with the compiler.

>0460,>006A	B @>006A	back to XB
>0460,>0070	B @>0070	back to XB
>0420,>2018,>12B8	BLWP @XMLLNK	CFI
>0420,>2018,>0020	BLWP @XMLLNK	CIF
>0420,>2018,>11AE	BLWP @XMLLNK	CSN
>0420,>2018,>0006	BLWP @XMLLNK	CNS
>0420,>2018,>0026	BLWP @XMLLNK	SCROLL

XMLLNK should only be accessed with BLWP @XMLLNK. Coding in any other way will be ignored by FIXAL. However, this offers a workaround if an assembly subprogram needs to use one of the unmodified XMLLNK routines. For example, the disk catalog program in T40XB and T80XB reads a floating point number from disk, converts to integer using CFI, and prints it on the screen. The following code works in XB, but the modifications by FIXAL bypass CFI. This keeps it from working properly when compiled.

```
BLWP @XMLLNK
DATA >12B8          cfi
```

I coded this differently so that FIXAL does not find and modify it. Now it works as expected.

```
LI R8,XMLLNK
BLWP *R8
DATA >12B8          cfi
```

Do not use B *R11 to return to XB. It must be done with B @>006A or B @>0070.

```
LWPI >83E0
B @>006A
```

FIXAL changes this so the return goes to the compiled code. If you had some reason to return to the XB interpreter instead of compiled XB, you can modify the code so that FIXAL does not find and modify it.

```
LWPI >83E0
LI R1,>006A
B *R1
```

Any XB program is erased when FIXAL creates the loader program, so be sure to save your work.

DEALING WITH ERRORS IN ASSEMBLY ROUTINES

Compiled XB code cannot report errors, but that can be done when using assembly routines. The standard BLWP @ERR gives some information, but it is best to use a custom error handling routine that works with EA5 programs and lets you use ON ERROR line number. This is especially useful when the user must enter the name of a file. If the name is wrong the error routine can print a message, then return. This is much better than simply ending the program when there is an error. You must use RETURN line number. The error handling routine described below is already built into T40XBC, T80XBC, and TMLC

The Missing Link program below asks for the name of a picture file to load. If there is an error, it will issue an error message, then return to line 100 where it prompts for the file name.

```
100 ON ERROR 200 :: CALL LINK("INPUT",1,1,P$):: CALL LINK("LOADP",P$)
120 CALL KEY(0,K,S):: IF S<1 THEN 120 ELSE CALL LINK("CLEAR"):: CALL LINK("COLOR",2,8):: GOTO 100
200 CALL LINK("CLEAR"):: CALL LINK("PRINT",100,1,"BAD FILE NAME, TRY AGAIN"):: RETURN 100
```

This works fine in XB. To enable this type of error reporting in compiled code, three steps must be taken.

1 – the ERR vector at >2034 must point to a new workspace and the new error handling code. I added the code below to the subprogram used in the first line of the program to initialize the video memory.

```
LI R0,BUFFER+100      ERR cannot use same WS as VSBW
MOV R0,@>2034
LI R0,ERRORX          change ERR to point to my error routine
MOV R0,@>2036
```

2 – Where BLWP @ERR is used, it must be changed to use the messages in the new error handling routine. Search for BLWP @ERR and put the new error message into R0.

```
RPTERR      LI R0,IOERR          was LI R0,>2200  FILE ERROR
            BLWP @ERR
```

3 – New error handling code must be added to the assembly routines. Below is the error handling routine used in T40XB, T80XB and TML. This prints an error message unless you are using ON ERROR line number.

```
ERRORX      MOV @>8314,R8        Move R10 of compiler WS into R8. LNKPTS
```

	MOV @4(R8),R8		3rd word in this table into R8 contains ERRLN
	C @2(R8),*R8		does ERRLN=DERRLN (default error = end of program)
	JEQ ERRORN		yes, report error and end program
ERRORY	MOV *R8,@>831A		ON ERROR was used, get line number into R13
	MOV @2(R8),*R8	o	n error is only used once, restore default
	CLR @>83AA		compiler wants this to be zero
*here TML, T40XB, and T80XB have code to restore the graphics mode	B @>006A		FIXAL changes this to return to compiler
ERRORN	MOV *R13,R6		mov R0 calling to R6
PRERRM	LI R0,>02A1		lower left of screen
PRERR1	INC R0		
	MOVB *R6+,R1		
	JEQ ERRORY		
	AI R1,>6000		
	BLWP @>2020		VSBW print error message
	JMP PRERR1		
IOERR TEXT 'I/O ERROR'			
	BYTE 0		
BVERR TEXT 'BAD VALUE'			
	BYTE 0		
	EVEN		

ADDING ASSEMBLY SUBROUTINES DIRECTLY TO THE COMPILER

The method described above is the easiest way to combine assembly routines with your compiled code. The FIXAL utility will automatically convert the assembly code to be compatible with the compiler, and you do not need to know any assembly to use it. You can use assembly routines like *The Missing Link* or *T40XB* without having to know anything about the code.

However, for more advanced users it is possible to add your own assembly code directly to the runtime routines used by the compiler. One advantage is compactness. If your routines only require 50 bytes, then that is all the memory that will be used. Also, subroutines run this way can be somewhat faster because the compiled code knows where they are without having to look up their address in a table. Modifying the program for compatibility is not automated; it must be done by the user. But if you wrote the assembly subs, you should have the ability to make the necessary changes.

1 – Write the assembly support routines and test them out in XB to be sure they work. If you are using an extension such as *The Missing Link* or *XB256*, there is no room in low memory for additional subroutines. The solution is to use HMLOADER to include them as embedded code in high memory where they can be combined with an XB program. See the HMLOADER docs for more information.

Here is an assembly language subroutine that takes a string and reverses the letters so the first will be last, etc. Notice that this uses STRREF to read the string into the assembly subroutine, and STRASG to pass the string back to XB.

*FLIP XB version

REF STRREF	can only use REF in XB2.9 G.E.M.
REF STRASG	other XB versions must use EQU
DEF FLIP	

*CALL LINK("FLIP",A\$,B\$) takes A\$ and reverses it, returns as B\$

FLIP	LWPI WKSP	
	CLR R0	
	LI R1,1	1st parameter in call link
	LI R2,GPBUFF	put the string in GPBUFF
	SETO *R2	longest possible string is 255
	BLWP @STRREF	string is in GPBUFF
	MOV R2,R3	
	MOVB *R2+,R4	length to r4 and r2 points to first char in string
	SRL R4,8	length to lsb
	A R4,R3	now r3 points to last char in string
FLIPLP	MOVB *R3,R4	
	MOVB *R2,*R3	
	MOVB R4,*R2+	
	DEC R3	
	C R2,R3	
	JLE FLIPLP	
	INC R1	2nd parameter
	LI R2,GPBUFF	
	BLWP @STRASG	send the string back
	LWPI >83E0	
	B @>006A	
WKSP	BSS 32	
GPBUFF	BSS 256	

Step 2 - When the code has been tested it must be modified for compatibility with the compiler.

When running as part of the runtime routines, your code starts with the workspace set to >8300

WKSP is >8300 Main workspace used by compiler. R13, R14, R15 are reserved for the compiler.

BLWPWS is >8320 Can be used for your own BLWP subs.

Code and addresses used by the compiler can also be used by your custom code.

GPBUFF is a >0118 byte long general purpose buffer.

BLWP @SCROLL scrolls the screen

BLWP @KSCAN, VWTR, VMBR, VMBW, GPLLNK, and DSRLNK can all be used normally.

BL @VSBW, VSBW, and VSBW96 (VSBW with >60 or 96 offset) are now BL routines.

They work just like the BLWP versions, but are faster.

Because the compiler only uses integers, CFI and CIF are not needed

The main difference between compiled code and XB is in how the parameter list is handled.

In XB you use NUMREF, NUMASG, STRREF, STRASG to transfer parameters from or to the list.

In the compiler you would use:

BL @GET1 puts next parameter in the list into R6

BL @GET2 puts next 2 parameters in the list into R5 and R6

BL @GET3 puts next 3 parameters in the list into R4, R5 and R6

BL @GET4 puts next 4 parameters in the list into R3, R4, R5 and R6

For numeric parameters, the registers will point directly to the number.

If R6 points to the number, you can simply MOV *R6 to another register and go from there.

For strings, the registers will *point to a pointer* to the string, so one more step is necessary.

If the pointer is in R6, then MOV *R6,R6. Now R6 points to the length byte at the beginning of the string. If necessary, you can move the string to GPBUFF for the equivalent of STRREF.

This is demonstrated in the converted FLIP routine on the next page.

BLWP @STRSTR is used to assign a string. This is the equivalent of STRASG in Extended BASIC.

R0 should point to the length byte at the beginning of the string (usually this will be GPBUFF)

R1 should point to the string address provided after GET1, GET2, GET3, or GET4.

STRSTR is demonstrated in the converted FLIP routine on the next page.

LI R0,GPBUFF

MOV R6,R1 send to the string pointed to by R6

BLWP @STRSTR send the string back (STRSTR = Store String)

To determine if a parameter is a string or a numeric value:

BL @ASTRNG returns EQ if R6 points to a string, NE if R6 points to number. You must use R6

Your code must read all the parameters in the list. It cannot read less, nor more.

If there were 6 parameters you could BL @GET1 six times, or BL @GET4, then BL @GET2, etc.

Some routines do not have a fixed number of parameters in the list. If so, you can:

C *R13,R15 See if next word is part of the parameter list or is an instruction

JLT MORE if LT then there is at least 1 more unread parameter in the list.

This is demonstrated in the MPY routine on the next page.

B @RTN returns to the compiler from the subroutine. Be sure the workspace is set to >8300.

This is equivalent to LWPI >83E0, followed by B @>006A or B *R11 in Extended BASIC.

With this information we can convert the FLIP routine to be compatible with the compiler.

*CALL LINK("FLIP",A\$,B\$) modified for the compiler

```
FLIP      BL @GET2          r5 points to pointer to A$, r6 points to pointer to B$
          MOV *R5,R5        now r5 points to the actual string
```

*now we have to move the string to gpbuff (This does the same as STRREF)

```
          MOVB *R5,R1        length byte
          SRL R1,8           length is in lsb
          INC R1             +1 to account for the length byte
          LI R2,GPBUFF       will put string into gpbuff
FLIP1     MOVB *R5+,*R2+
          DEC R1
          JNE FLIP1
```

*now the string is in gpbuff

```
          LI R2,GPBUFF
          MOV R2,R3
          MOVB *R2+,R4       length to r4 and r2 points to first char in string
          SRL R4,8           length to lsb
          A R4,R3           now r3 points to last char in string
FLIPLP    MOVB *R3,R4
          MOVB *R2,*R3
          MOVB R4,*R2+
          DEC R3
          C R2,R3
          JLE FLIPLP

          LI R0,GPBUFF
          MOV R6,R1
          BLWP @STRSTR       send the string back (equivalent of STRASG)
          B @RTN             and go back
```

Here is another assembly sub designed to work with the compiler. This just multiplies two numbers and returns the product. It can do this multiple times in a single CALL. This routine is not useful, but it is meant to show how to transfer numeric values between the compiled code and the subroutine.

*CALL LINK("MPY",A,B,C.....) for the compiler

*Multiplies the parameters A x B and returns product to C

*The only purpose of this is to show how to retrieve and assign numeric values

```
*      TEXT 'MULTIPLY'      uncomment this line and the text can help find the code when you need to debug
MPY     BL @GET3            r4, r5, r6 point to A,B,C

          MOV *R4,R1        get A (equivalent to NUMREF)
          MOV *R5,R2        get B (equivalent to NUMREF)
          MPY R1,R2         multiply them
          MOV R3,*R6        store the product in C (equivalent to NUMASG)

          C *R13,R15       See if next word is a number or an instruction
          JLT MPY          do it again if there are more numbers

          B @RTN           and go back
```

I combined FLIP and MPY into one file and saved it as DSK3.FLIPC.TXT

Now we have assembly source code that should be compatible with the compiler.

Step 3 – The “FLIP” and “MPY” subroutines need to be in the compiler's list of CALLs. When the compiler encounters a CALL not in its list, it assumes it is a user defined subprogram and it will not be encoded correctly. To be handled properly, they must be added to DATA statements starting at lines 10000+, followed by a single null string. Then the compiler will recognize them and encode them properly. Up to 32 subroutines can be used.

```
10000 DATA FLIP,MPY
```

```
10005 DATA ""
```

If the time comes when you do not need them, you can remove FLIP and MPY from the compiler, although leaving them there should cause no problems.

Lines 10020+ can contain the assembly source code files you wish to load.

```
10020 DATA DSK2.FLIPC.TXT          or FLIPC.TXT if you are using Asm994a
```

```
10025 DATA ""                    as before, these must be followed by a single null string.
```

When the compiler runs, you are asked “Add more runtime routines? Y/N” Enter Y

Each of the filenames in lines 10020+ is displayed. You can press Enter to include it, or clear it with Fctn 3, then press Enter to not include it.

When the compiler comes to the null string at the end of the files in lines 10020+, a generic file name is then displayed. This gives you a chance to include other files that are not in the DATA statements. You can add as many as you want. Enter a blank line when done.

SAVE DSK1.COMPILER to save your changes.

My XB program for testing FLIP and MPY was:

```
5 INPUT A$
```

```
6 IF A$="" THEN 50
```

```
10 CALL LINK("FLIP",A$,B$):: PRINT B$
```

```
20 GOTO 5
```

```
50 A=3 :: B=17 :: D=99 :: E=54
```

```
55 CALL LINK("MPY",A,B,C,D,E,F)
```

```
60 PRINT C,F
```

At this point you should be able to save the XB program, compile, assemble and load normally. The source code the compiler creates will load the necessary RUNTIME routines and then include FLIPC.TXT

Important! If you are using assembly routines in low memory such as *The Missing Link* or *T40XB*, you must change the upper case CALL LINKs to lower case using UC2LC as described in the previous section. But the CALL LINKs that have been made part of the compiler must be capital letters. So you will have to go through the program and convert those back to capital letters. A text editor is a big help for that.

RUNNING COMPILED & ASSEMBLY PROGRAMS USING AN XB MENU

There is a trend these days to save compiled programs as EA5 and convert them into cartridges. This makes a nice, neat package and works fine for self contained programs. The loader gives you the option to save a program as EA5.

CHAINING PROGRAMS

On the other hand, saving a compiled program as XB has other advantages. Remember that, as far as XB is concerned, the compiled program is just a large assembly language subroutine. Because of this, it is possible to write a menu program that loads from disk and runs in XB, and use it to select different compiled programs on the disk. When a compiled program ends it returns back to the XB interpreter. If the next statement in the XB program is RUN "DSK1.MENU" then the menu program is be loaded and you can use it to choose a different compiled program.

Below is a short tutorial showing how to do this. This demo runs two programs that use compiled TML. One program draws a polyspiral; the other draws circles using CALL LINK("PR") to make an interesting graphics demo. DSK1 is used throughout.

First the menu program:

```
10 CALL CLEAR :: PRINT "PRESS:":"1 - POLYSPIRAL":"2 - CIRCLES WITH PENREVERSE"
20 CALL KEY(0,K,S):: IF K=49 THEN CALL CLEAR :: RUN "DSK1.POLY-X"
21 IF K=50 THEN CALL CLEAR :: RUN "DSK1.CIRCLES4-X"
22 GOTO 20
```

Save this as DSK1.MENU

Here are the two programs for The Missing Link. The first is POLY and the second is CIRCLES4.

```
10 !CALL LINK("tml16")
20 FOR I=1 TO 500 :: CALL LINK("FWD",I,123):: NEXT I
30 CALL LINK("GRAFIX")

80 !CALL LINK("tml16")
100 CALL LINK("PR"):: FOR K=1 TO 2 :: FOR C=1 TO 90 STEP 1 :: FOR I=-1 TO 1 STEP 2 :: FOR J=-1 TO 1 STEP 2 ::
CALL LINK("CIRCLE",96+I*C,120+J*C,C,0):: NEXT J :: NEXT I :: NEXT C :: NEXT K :: CALL LINK("PD")
120 CALL LINK("GRAFIX")
```

Two things are worth noting here:

The first line of each program is !CALL LINK("tml16") As a comment, this has no effect when testing under The Missing Link. It is uncommented before compiling so it can initialize TML.

The CALL LINK("GRAFIX") at the end of each program will return to the normal XB graphics mode so the XB menu program can run properly. For T40XB or T80XB this would be CALL LINK("G32").

These programs are compiled as described above on pages 17-19. To recap:

With the XBGDP disk in drive #1, go to the color bar screen. (Quit or BYE)

Press 2 for XB, then select XB from the Jewel menu.

Type OLD DSK1.POLY or OLD DSK1.CIRCLES4 to load the program.

Uncomment the first line so the compiled program knows to initialize TML

Type CALL LOAD("DSK1.UC2LC") then CALL LINK("X") - converts assembly names to lower case.

Type SAVE and the file name should be filled in automatically.

Press ENTER repeatedly to compile and assemble.

When you come to the loader select Y for using assembly routines.

The file name of the compiled program should be filled in automatically

Enter the file name of the assembly support. In this case it is DSK1.TMLC

Keep pressing ENTER to load and save as EA5 and the XB programs POLY-X and CIRCLES4.

When the cursor returns, type RUN to test that the program works as expected.

Now let's put it all together

We need to modify TMLC by adding two statements:

```
10 CALL INIT::CALL LOAD(8192,255,172):: CALL LINK("X")::RUN "DSK1.MENU"
```

Then save it to a different file name: SAVE "DSK1.LOADTMLC" or SAVE "DSK1.LOAD"

This does CALL INIT, loads the compiled version of The Missing Link into low memory, then runs the MENU program. *Once this has run, the Missing Link routines are in low memory. They do not have to be loaded again as long as you do not CALL INIT or otherwise do something to overwrite them.*

Now the two compiled programs, POLY-X and CIRCLES4-X need to be modified by adding a comment and a statement:

```
1 !POLYSPIRAL or CIRCLES4 This will help you identify what the program is.
```

```
10 CALL LOAD(8192,255,158):: CALL LINK("RUN"):: RUN "DSK1.MENU"
```

Save the modified program as POLY-X or CIRCLES4-X

Now to test:

RUN "DSK1.LOADTMLC". This loads The Missing Link routines, then runs the menu program.

In the menu program, press 1 for POLYSPIRAL or 2 for CIRCLES WITH PEN REVERSE. The compiled program is loaded and runs. When the compiled program is finished it will return to XB, where RUN "DSK1.MENU" is performed to reload the menu program.

USING RUNTIME ROUTINES FROM THE ORIGINAL COMPILER

If your program is written in TI BASIC you can now use the runtime routines that were part of the original TI BASIC compiler. The advantage is that the program created is considerably smaller, plus it may run a bit faster due to less overhead in the interrupt routine. The big disadvantage is that it only supports TI BASIC instructions (with a few additions from XB), and there have been no improvements for many years.

Most users will not want to use this, so it is turned off by default. To enable this option type:

OLD DSK1.COMPILE

uncomment line 230

SAVE DSK1.COMPILE

Now when the compiler runs you can press "Y" when prompted "Use TI BASIC runtime?" Default for this prompt is always "N".

The procedure for compiling a program is identical to the current version described above and in *Using XB GDP*. The limitations of this earlier compiler are described below, taken verbatim from the original manual. Do *not* put the runtime routines in low memory!

The BASIC compiler is able to compile many TI BASIC programs, although sometimes minor changes have to be made to the BASIC code. Some examples:

32767+1=32768 in BASIC

32767+1=-32768 in the compiled code

10 IF RND>.5 THEN 100 ELSE 200 won't work properly in the compiled code.

Instead, use 10 IF INT(RND*2)=1 THEN 100 ELSE 200 which gives either a 0 or a 1 in both BASIC and the compiled code.

200*200=40000 in BASIC; -25536 in compiled code.

Remember that the compiler only works with integer numbers from -32768 to 32767. 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.

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 BASIC 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 BASIC or compiled code.

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

As in XB, simple multiple statement lines can be used, separating the statements with the double colon

CALL LINK("RUN") - same as RUN in XB Cannot use RUN or RUN line # within a program.

CALL LINK("CON") - same as CON in XB

<FCTN 4> breaks the program as in XB except during INPUT.

All relational operators work the same as in BX. These include < > = <> <= >=

Arithmetic operators all work as they do in BX. Exponentiation (^) not supported. Remember

that dividing 5/2 will give 2, not 2.5. You can use INT in the BASIC program when dividing (for example INT(5/2)) to be certain that BASIC and the compiler give the same results.

Logical operators from XB have been included: NOT; AND; XOR; OR

LET - optional

REM - All remarks will be removed from the compiled program, but you can GOTO a REM statement just like in BX. Use of REM will not increase the size of the compiled program.

! - the exclamation point REM from XB has been included.

END

STOP

GOTO

ON-GOTO

IF-THEN-ELSE - XB style of IF-THEN-ELSE *is now* supported, with the same minor restrictions found in the XB compiler.

FOR-TO-STEP - step optional; +1 assumed

NEXT

INPUT - Can use the optional prompt, but can input only 1 string or number per INPUT statement.

READ

DATA (Do not GOTO a DATA statement!)

RESTORE

PRINT - works like TI BASIC, including TAB and the print separators ;,:

DISPLAY - equivalent of PRINT.

CALL CLEAR

CALL COLOR - expanded to work like XB except for color of sprites.

CALL SCREEN

CALL CHAR - expanded to work like XB.

CALL HCHAR

CALL VCHAR

CALL SOUND - cannot handle frequencies greater than 32767. (Neither can my ears!)

CALL GCHAR

CALL KEY

CALL JOYST

ABS

INT

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 in BX.

SGN

SQR - gives same number as INT(SQR(N)) in BX

ASC

CHR\$

LEN

POS

SEG\$

STR\$

VAL

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

DIM is optional 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 would have to split up the statement something like this: X=B(7)::Q=A(X)

GOSUB

RETURN

ON-GOSUB

NOT SUPPORTED:

DEF

ATN

COS

EXP

LOG

SIN

TAN

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")

TRACE

UNTRACE

EDIT