

XB32K

by Harry Wilhelm – April 2019

The memory expansion for the TI99/4A provides 32K of cpu memory. This is divided into two segments; 24K of “high memory” from >A000 to >FFFF, and 8K of “low memory” from >2000 to >3FFF. TI Extended BASIC can only use the 24K segment. The 8K segment is normally used for assembly subroutines such as XB256. But if you are just writing in XB and do not want to use assembly subroutines, then that 8K segment goes to waste. XB32K provides a way to access this extra memory so you can use the entire 32K for an XB program.

In order to do this, your XB program must be divided into two segments and each segment saved separately to disk. If you have selected Extended BASIC from the compiler menu the code for autosave will be running in low memory and must be turned off. You can do that by pressing F3 at the OLD DSK prompt, or you can enter CALL LINK(“OFF”).

The segment destined for the 8K low memory should have the highest line numbers. This program can be no larger than 8026 bytes. Put another way, if you load the program and type SIZE there must be at least 16462 bytes of program space free. Program lines loaded here are not so easy to edit, so this code should be well tested. This is a perfect place for subprograms as will be shown later. Let's call this segment PART1.

The main program will be in the 24K high memory and it should have the lowest line numbers. Let's call this segment PART2. These lines of code can be edited normally. It is important that the highest line number in PART2 is lower than the lowest line number in PART1.

Important information for TIdBit (TI BASIC TRANSLATOR) users: when dividing the program into 2 parts, you need to make sure that all label references are within the same section as the label declaration, otherwise the label references will not be replaced by the appropriate line number, leading to unpredictable results when you compile and run the program.

Now you have to create an XB program that can load PART1 into low memory. Enter the following:
CALL INIT::CALL LOAD(“DSK1.XB32K.OBJ”) - This loads some assembly routines and sets pointers in the scratchpad which force XB to load a program into low memory.

OLD DSK1.PART1 - the first segment of the program is loaded into low memory. (You cannot use the paste XB feature of Classic99 for this). Now that the program is in low memory we need to create an XB loader that can save these lines of code and reload them back to low memory when needed.

This is a little unorthodox, so pay attention. Type the name of the second segment of the program.

DSK1.PART2A – then press F4. When you mess up and press Enter you get an error message. Don't panic, just type DSK1.PART2A again and press F4 this time. The objective here is to put the filename for the second segment at row 23, column 1.

CALL LINK(“X”) - this runs an assembly routine that creates an XB loader containing PART1.

The visible part of the loader consists of the XB line:

```
10 CALL INIT::CALL LOAD(8192,255,200)::CALL LINK(“X”)::RUN “DSK1.PART2A”
```

The 15 characters at row 23, column 1 were read and used as the file name for the second segment. If you prefer, you can change the file name directly by editing line 10 of the program. The program created is in low memory. XB doesn't think it should be there and so will not let you run or edit the program, but it is nice enough to allow you save it normally. Let's do it.

SAVE DSK1.PART1A – Notice that I appended an “A” to avoid overwriting the original file PART1 in case you need to make changes.

When this program is loaded and run, XB thinks it is a normal program and loads it into high memory. Then assembly instructions embedded in the loader copy all the code over to low memory where it was originally. The loader then loads and runs the second segment (DSK1.PART2A)

Now for PART2. Type:

NEW

OLD DSK1.PART2 - the high memory segment of the program is now in the 24K memory where XB usually keeps programs. The low memory part if the program is still sitting in the 8K of low memory, but the program just loaded knows nothing about this. We have to merge the line number table from the low memory program with the program in high memory.

CALL LINK("X") - now the line number tables have been merged and the program can find the lines of code stored in low memory. Now save it:

SAVE DSK1.PART2A – I appended an “A” to avoid overwriting PART2.

SAVE DSK1.PARTALL-M,MERGE – Only do this if you intend to compile the program later.

Here is where the magic happens. When you RUN "DSK1.PART1A" the first segment of the program is loaded into high memory as usual. When it runs, it copies itself back into low memory where it started out originally. Then PART2A is loaded and when it runs everything behaves normally even though much of the program is in the "wrong" memory location. Remember that the program is in two parts. PART1A has to run before PART2A runs, otherwise the program is not all there.

EDITING

Although you can list all the lines of code in the program, the code in PART1 is off limits. If you have to edit it you must go back to the original program to make any changes and then repeat the steps above.

It is possible to edit PART2. You can add lines to it in the usual manner. If you change any line in PART2 then XB adjusts the line number pointers throughout the program, including the pointers to PART1 in low memory. These pointers *must* be reset – you can do this with CALL LINK("X"). *Do not* list or run the program until you have reset the pointers. When in doubt enter CALL LINK("X") and be sure to make frequent backup copies.

You can easily hide the low memory code this way:

CALL LOAD(8193,196)::CALL LINK("X") - The lines of code in low memory are now hidden. At this point you can RESequence safely. CALL LINK("X") a second time will make the low memory code available again. Remember that if you resequence this way, it only applies to the segment of the program in high memory. Any references to lines contained in low memory, such as GOTO 5000, will become GOTO 32767 .

This brings us to subprograms. These are self contained mini-programs that are called by name, not by line number like most other XB instructions. If you put them in low memory you can resequence the main program as described above. Even though the code in low memory was not resequenced it will not matter because the subprograms are called by name. Naturally their line numbers should be high enough so the main program will not encroach on them. Because code in low memory cannot be edited, be sure they are thoroughly tested. Be sure to save it again when the program is debugged.

COMPILING

A program using XB32K can be compiled like a normal program. For this example we will use the file names in the example above. Get the program debugged and running properly from XB. Save the program as PART2A. Then save it in merge format: SAVE DSK1.PARTALL-M,MERGE. This will save both segments of the program as one file. The resulting file may be too big for XB to load into high memory, but the compiler doesn't care about that and will treat it as a normal XB program. Then quit and choose XB. This brings up the compiler menu. Choose the compiler. You will have to enter the file name by hand. From there, compiling, assembling and loading is done in the usual manner.

PRACTICE PROGRAM

To get you started, here is a very simple example to practice on.

```
10 PRINT "IN MAIN PROGRAM"      \
20 CALL STEST1
30 PRINT "IN MAIN PROGRAM"      Save this as PART2
40 CALL STEST2
50 PRINT "IN MAIN PROGRAM"
60 END                            /

1000 SUB STEST1                  \
1001 PRINT "IN SUBTEST1"
1002 SUBEND                      Save this as PART1
1010 SUB STEST2
1011 PRINT "IN SUBTEST2"
1012 SUBEND                      /
```

Follow the directions above and then run PART1A. It should run normally. You can list the program and see that both segments are there. Then type:

55 PRINT "LINE 55" - Line 55 is added to the program You can list and run the program normally.
55 PRINT "IN LINE 55" then LIST. This makes a change to line 55. Lines 1000-1012 are messed up.

Depending on what is in the lines, the computer might crash if you LIST or RUN the program.
CALL LINK("X") then LIST. The line number table is repaired and lines 1000-1012 are correct again.
CALL LOAD(8193,196)::CALL LINK("X") and LIST the program. Only lines 10-60 are listed.
RES and LIST. Lines 10-60 have been resequenced to lines 100-150. Lines 1000-1012 are not there.
CALL LINK("X"). Lines 1000-1012 are back and the program will run properly.

NEW. When you LIST the program you get the message "No Program Present" even though the subprograms are still in low memory.

CALL LINK("X"), then LIST. The segment of the program in low memory can be seen.

You can use "Paste XB" in Classic99 when developing the main program. Just remember to CALL LINK("X") to fix the line number table so the low memory segment can be recognized.

The code that merges the line numbers is not particularly elegant. It compares the last line number of the XB program in high memory with the last line number of the code stored in low memory. If they match then it knows the tables have already been combined so it just refreshes the line number pointers. If they do not match then it combines the two line number tables.

CALL LINK("X")

You may have noticed that CALL LINK("X") is used to access assembly language routines. How can one CALL LINK do so many different things? When CALL LINK is executed, XB first checks that a string has been included for the subroutine name. If it finds a name, in this case "X", it fetches the address of the name lookup routine from >2000 and branches to that address. Normally this is >205A, and the code there would look through a table for the string "X". But you can put any address you want in there and CALL LINK will branch to the code at that address. Because each assembly routine updates >2000 with a new address, several different routines can be executed with the same CALL LINK.