HIGH MEMORY ASSEMBLY CODE

Normally Extended BASIC partitions the 32K memory expansion so that the 8K low memory holds assembly language subroutines, and the 24K of high memory holds the XB program. But is an easy task to trick XB's assembly language loader to load the code into high memory where it can be embedded within an XB program. Once loaded there, the A/L code can be run directly out of high memory.

To understand the pros and cons of this method, let's first describe how the other loaders such as SYSTEX and ALSAVE work. These loaders save assembly language subroutines that have been loaded into low memory. They do this by copying an exact byte by byte "snapshot" into high memory where it is "embedded" within a short XB program, which can then be saved to disk in the standard XB manner. This works because Extended BASIC assumes that anything in high memory must be XB code. When loading the routines, this XB program is loaded to high memory, then the A/L code is almost instantly copied back into the same low memory addresses where it came from. This type of loader is ideally suited for loading XB extensions such as THE MISSING LINK, T40XB, etc. Once the A/L code has been safely moved into low memory, the entire 24K of high memory is available and you can NEW, LOAD, RUN or write XB programs at will without disturbing the contents of low memory.

A slightly different use for this type of loader occurs when an XB program requires a few specific A/L support routines. Normally CALL LOAD is used to load the routines directly from disk into low memory, but this can be very S L O W ! You can embed your A/L subroutines into the ALSAVE loader and then add an XB program to the loader. When the program RUNs, the A/L subroutines get copied into low memory, then the XB program takes over, using the A/L subroutines as needed. This method works perfectly, but I object to it on general principle. The problem is that there are two copies of the A/L subroutines taking up space in memory, only one of which is used. The copy in high memory simply sits there, still embedded in the XB code but doing nothing. Wasted memory is no small matter on a machine with only 32K. But it works, so this complaint is mainly on aesthetic grounds - or is it?

Suppose you want to add some assembly routines to a utility such as THE MISSING LINK, which uses the entire low memory for its routines? Suppose your XB program needs more than 8K of support routines? Suppose you want to run large (up to 24222 bytes long) A/L programs out of XB, just using XB to load them and run them? You cannot do this with the low memory loaders.

To deal with these problems I wrote a high memory loader program called HMLOADER, which runs in the standard XB environment. To use it simply type: RUN "DSKx.HMLOADER"

When HMLOADER runs it prompts you for a list of file names for the A/L subprograms you want to load. The list can have any number of file names. After entering the last file name you want loaded, clear the line with Fctn 3, then press enter.

You are asked, “Continue lookup in low memory table? Y/N”
Press N if all of your assembly subroutines will be in high memory.
Press Y if you will have assembly routines in both high memory and low memory.
Later on, when the XB program has been added to the embedded code you can run it as usual. When the program encounters a CALL LINK, it first searches high memory for the subroutine. If it cannot find a match, it will issue an error message, or continue searching in low memory, depending on what you told it to do.

The subprograms are then loaded twice. The first time simply finds out how big they actually are so the right amount of memory can be reserved. The second time loads them at the top of high-memory. Then a couple of very important things are done. A separate subprogram look-up routine is placed in high-memory; and a single line (line #3) of XB code is generated that will poke the address of this routine into 8192, which is the pointer to the CALL LINK routine.

If your XB program does not use low memory assembly routines such as TML or T40XB then you should add CALL INIT to line #3 before the CALL LOAD to avoid getting a syntax error message. Remember that CALL INIT erases any A/L code in low memory, so you should not do this if additional assembly routines such as The Missing Link will be located in low memory. Then back up your work with SAVE DSKx.HMCODENAME.

If you're using assembly routines in low memory such as THE MISSING LINK you should load them at this time. Then OLD DSKx.HMCODENAME will reload the high memory code.

At this point the assembly language code is embedded and you can add lines of XB code from the keyboard, from disk by using MERGE, or by pasting into Classsic99. The CALL LOAD(8192,N1,N2) does not have to be in line #3, but if you move it, be sure it executes before any CALL LINKs to the high memory code. **Do not type NEW and do not load a program using OLD DSKx.PROGNAME.** Either of these will overwrite the assembly code that HMLOADER just embedded! SAVE DSKx.PROGNAME will save both your XB code along with the embedded A/L routines. After the program is saved you can load the program with OLD DSKx.PROGNAME or RUN "DSKx.PROGNAME" and the embedded A/L files will be loaded along with the XB program.

Resequencing can lead to problems if the XB interpreter finds a byte sequence in the embedded A/L code that it thinks is a line number. To avoid this, the A/L code should be removed before RESequencing. Here's how:

First back up your XB program in merge format: SAVE DSKx.PROGNAME,MERGE
Here you can try resequencing the XB program, but keep in mind that the A/L code may be altered.
If that happens, type NEW to start with a clean slate.
Then MERGE DSKx.PROGNAME which will restore your XB program minus the A/L code.
Then resequence
Now you can save the program again in merge format: SAVE DSKx.PROGNAME,MERGE
Reload the embedded high memory code created by HMLOADER: OLD DSKx.HMCODENAME
And you can then merge in the resequenced program: MERGE DSKx.PROGNAME