COBB

September 1991
Vol. 2 No. 9

INSIDE MICROSOFT BASIC

Tips & techniques for Microsoft Basic

Extending Basic with custom runtimes

Evangelists of next-generation operating systems like OS/2 and Windows praise these systems' ability to place sections of code common to many programs into dynamically linked libraries (DLLs). Isolating functions in stand-alone libraries that programs access at runtime allows several programs to share common sections of code. But you don't need OS/2 or Windows to use this technology. Microsoft implements it in all versions of its DOS Basic compilers in the form of runtimes.

When you compile a Basic program without the /O option, your executable file contains addresses for Basic functions in the runtime rather than the actual code used to perform the function. When you execute the program, Basic loads the default runtime, and your program calls Basic routines in that module. With Basic Version 6.0, Microsoft included the ability to create custom runtimes. You can use this feature to collect objects used by several Basic programs and place them in DOS EXE or OS/2 DLL runtimes. Your Basic programs can then use these routines as if they were part of the Basic language.

This arrangement provides two important advantages:

1. When using runtimes, your executable files consume less disk space because you don't link common routines into every program; instead, you include them in the runtime.

2. Since you define the contents of the runtime, you can include your own routines or third-party utilities and exclude portions of the default Basic runtime your programs don't use. Of course, you must make sure the user doesn't delete the runtime, or none of your programs can access the routines it contained.

In this article, we'll first take a look at the basic process for creating a custom runtime. Then, we'll create a simple basic subprogram and walk through the process of adding it to a runtime library. We'll also look at three special cases: using Quick libraries with custom runtimes, reducing the size of runtimes with stub files, and adding ISAM support to runtimes. Finally, we'll discuss some caveats to keep in mind when using runtime libraries.

The basics

The mechanics of creating a custom runtime are straightforward. First, compile all the routines you intend to add to the runtime. Next, use a text editor to create an export list that contains a list of the modules you want to add to the runtime, a list of the procedures you want to call from the modules, and a list of libraries your procedures reference. Then, start the BUILDRTM utility to actually create the runtime and its support files. Finally, link your program files with the support files BUILDRTM produces to create programs that use the alternate runtime.

Before we look at an example of this process, you should verify that you installed Microsoft Basic so you can generate new runtimes. When installing Version 7.X, the Install program asks whether it should delete the component libraries it uses to generate the base system. In order to build alternate runtimes, you must tell the Install program to retain the component libraries. As a result, your LIB subdirectory will contain the component library files listed in Table A.

Table A: *Alternate runtime component libraries*

B710BJ.LIB	B71mORN.LIB
B710BN.LIB	B71mOEJ.LIB
B710RN.LIB	B71mOEN.LIB
B71S.LIB	B71mLN.LIB
BLIBFP.LIB	B71mCN.LIB
B71mOBJ.LIB	EMm.LIB
B71mORJ.LIB	OS2.LIB (protected mode only)
B71mOBN.LIB	

*m represents R for Real mode or P for Protected Mode

If you don't see these files in your LIB directory, you need to reinstall Microsoft Basic and tell the Install program to retain the component libraries.

Creating a runtime library with one subroutine

After you've configured the system, you're ready to create a simple runtime. In order to demonstrate how to add your

Continued on page 2

IN THIS ISSUE

DATA STORAGE TECHNIQUE

Compressing data with LZW

By Gary Conway and Blake Ragsdell

See Listing 2 on page 14

Computers were made for data compression. Can you imagine compressing data by hand? For example, suppose you want to compress a 100K file with one of the high-performance data-compression techniques available today. Let's assume your chosen compression method requires you to maintain a table with 32,000 entries. To compress the file, you must search the entire table linearly—crude, but workable—for each byte of the input stream. If you allow one second for each search of the table, the 3.2 billion probes would require more than 100 years to compress the file. Hardly livable in the most literal terms. Fortunately, computers can perform these probes in millionths of a second, providing an extremely practical and cost-effective platform for data compression. As we said, computers were made for data compression.

In this article, we're going to examine the Lempel-Ziv-Welch (LZW) compression algorithm. Once we cover a little compression theory, we'll look at a program that implements the LZW algorithm.

The LZW algorithm

The LZW algorithm actually learns a file as it reads it. By *learns*, we mean that the algorithm gets better at compressing a file as it reads more and more of the file. The LZW algorithm essentially reads a character from the input file and looks to see if the character exists in a table of strings. If the table contains the character, the algorithm looks at the next character to see if this string of two characters exists in the table. If it finds the pair of characters, it reads the next character. The algorithm continues reading characters and searching for matches in the table until it doesn't find a match. When this occurs, the algorithm places an entry in the table representing this newly located and now-memorized string.

Output from the LZW algorithm consists of codes. In this way, it's similar to the Huffman coding scheme, in which the shorter bit-length codes generally represent the longer strings of characters. Typically, code lengths begin at nine bits and run to 12 or 13 bits, with the shorter codes being assigned first.

The role of the string table

Without question, the most difficult concept of LZW compression is the string table. To get a grasp of how the algorithm uses the table, envision nailing a two-by-four on the wall horizontally and writing all the characters of the alphabet on the board. Now, imagine taking the sentences in this article and writing them on pieces of string—using a very small pen—one sentence per string. As you complete each string, examine the first character of its string and pin this character to the matching character on the board. You now have a string table—or should we say board?

For example, the sentence *Just earned my PHASE II wings* would hang by the *J* in the word *Just* from our two-by-four board. Next, to each character in the sentence, we assign the somewhat arbitrary codes

```
11111111111111111111111111111
00000000001111111111222222222
01234567890123456789012345678
```

The code is merely the position the character holds in our table. More important than the code itself is the fact that each code represents the entire string from that character all the way to the start of the sentence.

If we encounter the string *Just earned my free airline ticket* in another sentence, we can use a single code—the code representing the space " " right after the word *my*—to represent the first 15 characters in the string. In this case, we could store the new sentence as *airline ticket* using just the codes

```
1xxxxxxxxxxxxx
1yyyyyyyyyyyyy
401234567890123
```

A significant benefit of LZW-coded files is that there's no need to include the string table in the compressed file, as with Huffman coding—thus, the LZW coding scheme increases the amount of compression you get per file. As you'll see later, when you uncompress an LZW-coded file, you recreate the table as you uncompress the file. Additionally, LZW is a comparatively fast compression method because it requires only a single pass through the input file. Other methods, like Huffman coding, require two.

Creating the string table

The LZW algorithm begins by creating an atomic—i.e., Basic—table of characters in memory. Each entry contains two values—a Prefix pointer and a Suffix character. It uses the Prefix and Suffix arrangement to build long strings.

Table A shows how the string table appears after the LZW algorithm initializes it. As you can see, when the compression routine begins, the table contains just 256 entries—one for each character in the extended ASCII set. The LZW algorithm uses the Prefix pointer to point to previous characters in a string. When it initializes the table, there are no strings and no previous characters, so it sets this Prefix pointer to -1 to represent a pointer to nowhere.

On the other hand, the initial values for the Suffix codes range sequentially from 0 to 255. (Table A shows the ASCII

value followed by the character representation.) You must initialize the table with all 256 values before you can start compressing a file with the LZW method, since all files contain some portion of the character set and there's no simple way to predict which characters any given file will contain or exclude.

Table A: *The string table after initialization*

Entry #	Prefix pointer	Suffix character
0	-1	0
1	-1	1 ^A
2	-1	2 ^B
3	-1	3 ^C
:	:	:
255	-1	255
:	:	:
MaxCodes	??	??

Coding the algorithm

Now that you've gotten a feel for the nature of LZW compression and the structure of the string table, let's see how you might code the algorithm. For the moment, let's just use pseudo-code to analyze the implementation of the algorithm. Figure A lists the steps required to compress a file using LZW compression.

Figure A: *The LZW compression algorithm in pseudo-code*

```
  Initialize the string table
  Read char from input file to Prefix
Loop:
  Read char from input file to Suffix
  If end of file Then
    Output the code for the Prefix
    End program
  If Prefix-Suffix pair exists in the table Then
    Prefix = Code for Prefix-Suffix combination
  Else Prefix-Suffix combination not in table
    Output the code for the Prefix
    Insert the Prefix-Suffix combination
      into string table as a new entry
    Prefix = Suffix
  Goto Loop
```

Let's walk through a manual compression session to see how the code in Figure A works with some characters from a sample input stream. For example, suppose the first five characters from a file you want to compress are ABCABA.

After we read the first two characters into Prefix and Suffix variables, we can check to see if the table contains the pair. Obviously, it won't, since they're the first pair of characters. So we'll first write the Prefix character to the output file.

Next, we'll form a new entry in the string table. This entry has as its Prefix pointer a value that *points* to the table entry with A as the Suffix byte—the 65th entry—which also happens to be the ASCII value of A. The Suffix character in this new entry will be the letter B. At this point, the table will contain the values shown in Table B, and the output file will contain a single letter A.

Following the instructions in Figure A, we'll now assign the value of Suffix to Prefix and go to the top of the loop to read another character. When we apply our sample characters to the code, the current Prefix-Suffix pair is equal to BC. After checking the table and not finding the pair there, we output a B to the output file, add the pair of characters to the table, assign a new Prefix value, and go to the top of the loop to do it all over again.

Table B: *The string table after inserting the first entry*

Entry #	Prefix pointer	Suffix character
0	-1	0
1	-1	1 ^A
:		
255	-1	255
256	65 *A*	66 *B*
:		
MaxCodes	??	??

We repeat this process once more for the next Prefix-Suffix pair—CA—and end up with the values shown in Table C.

Table C: *The string table after inserting the first entry*

Entry #	Prefix pointer	Suffix character
0	-1	0
1	-1	1 ^A
:		
255	-1	255
256	65 *A*	66 *B*
257	66 *B*	67 *C*
258	68 *C*	65 *A*
:		
MaxCodes	??	??

Now we're getting to the compression part. The next pair of Prefix-Suffix characters is AB—and we have a matching pair of characters already in the table at entry number 256. Since the pair exists, we store 256 in the Prefix variable and go to the top of the loop to read another Suffix character. When we check to see if the pair (256)-C is in the table, we find it isn't. So, we output the Prefix code to the output file, insert the (256)-C pair of characters into the string table, and so on. Once this is done, the string table contains the values shown in Table D on page 12.

Table D: *The string table after inserting (256)-C*

Entry #	Prefix pointer	Suffix character
0	-1	0
1	-1	1 'A
:		
255	-1	255
256	65 *A*	66 *B*
257	66 *B*	67 *C*
258	68 *C*	65 *A*
259	256 *AB*	67 *C*
:		
MaxCodes	??	??

The output file contains the values shown in Figure B, in which *(256)* represents the entry number for the Prefix pair of AB.

Figure B: *Current output file contents*

```
ABC(256)C
```

The compressor continues learning strings in this way until the input file is exhausted or the table gets filled up with pairs of characters.

Uncompressing a compressed file

Unless our data is worthless, we'll want to restore it to its original uncompressed form at some point. Figure C contains a pseudo-code outline to uncompress our compressed file. Ignoring some of the complexities we can encounter as we uncompress our data, let's see how we'd uncompress the data stream in Figure B using the code in Figure C.

Figure C: *The LZW uncompression algorithm in pseudo-code*

```
  Initialize the string table
  Read char from input file to Prefix
Repeat:
  If Prefix < 256 output the ASCII character
  Read char from input file to Suffix
  If the suffix is a code
     Expand the code
  If Prefix-Suffix pair exists in the table Then
    Prefix = Code for Prefix-Suffix combination
  Else Prefix-Suffix combination not in table
     Insert the Prefix-Suffix combination
       into string table as a new entry
  Prefix = Suffix
  Until the end of the input file
```

The first step is to initialize the string table, just as we did when we were compressing data. Next, we read a Prefix character from the compressed file. Since the ASCII value of A is less than 256, we'll write it to the output file. Next, we'll read a Suffix character—B, in this case—from the input file. Since Suffix isn't a code, we'll check to see if AB appears in the table—which it obviously won't—and insert the entry into the table. The values in the string table correspond to Table B at this point.

Next, we'll set the new Prefix value, write it to the output file, and get a new Suffix. Now the output file contains the characters AB, the value of Prefix is B, and the value of Suffix is C. Since the table doesn't contain a BC character pair, we'll add them, set Prefix equal to C, and output it.

When we read the next character from the compressed file, we find it's one of our compression codes—its value being greater than 255. Let's take stock for just a moment. Currently, the string table contains the values shown in Table E. Prefix is equal to C, Suffix is equal to (256), and the output file contains the characters ABC.

Table E: *Current string table values*

Entry #	Prefix pointer	Suffix character
0	-1	0
1	-1	1 'A
:		
255	-1	255
256	65 *A*	66 *B*
257	66 *B*	67 *C*
:		
MaxCodes	??	??

Since Suffix is a code—in other words not an ASCII character—we must expand it into the characters it represents. Fortunately, this is relatively easy. All we have to do is look up the values for entry number 256, which just happen to be A and B, then write them to the output file. Couldn't be simpler, right? Well, almost. Stay with us for just a moment; this is the hardest part—really. As we said, the current Prefix-Suffix pair is (256)-C. However, the Suffix character in the string table must always be an extended ASCII character, since we can never read a compression code from an uncompressed file. The solution is simply to look up the first character represented by the compression code and use that as the Suffix value when we update the string table. When we take this approach, the Prefix-Suffix pair becomes CA. And since the string table doesn't contain that pair of characters, we can add it to the table.

Aside from the complexity of expanding the compression codes, that's all it takes to uncompress the compressed files. To finish uncompressing the file, we just continue to read characters, writing them to the output file if they're ASCII characters or expanding their values if they're compression codes.

The LZW.BAS program

To demonstrate how LZW compression works, we created the LZW.BAS program, shown in Listing 2 on page 14, which contains routines that implement LZW encoding. Table F lists each routine and its purpose.

To test the program, just load LZW into QBX and run it. Alternatively, you can compile it with BC. LZW expects a filename on the command line, so you'll need to supply one. If you're running the program from QBX, simply choose the Modify Command$ option on the Run menu, enter a name, then select the Start option.

Table F: *LZW.BAS routines*

Name	Purpose
`Init`	Initializes string table and table management variables
`CompressFile`	Manages file compression process
`UnCompressFile`	Uncompresses file
`ManageTbl`	Looks up Prefix-Suffix character pairs and manages insertion process
`InsertEntry`	Adds Prefix-Suffix pair to table
`CompressDone`	Cleans up file compression processes
`ExpandValue`	Expands a compressed value into its original values
`OutputTable`	Outputs the string table for debugging and inspection
`InputCompValue`	Gets a value from a compressed file
`InputUnCompValue`	Gets a value from an uncompressed file
`OutputCompValue`	Writes a value to a compressed file
`OutputUnCompValue`	Writes a value to an uncompressed file

LZW will first call Init to initialize the string table, then the CompressFile subprogram to compress the file. CompressFile compresses into a file named OUTPUT.BAS the file whose name you pass on the command line—thus, LZW.BAS doesn't change your original file. Once LZW finishes compressing your file, it closes the input and output files, then starts uncompressing OUTPUT.BAS into a file named RESTORED.BAS. When LZW finishes the uncompression process, there will be two new files on your disk—a compressed version of your file in a file named OUTPUT.BAS and an uncompressed file named RESTORED.BAS, which was created from the compressed file.

If you run the DOS file COMPARE command on your original file and RESTORED.BAS, they should match perfectly. We tested this program on more than 350 files of various types, including programs and source code, before we were confident enough to publish it. However, because errant compression programs can ruin otherwise good files, you may want to test the program several times yourself to make sure your program matches ours.

Program limitations

The LZW.BAS program has one severe limitation imposed to enhance the readability and simplicity of the program listing. We wrote it with this limitation because we wanted to demonstrate the fundamentals of the compression method without bogging you down with distracting details.

The primary problem is that the codes we send to the output file are all the same length, since we write them to the compressed file using integer type variables. In Basic, integers are 16 bits long, so each code written to the output file has this same length. The optimum method is to recognize that the code values range from 0 to MAXCODES—the length of the array—and output only the exact number of bits needed to define a given code. For example, the first code available for output is 256. Since you only need nine bits to write 256 in binary, you could use only nine bits instead of 16—a savings of more than 40 percent. By varying the bit length of the codes, you can easily realize additional compression over the current version of LZW. To make it easier to solve this problem, we isolated each file input and output operation in its own routine, which you can replace without disturbing the rest of the program.

Program enhancements

The early implementations of LZW stopped learning once they filled the string table. This gave poor compression statistics on some files. A newer feature, called adaptive reset, clears the string table after it fills and begins all over again! As you might imagine, this capability usually helps larger files more than smaller files. In addition, there are numerous variations on the LZW theme, including running the RLE algorithm *before* doing LZW compression and clearing only the least-used codes from the string table rather than the entire table.

We made the program slightly more complex than it needed to be to achieve some performance gains we felt were necessary to present the method. Specifically, we implemented a linked list to search the string table instead of just probing it sequentially. Consequently, we heavily commented the listing to help you when you examine it.

You should notice that the LZW algorithm makes no attempt to optimize the strings it picks for compression—it chooses the first match that comes along when searching the string table. However, this may not always be the best match.

Conclusion

There are many compression algorithms available to address the problems inherent in the LZW method, but as we found out

in July's article, data-compression algorithms are much the same as sorting algorithms—each exhibits best performance when used in the correct situation. Experiment with LZW and see if you can't tweak it to make it run faster or create smaller compressed files—the twin goals of data compression. While our code provides impressive compression of some files, you may notice that others are actually larger than the original file. Remember, we're writing fixed-length integers to the output file. To maximize file compression, you must vary the number of bits per output byte written to the compressed file. Unfortunately, that's a topic for a future article.

Gary Conway is the president of Infinity Design Concepts, Inc., (IDC) and co-developer of the ZIP file format. For information on IDC's products call (502) 636-1234.

Blake Ragsdell is a consulting editor for Inside Microsoft Basic *and editor-in-chief of The Cobb Group's new* Inside Visual Basic.

Source code listings *(You can download listings from CGIS.)*

The following are the complete programs described in the preceding articles. We placed the listings at the end to preserve continuity, and we restricted them to 73 columns to maintain readability in the two-column format. You can download the listings from our online service, CGIS, at any time of day. Use your 300, 1200, or 2400 baud modem with 8 data bits, 1 stop bit, and no parity to call CGIS at (502) 499-2904. Your user ID is your customer number (the one- to seven-digit number on the top line of your mailing label after the *C*).

Listing 1A: *A better way to use BSAVE and BLOAD*

```
' MakeImag.Bas
' Program to demonstrate video-hardware independent
' method of using BSAVE
'
DIM Image%(1 TO 14002)
SCREEN 9
' Draw an image on the screen
WINDOW SCREEN (0, 10)-(199, 119)
CIRCLE (99, 72), 27, 15
PAINT (99, 72), 15
CIRCLE (99, 75), 25, 0
PAINT (99, 75), 0
CIRCLE (99, 79), 15, 15
LINE (81, 68)-(117, 84), 0, BF
CIRCLE (99, 75), 25, 15
CIRCLE (91, 70), 4, 15
CIRCLE (108, 70), 4, 15
CIRCLE (91, 70), 1, 15
PAINT (91, 70), 15
CIRCLE (108, 70), 1, 15
PAINT (108, 70), 15
CIRCLE (100, 77), 3, 15
PAINT (100, 77), 15
CIRCLE (68, 49), 19, 15
PAINT (68, 49), 15
CIRCLE (130, 49), 19, 15
PAINT (130, 49), 15
LINE (20, 14)-(180, 110), , B
' Restore physical device viewport and window
' so GET will work as advertised
VIEW
WINDOW
' Save the four quadrants that form this image
GET (0, 0)-(319, 174), Image%
DEF SEG = VARSEG(Image%(1))
BSAVE "QUAD1.IMG", VARPTR(Image%(1)), 28004
DEF SEG
GET (320, 0)-(639, 174), Image%
DEF SEG = VARSEG(Image%(1))
BSAVE "QUAD2.IMG", VARPTR(Image%(1)), 28004
DEF SEG
GET (0, 175)-(319, 349), Image%
DEF SEG = VARSEG(Image%(1))
BSAVE "QUAD3.IMG", VARPTR(Image%(1)), 28004
DEF SEG
GET (320, 175)-(639, 349), Image%
DEF SEG = VARSEG(Image%(1))
BSAVE "QUAD4.IMG", VARPTR(Image%(1)), 28004
DEF SEG
' All done!
END
```

Listing 1B

```
' ShowImag.Bas
' Program to demonstrate video-hardware independent
' method of using BLOAD
'
DIM Image%(1 TO 14002)
' Set screen mode 9; show page 1 (which is blank)
' while we load the image into page 0.
SCREEN 9, , 0, 1
' Start loading the image
' Load the first quadrant
DEF SEG = VARSEG(Image%(1))
BLOAD "QUAD1.IMG", VARPTR(Image%(1))
DEF SEG
PUT (0, 0), Image%
' Load the second quadrant
DEF SEG = VARSEG(Image%(1))
BLOAD "QUAD2.IMG", VARPTR(Image%(1))
DEF SEG
PUT (320, 0), Image%
' Load the third quadrant
DEF SEG = VARSEG(Image%(1))
BLOAD "QUAD3.IMG", VARPTR(Image%(1))
DEF SEG
PUT (0, 175), Image%
' Load the fourth quadrant
DEF SEG = VARSEG(Image%(1))
BLOAD "QUAD4.IMG", VARPTR(Image%(1))
DEF SEG
PUT (320, 175), Image%
' Now, let's make both active and visible pages
' be page 0, to display the complete image.
SCREEN 9, , 0, 0
' Allow the user to view the image
SLEEP
END
```

Listing 2: *Compressing data with LZW*

```
DECLARE SUB OutputUnCompValue (I%)
DECLARE SUB OutputCompValue (I%)
DECLARE SUB CompressFile ()
DECLARE SUB Done ()
DECLARE SUB Init (Ind AS STRING, Outf AS STRING)
DECLARE SUB InsertEntry ()
DECLARE SUB StoreTable (FileTbl$)
DECLARE SUB UnCompressFile ()
DECLARE FUNCTION InputUnCompValue% ()
DECLARE FUNCTION InputCompValue% ()
DECLARE FUNCTION ManageTbl% ()
```