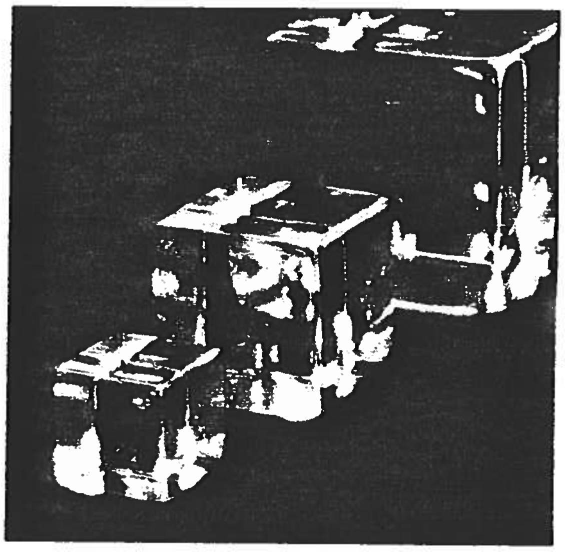
LZW Data Compression

Here's an all-purpose data compression technique that belongs in your programming toolbox

Mark R. Nelson

Every programmer should have at least some exposure to the concept of data compression. Programs such as ARC by System Enhancement Associates (Wayne, N.J.) and PKZIP by PKWARE (Glendale, Wisc.) are ubiquitous in the MS-DOS world. ARC has also been ported to quite a few other operating systems, for example, Unix, CP/M, and so on. CP/M users have long had SQ and USQ to squeeze and expand programs. Unix users have the *COMPRESS* and *COMPACT* utilities. Yet the data compression techniques used in these programs typically show up in only two places: File transfers over phone lines and archival storage.



Data compression has an undeserved reputation for being difficult to master, hard to implement, and tough to maintain. In fact, the techniques used in the previously mentioned programs are relatively simple, and can be implemented with standard utilities taking only a few lines of code. In this article, I'll discuss Lempel-Ziv-Welch (LZW) compression, a good, all-purpose data compression technique that belongs in every programmer's toolbox.

LZW, for example, by compressing the screens, can easily chop 50K bytes off a program that has several dozen help screens. With LZW compression, 500K bytes of software could be distributed to end users on a single 360K byte floppy disk. Highly redundant database files can be compressed to ten percent of their original size.

Mark is a programmer for Greenleaf Software, Inc., Dallas, Texas. Mark can be reached through the DDJ *office.*

LZW Fundamentals

The original Lempel/Ziv approach to data compression was first published in 1977, and Terry Welch's refinements to the algorithm were published in 1984. The algorithm is surprisingly simple. In a nutshell, LZW compression replaces strings of characters with single codes. It does not do any analysis of the incoming text. Instead, it just adds every new string of characters it sees to a table of strings. Compression occurs when a single code is output replacing the string of characters.

The code generated by the LZW algorithm can be of any length, but it must have more bits in it than a single character. The first 256 codes (when using 8-bit characters) are by default assigned to the standard character set. The remaining codes are assigned to strings as the algorithm proceeds. The sample program runs, as shown in Listing One, page 86, with 12-bit codes. This means codes 0 – 255 refer to individual bytes, and codes 256 – 4095 refer to substrings.

Compression

The LZW compression algorithm in its simplest form is shown in Figure 1. Each time a new code is generated, it means a new string has been added to the string table. Examination of the algorithm shows that LZW always checks whether the strings are already known and, if so, outputs existing codes rather than generating new codes.

A sample string used to demonstrate the algorithm is shown in Figure 2. The input string is a short list of English words separated by the / character. As you step through the start of the algorithm for this string, you can see that in the first pass through the loop the system performs a check to see if the string /W is in the table. When it doesn't find the string in the table, it generates the code for /, and the string /W is added to the table. Because 256 characters have already been defined for codes 0 – 255, the first string definition can be assigned to code 256. After the system reads in the third letter, E, the second string code, WE, is added to the table, and the code for letter W is output. This process continues until, in the second word, the characters / and W are read in, matching string number 256. In this case, the system outputs code 256, and adds a three-character string to the string table. The process again continues until the string is exhausted and all of the codes have been output.

The sample output for the string is also shown in Figure 2, along with the

(continued on page 32)

(continued from page 28)

resulting string table. As you can see, the string table fills up rapidly, because a new string is added to the table each time a code is generated. In this highly redundant example input, five code substitutions were output, along with seven characters. If we were using 9-bit codes for output, the 19-character input string would be reduced to a 13.5-byte output string. Of course, this example was carefully chosen to demonstrate code substitution. In the real world, compression usually doesn't begin until a sizable table has been built, usually after at least 100 or so bytes have been read in.

```
ROUTINE LZW_COMPRESS
STRING = get input character
WHILE there are still input characters DO
    CHARACTER = get input character
    IF STRING+CHARACTER is in the string table THEN
        STRING = STRING+character
    ELSE
        output the code for STRING
        add STRING+CHARACTER to the string table
        STRING = CHARACTER
    END of IF
END of WHILE
output the code for STRING
```

***Figure 1:** The compression algorithm*

Input string: /WED/WE/WEE/WEB/WET

Character input	Code output	New code value and associated string
/W	/	256 = /W
E	W	257 = WE
D	E	258 = ED
/	D	259 = D/
WE	256	260 = /WE
/	E	261 = E/
WEE	260	262 = /WEE
/W	261	263 = E/W
EB	257	264 = WEB
/	B	265 = B/
WET	260	266 = /WET
<EOF>	T	

***Figure 2:** The compression process*

```
ROUTINE LZW_DECOMPRESS
Read OLD_CODE
output OLD_CODE
WHILE there are still input characters DO
    Read NEW_CODE
    STRING = get translation of NEW_CODE
    output STRING
    CHARACTER = first character in STRING
    add OLD_CODE + CHARACTER to the translation table
    OLD_CODE = NEW_CODE
END of WHILE
```

***Figure 3:** The decompression algorithm*

Input codes: / W E D 256 E 260 261 257 B 260 T

Input NEW_CODE	OLD_CODE	STRING Output	CHARACTER	New table entry
/	/	/		
W	/	W	W	256 = /W
E	W	E	E	257 = WE
D	E	D	D	258 = ED
256	D	/W	/	259 = D/
E	256	E	E	260 = /WE
260	E	/WE	/	261 = E/
261	260	E/	E	262 = /WEE
257	261	WE	W	263 = E/W
B	257	B	B	264 = WEB
260	B	/WE	/	265 = B/
T	260	T	T	266 = /WET

***Figure 4:** The decompression process*

Decompression

The companion algorithm for compression is the decompression algorithm. It takes the stream of codes output from the compression algorithm and

LZW compression excels when confronted with data streams that have any type of repeated strings

uses it to exactly recreate the input stream. One reason for the efficiency of the LZW algorithm is that it does not need to pass the string table to the decompression code. The table can be built exactly as it occurred during compression, using the input stream as data. This is possible because the compression algorithm always outputs the STRING and CHARACTER components of a code before it uses the code in the output stream. This means that the compressed data is not burdened with carrying a large string translation table.

The decompression algorithm is shown in Figure 3. Just like the compression algorithm, it adds a new string to the string table each time it reads in a new code. All it needs to do in addition is to translate each incoming code into a string and send it to the output.

Figure 4 shows the output of the algorithm given the input created by the compression discussed earlier in the article. The important thing to note is that the decompression string table ends up looking exactly like the table built up during compression. The output string is identical to the input string from the compression algorithm. Note that the first 256 codes are already defined to translate to single character strings, just like in the compression code.

The Catch

Unfortunately, the nice, simple, decompression algorithm shown in Figure 4 is just a little too simple. There is a

(continued from page 32)
single exception case in the LZW compression algorithm that causes some trouble on the decompression side. If there is a string consisting of a (STRING, CHARACTER) pair already defined in the table, and the input stream sees a sequence of STRING, CHARACTER, STRING, CHARACTER, STRING, the compression algorithm outputs a code before the decompressor gets a chance to define it.

A simple example illustrates the point. Imagine the string JOEYN is defined in the table as code 300. When the sequence JOEYNJOEYNJOEY appears in the table, the compression output looks like that shown in Figure 5.

When the decompression algorithm sees this input stream, it first decodes the code 300, then outputs the JOEYN string and adds the definition for, lets say, code 399 to the table, whatever that may be. It then reads the next input code, 400, and finds that it is not in the table. This is a problem.

Fortunately, this is the only case where the decompression algorithm will encounter an undefined code. Because it is, in fact, the only case, you can add an exception handler to the algorithm. The modified algorithm just looks for the special case of an undefined code and handles it. In the example in Figure 6, the decompression routine sees code 400, which is undefined. Because it is undefined, it translates the value of *OLD_CODE*, which is code 300. It then adds the CHARACTER value, *J*, to the string. This results in the correct translation of code 400 to string JOEYNJ.

The Implementation Blues

The concepts used in the compression algorithm are so simple that the whole algorithm can be expressed in only a dozen lines. But because of the management required for the string table, implementation of this algorithm is somewhat more complicated.

Highly redundant database files can be compressed to ten percent of their original size

In the code accompanying this article (see Listing One), I have used code sizes of 12-, 13-, and 14-bits. In a 12-bit code program, there are potentially 4096 strings in the string table. Each and every time a new character is read in, the string table has to be searched for a match. If a match is not found, a new string has to be added to the table. This causes two problems. First, the string table can get very large very fast. Even if string lengths average as low as 3 or 4 characters each, the overhead of storing a variable length string and its code can easily reach 7 or 8 bytes per code. In addition, the amount of storage needed is indeterminate, as it depends on the total length of all the strings.

The second problem involves searching for strings. Each time a new character is read in, the algorithm has to search for the new string formed by STRING+CHARACTER. This means keeping a sorted list of strings. Searching for each string takes on the order of *log2* string comparisons. Using 12-bit words potentially means doing 12-string comparisons for each code. The computational overhead can be prohibitive.

The first problem can be solved by storing the strings as code/character combinations. Because every string is actually a combination of an existing code and an appended character, you can store each string as a single code plus a character. For example, in the compression example shown, the string /WEE is actually stored as code 260 with appended character E. This takes only 3 bytes of storage instead of 5 (counting the string terminator). By backtracking, you find that code 260 is stored as code 256 plus an appended character E. Finally, code 256 is stored as a /character plus a W.

Doing the string comparisons is a little more difficult. The new method of storage reduces the amount of time needed for a string comparison, but it doesn't cut into the number of comparisons needed to find a match. This problem is solved by using a hashing algorithm to store strings. What this means is that you don't store code 256 in location 256 of an array, you store it in a location in the array based on an address formed by the string itself. When you are trying to locate a given string, you can use the test string to generate a hashed address and, with luck, can find the target string in one search.

Because the code for a given string is no longer known merely by its position in the array, you need to store the code for a given string along with the string data. In Listing One, there are three array elements for each string. They are: *code_value[i]*, *prefix_code[i]*, and *append_character[i]*.

When you want to add a new code to the table, use the hashing function in routine *find_match* to generate the correct *i*. *find_match* generates an address, then checks to see if the location is already in use by a different string. If it is, *find_match* performs a secondary probe until an open location is found.

The hashing function in use in this

Input string: . . . JOEYNJOEYNJOEY . . .

Character input	New code value and associated string	Code output
JOEYN	300 = JOEYN	288 (JOEY)
A	301 = NA	N
.	.	.
.	.	.
.	.	.
JOEYNJ	400 = JOEYNJ	300 (JOEYN)
JOEYNJO	401 = JOEYNJO	400

Figure 5: *Sample problem*

```
ROUTINE LZW_DECOMPRESS
Read OLD_CODE
output OLD_CODE
WHILE there are still input characters DO
    Read NEW_CODE
    IF NEW_CODE is not in the translation table THEN
        STRING = get translation of OLD_CODE
        STRING = STRING+CHARACTER
    ELSE
            STRING = get translation of NEW_CODE
    END of IF
    output STRING
    CHARACTER = first character in STRING
    add OLD_CODE + CHARACTER to the translation table
    OLD_CODE = NEW_CODE
END of WHILE
```

Figure 6: *The modified decompression algorithm*

(continued from page 34)
program is a straightforward *xor*-type hash function. The prefix code and appended character are combined to form an array address. If the contents of the prefix code and character in the array are a match, the correct address is returned. If that element in the array is in use, a fixed offset probe is used to search new locations. This continues until either an empty slot is found, or a match is found. The average number of searches in the table usually stays below 3 if you use a table about 25 percent larger than needed. Performance can be improved by increasing the size of the table. Note that in order for the secondary probe to work, the size of the table needs to be a prime number. This is because the probe can be any integer between 1 and the table size. If the probe and the table size are not mutually prime, a search for an open slot can fail even if there are still open slots available.

Implementing the decompression algorithm has its own set of problems. One of the problems from the compression code goes away. When you are compressing, you need to search the table for a given string. During decompression, you are looking for a particular code. This means that you can store the prefix codes and appended characters in the table indexed by their string code. This eliminates the need for a hashing function and frees up the array used to store code values.

Unfortunately, the method used to store string values causes the strings to be decoded in reverse order. This means that all the characters for a given string have to be decoded into a stack buffer, then output in reverse order. In the program given here, this is done in the *decode_string* function. Once this code is written, the rest of the algorithm turns into code easily.

A problem encountered when reading in data streams is determining when you have reached the end of the input data stream. In this particular implementation, I have reserved the last defined code, *MAX_VALUE*, as a special end of data indicator. Though this may not be necessary when reading in data files, it is helpful when reading compressed buffers out of memory. The expense of losing one defined code is minimal in comparison to the convenience gained.

Results

It is somewhat difficult to characterize the results of any data compression technique. The level of compression achieved varies quite a bit, depending on several factors. LZW compression excels when confronted with data streams that have any type of repeated strings. Because of this, it does extremely well when compressing English text. Compression levels of 50 percent or better can be expected. Likewise, compressing saved screens and displays generally shows great results. Trying to compress data files is a little more risky. Depending on the data, compression may or may not yield good results. In some cases, data files compress even more than text. A little bit of experimentation usually gives you a feel for whether your data will compress well or not.

Your Implementation

The code accompanying this article works. It was written, however, with the goal of being illuminating, not efficient, with some parts of the code being relatively inefficient. The variable length input and output routines, for example, are short and easy to understand, but require a lot of overhead. You could experience real improvements in speed in an LZW program using fixed-length 12-bit codes, just by recoding these two routines.

One problem with the code listed here is that it does not adapt well to compressing files of differing sizes. Using 14- or 15-bit codes gives better compression ratios on large files (because they have a larger string table to work with), but poorer performance on small files. Programs such as ARC get around this problem by using variable length codes. For example, when the value of *next_code* is between 256 and 511, ARC inputs and outputs 9-bit codes. When the value of *next_code* increases to the point where 10-bit codes are needed, both the compression and decompression routines adjust the code size. This means that the 12-bit and 15-bit versions of the program do equally well on small files.

Another problem on long files is that frequently the compression ratio begins to degrade as more of the file is read in. The reason for this is simple: Because the string table is of finite size, after a certain number of strings have been defined, no more can be added. But the string table is good only for the portion of the file that was read in while it was built. Later sections of the file may have different characteristics and really need a different string table.

The conventional way to solve this problem is to monitor the compression ratio. After the string table is full, the compressor watches to see if the compression ratio degrades. After a certain amount of degradation, the entire table is flushed and gets rebuilt from scratch. The expansion code is flagged when this happens because the compression routine sends out a special code. An alternative method is to keep track of how frequently strings are used, and to periodically flush values that are rarely used. An adaptive technique like this may be too difficult to implement in a reasonable-sized program.

One final technique for compressing the data is to take the LZW codes and run them through an adaptive Huffman coding filter. This generally exploits a few more percentage points of compression, but at the cost of considerably more complexity in the code as well as quite a bit more run time.

Portability

The code in Listing One was written and tested on MS-DOS machines and has successfully compiled and executed with several C compilers. It should be portable to any machine that supports 16-bit integers and 32-bit longs in C. MS-DOS C compilers typically have trouble dealing with arrays larger than 64K bytes, preventing an easy implementation of 15- or 16-bit codes in this program. On machines using different processors, such as the VAX, these restrictions are lifted, and using larger code sizes becomes much easier.

In addition, porting this code to assembly language should be fairly easy on any machine that supports 16- and 32-bit math, and offers significant performance improvements. Implementations in other high-level languages should be straightforward.

Bibliography

Terry Welch, "A Technique for High-Performance Data Compression," *Computer*, June 1984.

J. Ziv and A. Lempel, "A Universal Algorithm for Sequential Data Compression," *IEEE Transactions on Information Theory*, May 1977.

Rudy Rucker, *Mind Tools*, Houghton Mifflin Company, Boston, Mass.: 1987.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listing begins on page 86.)

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

LZW Revisited

Speeding up an old data compression favorite

Shawn M. Regan

When the October 1989 issue of *Dr. Dobb's* came out, I was delighted to see Mark Nelson's article on LZW data compression ("LZW Data Compression"). Mark presented a clear description with code of a basic LZW compression program. As I used the program, however, I discovered that for some larger files, even when using 14-bit codes, the compressed file would actually be larger than the original. Because Mark's intention was to enlighten and not to complicate the subject, his code omitted some optimizing additions to LZW compression. Although, he did describe some optimization techniques — including the use of variable code size — as well as clearing the string table after the compression ratio degrades.

The original program's lack of performance on larger files can be traced to the fixed-length string table. When the table is full, new codes cannot be added and must be sent out in character form with no compression done. When this happens, you could actually be sending out an 8-bit character using a 14-bit code; this explains how the file can get larger. If your incoming data changes, even moderately, with the string table full, then the compression ratio begins to degrade rapidly.

With this in mind, imagine compressing a small file with your code size set at 14 bits. If your string table needs less than 511 entries, you could have used 9-bit codes saving 5 bits per output code. Of course you wouldn't want to fix your code size at 9 bits because the compression on larger files would suffer. What you would like is for the code size to start at 9 bits and if that table filled, the code size could increment to 10, thus providing optimal performance on any size file.

My Implementation

One of the more elegant features about LZW compression are that the compression and expansion programs build the exact same string table for a particular file. This means at the same point the compression program's string table is full so is the expansion program's. At this point you should try to adjust your code size. As you can see in the compression section of Listing One (page 127), I wait until after I have sent out the current code before the code size is incremented because the current code belongs to the previous code size. Notice in the compression section I increment when the code size is greater than *max_code*, while in the expansion program I increment when the code size equals *max_code*. This is because the expansion section is working a code behind using *old_code* instead of *new_code*. It is also because of this that I must handle a special case when incrementing the code size on an end-of-file condition.

Finally, you should also set some arbitrary limit on your code size. If you use 14 bits, your codes stay well under the positive integer maximum of 32767, which suits the program without any modification. Don't forget your table size needs to be a prime number somewhat larger than 2^MAX_CODE_SIZE. To implement the table clearing, start by monitoring the number of bytes (not codes) read in and then sent out. After a predetermined interval, compute the new compression ratio and check it against the previous one. If the ratio has increased, you need to clear out the string table and start over. You then need a device to send a signal from the compression program to the expansion program to clear the string table. The easiest way to accomplish this is by reserving the first of the 9-bit codes. In my example, I used 256 as the *CLEAR_TABLE* code. I also used 257 as the *TERMINATOR* to signal the end-of-file condition. This means the first available code for compression is now 258, which I've defined as *FIRST_CODE*.

When combining both methods, you should not experience any degradation in compression until the table is full. When the table is full, you will first check to see if you can increase the code size. If you can't, then (and only then) will you start to monitor your compression ratio at your predefined interval and ultimately clear the string table. When this happens, you can reset your code size back to 9-bits because basically you're starting from scratch. Although you still won't get performance as good as PKZIP (from

Before	After	% of Original	File type
115,094	40,636	35 %	.C - C program
11,054	4,811	43 %	.C - C program
230,582	141,659	61 %	.EXE - Executable
16,944	12,905	76 %	.EXE - Executable
90,610	20,806	22 %	.TXT - Redundant text file
110,592	64,804	58 %	.LIB - C object library

Table 1: *Typical compression levels using revised LZW*

hawn is a programmer/analyst for Mi-roBilt Inc. of Atlanta, Georgia. You an reach him through Interlink; or rite to him at 2127B Powers Ferry d., Marietta, GA 30067.

PKWARE, Glendale, Wisc.), you now have the source for a much improved version of this basic LZW compression program. Table 1 lists some typical compression levels I've achieved with this program.

Your Implementation

Even though the program works well as is, there are still some improvements that can be made. As Mark suggested, the input and output routines can be modified for more speed. Also, a more sophisticated hashing routine might speed it up. For better compression, you might experiment with table clearing. I found on .EXE files the compression ratio drops steadily after a code size increase, then bottoms out and then starts rising again. If you suspend clearing until you are back to just below the starting ratio you can get a somewhat better compression. I also noticed that in smaller text files, I can at times get better compression by clearing the table instead of increasing the code size. Be careful, however, about basing any optimization methods on any preanalysis of the data. If, for example, you wish to use it with stream I/O you will be working with buffers and not files where any preanalysis might be difficult or useless.

For better compression, you might experiment with table clearing

Availability

All source code is available on a single disk and online. To order the disk, send $14.95 (Calif. residents add sales tax) to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 800-356-2002 (from inside Calif.) or 800-533-4372 (from outside Calif.). Please specify the issue number and format (MS-DOS, Macintosh, Kaypro). Source code is also available online through the *DDJ* Forum on CompuServe (type GO DDJ). The *DDJ* Listing Service (603-882-1599) supports 300/1200/2400 baud, 8-data bits, no parity, 1-stop bit. Press SPACEBAR when the system answers, type: listings (lowercase) at the log-in prompt.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 12.**

Listing One

```
/* Basic LZW Data Compression program published in DDJ October 1989 issue.
 * Original Author: Mark R. Nelson
 * Updated by: Shawn M. Regan, January 1990
 * Added: - Method to clear table when compression ratio degrades
 *        - Self adjusting code size capability (up to 14 bits)
 * Updated functions are marked with "MODIFIED". main() has been updated also
 * Compile with -ml (large model) for MAX_BITS == 14 only
 */

#include <stdio.h>

#define INIT_BITS 9
#define MAX_BITS 14                /* Do not exceed 14 with this program */
#define HASHING_SHIFT MAX_BITS - 8

#if MAX_BITS == 14                 /* Set the table size. Must be a prime    */
#define TABLE_SIZE 18041           /* number somewhat larger than 2^MAX_BITS.*/
#elif MAX_BITS == 13
#define TABLE_SIZE 9029
#else
#define TABLE_SIZE 5021
#endif

#define CLEAR_TABLE 256    /* Code to flush the string table */
#define TERMINATOR  257    /* To mark EOF Condition, instead of MAX_VALUE */
#define FIRST_CODE  258    /* First available code for code_value table */
#define CHECK_TIME  100    /* Check comp ratio every CHECK_TIME chars input */

#define MAXVAL(n) (( 1 <<( n )) -1)   /* max_value formula macro */

unsigned input_code();
void *malloc();

int *code_value;                         /* This is the code value array */
unsigned int *prefix_code;               /* This array holds the prefix codes */
unsigned char *append_character;         /* This array holds the appended chars */
unsigned char decode_stack[4000];        /* This array holds the decoded string */

int num_bits=INIT_BITS;                  /* Starting with 9 bit codes */
unsigned long bytes_in=0,bytes_out=0;    /* Used to monitor compression ratio */
int max_code;                            /* old MAX_CODE */
unsigned long checkpoint=CHECK_TIME;     /* For compression ratio monitoring */

main(int argc, char *argv[])
{
   FILE *input_file, *output_file, *lzw_file;
   char input_file_name[81];
/* The three buffers for the compression phase.  */
   code_value=malloc(TALLE_SIZE*sizeof(unsigned int));
   prefix_code=malloc(TABLE_SIZE*sizeof(unsigned int));
   append_character=malloc(TABLE_SIZE*sizeof(unsigned char));

   if (code_value==NULL :: prefix_code==NULL :: append_character==NULL) {
      printf("Error allocating table space!\n");
      exit(1);
   }
/* Get the file name, open it, and open the LZW output file. */
   if (argc>1)
      strcpy(input_file_name,argv[1]);
   else {
      printf("Input file name: ");
      scanf("%s",input_file_name);
   }
   input_file=fopen(input_file_name,"rb");
   lzw_file=fopen("test.lzw","wb");
   if (input_file == NULL :: lzw_file == NULL) {
      printf("Error opening files\n");
      exit(1);
   }
   max_code = MAXVAL(num_bits);      /* Initialize max_value & max_code */
   compress(input_file,lzw_file);       /* Call compression routine */

   fclose(input_file);
   fclose(lzw_file);
   free(code_value);
                                        /* Needed only for compression */
   lzw_file=fopen("test.lzw","rb");
   output_file=fopen("test.out","wb");
   if (lzw_file == NULL :: output_file == NULL) {
      printf("Error opening files\n");
      exit(1);
   }
   num_bits=INIT_BITS;
   max_code = MAXVAL(num_bits);          /* Re-initialize for expansion */
   expand(lzw_file,output_file);
                                         /* Call expansion routine */
   fclose(lzw_file);
   fclose(output_file);                  /* Clean it all up */
   free(prefix_code);
   free(append_character);
}
/* MODIFIED This is the new compression routine. The first two 9-bit codes
 * have been reserved for communication between the compressor and expander.
 */
compress(FILE *input, FILE *output)
{
   unsigned int next_code=FIRST_CODE;
   unsigned int character;
   unsigned int string_code;
   unsigned int index;
   int i,               /* All purpose integer */
   ratio_new,           /* New compression ratio as a percentage */
   ratio_old=100;       /* Original ratio at 100% */

   for (i=0;i<TABLE_SIZE;i++)   /* Initialize the string table first */
      code_value[i]=-1;
   printf("Compressing\n");
   string_code=getc(input);     /* Get the first code */
```

(continued on page 167)

Run Length Encoding

RLE is an efficient—yet simple—way of reducing storage requirements

Robert Zigon

Run Length Encoding (RLE) is one of several techniques that can be used to reduce the storage requirements of text files, databases, and digital images. The algorithm is very simple to implement, it produces output files that, on the average, require 80 percent of the input file space, and executes quickly. Because the compression factor is of prime interest, it should be pointed out that the worst case performance of the algorithm causes the output file to double its size. My use of the algorithm, however, for compressing and saving the digitized output of a frame-grabber board has never resulted in this degeneracy.

The algorithm works as follows: An input buffer is scanned for sequences of identical characters. When a character transition occurs the repetition count of the previous character (along with the character itself) is sent to an output buffer.

This continues until the end of the input buffer is reached. My implementation currently uses 1 byte for the repetition count. This allows for sequences of identical characters to be reduced to exactly 2 bytes. The output for the input buffer looks like this:

Input Buffer : AAAA BBBBBB CC D EEEEE
Output Buffer : 4A 6B 2C 1D 5E

In this example, the 18 characters in the input buffer were reduced to 10 characters in the output buffer (the spaces in the input and output buffers are there to make it easier to read). Notice that no space savings was gained by compressing the CC and that the space requirements of the D doubled.

Listing One contains a routine called PACK that can be used to compress an input buffer. The code is written in the assembly language of the Intel 80X86 family. The routine is designed so that multiple calls will result in having the output concatenated to the contents of the previous call.

After the information is packed and saved to disk, the day will come when you will need to unpack it. Unpacking is significantly easier. A repetition count is read from the packed buffer, and the corresponding character is sent to the output buffer that many times. The string load (LODSB) mnemonic permits the efficient reading of the pairs, while the string store (STOSB) and REP prefix perform the duplication and restoration of the input file. Listing One also contains the necessary UNPACK routine to accomplish this.

Run Length Encoding is a fast and efficient technique for reducing the space requirements of an input file. Though more complex algorithms exist with better compression factors (and worse compression times), its elegance lies in its simplicity.

Availability

All source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063, or call 415-366-3600, ext. 221. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 16.**

Robert Zigon is a senior software engineer and can be reached at 4505 Candletree Circle, Apt. 7, Indianapolis, IN 46254.

Listing One

```
;-----------------------------------------------------------------------
; RLE.asm   Run Length Encoding Routines
; Author : Bob Zigon
;-----------------------------------------------------------------------
; ----------------------------------------------------------------------
; PACK
;    This routine will pack a source buffer into a destination buffer
;    by reducing sequences of identical characters down to a 1 byte
;    repetition count and a 1 byte character
;    INPUT : DS:SI -- Points to source buffer to pack
;            ES:DI -- Points to destination
;            DX    -- Number of characters in source buffer to pack
;   OUTPUT: DI    -- Is updated to allow for multiple calls.
; ----------------------------------------------------------------------
pack            proc   near
                push   ax
                push   bx
                push   cx
                lodsb
                mov    bl,al
                xor    cx,cx                 ; Counts number of characters packed
                cld                          ; All moves are forward

p10:            lodsb                        ; Get chara into AL
                inc    cx                    ; Inc chara count
                sub    dx,1                  ;
                je     p30                   ; Exit when DX = 0
                cmp    cx,255                ; 255 characters in this block yet?
                jne    p20                   ; If not then jump

                mov    [di],cl               ; output the length
                inc    di
                xor    cx,cx
                mov    [di],bl               ; output the character
                inc    di

p20:            cmp    al,bl                 ; Has there been a character transition?
                je     p10                   ; NOPE ... so go back for more

                mov    [di],cl               ; output the length
                inc    di
                xor    cx,cx
                mov    [di],bl               ; output the character
                inc    di
                mov    bl,al                 ; Move Latest chara into BL
                jmp    p10                   ; Loop back for more
;
;  We will get here when DX finally goes to zero.
;
p30:            mov    al,cl                 ; Output the length
                stosb
                mov    al,bl                 ; Output the character
                stosb

                pop    cx
                pop    bx
                pop    ax
                ret
pack            endp

; ----------------------------------------------------------------------
; UNPACK
;    This routine will unpack a sequence of characters that were
;    previously PACKed.
;    NOTE  : Source Region must be terminated with a NULL.
;    INPUT : DS:SI -- Source buffer to unpack
;            ES:DI -- Destination buffer to unpack into
;    OUTPUT:
; ----------------------------------------------------------------------
unpack          proc   near
                push   ax
                push   cx
                xor    cx,cx     ; Make sure CH is zero
                cld              ; All moves are forward

unp10:          lodsb            ; Load into AL a character from DS:[SI]
                cmp  al,0        ; Length of zero is end of region
                je   unp40       ;
                mov  cl,al       ; Length goes into CL
                lodsb            ; AL has chara to repeat

                rep stosb        ; Store AL to [DI] and repeat while CX <> 0
                jmp unp10        ; Loop back forever

unp40:          pop  cx
                pop  ax
                ret
unpack          endp
```

End Listing