Many of today’s embedded systems are providing more sophisticated solutions to a wide variety of applications and industries. With this increase in sophistication, there is a corresponding increase in the amount of data being collected and stored. This rise in data volume is placing even more emphasis on design solutions that need to meet stringent performance requirements while staying within cost targets.

Collecting and storing large amounts of data can create a number of technical challenges. In addition, the embedded system itself can become overtaxed by the increased burden imposed by processing large amounts of data. Faced with the issues of collecting and storing data, embedded system designers are finding data compression as a viable means of dealing with large amounts of data.

Data compression
Data compression is all around us. We see it in a variety of products, such as, high definition televisions, DVD and MP3 players, cell phones, digital cameras, fax machines, automobiles, etc. When you look at all the embedded products out there you will quickly see that data compression is an integral part of their operation.

There are a significant number of data compression techniques available to meet the specific needs of most applications. But whatever the solution may be, all compression algorithms fall into two main categories, lossy and lossless data compression.

Lossy data compression is allowed in the compression algorithm. We see lossy data compression predominantly in the areas of imagery, video, and audio applications. When it comes to lossless data compression, there are two options you can choose, Statistical or Dictionary-Based compression methods. The method chosen is based primarily upon the type of data being compressed.

Statistical-based compression programs exploit the statistical redundancy of individual characters within the data. The program counts each individual character and determines its frequency of occurrence. Those characters with the highest frequency of occurrence are assigned the shortest codes, thereby taking up less memory.

Dictionary-based compression programs do not use statistics or variable-length codes. Instead they select strings of characters and encode each string as a fixed-length token using a dictionary. For each string of characters that occur in the data, a token is used to represent it. Dictionary-based compression is the most popular method used for lossless data compression.

Lossless compression
Lossless data compression, as its name implies, is the process of compressing data without altering or destroying its original content. This form of compression is specifically for applications where a loss in data cannot be tolerated.

There are a number of lossless data compression algorithms to choose from and most vary in methodology, code size and complexity. The one you choose will depend primarily on the specific structure of the data, as well as the objectives of your particular application.

A number of websites offer embedded source code that is compatible with most of the popular compression standards, such as PKZIP, GZIP, 7ZIP, or UNIX’s Compress program. Whichever compression standard is chosen, chances are it will require a large amount of RAM memory. Typically, the more RAM you dedicate to the compression program the higher the compression ratio.

However, not all embedded systems support large blocks of RAM memory that can be allo-
cated to a hungry compression algorithm. For embedded designs with limited RAM memory the challenge is to find software that can achieve acceptable efficiencies within a small memory footprint.

This article describes a lossless compression algorithm based on the popular Lempel-Ziv-Welch (LZW) compression standard. This compression technique is capable of achieving respectable compression ratios, typically on the order of 50-60 per cent, while consuming about 2K of RAM. In larger RAM memory sizes, 8K or 16K, it is possible to achieve 80 per cent efficiency or more.

**LZW algorithm**

LZW is a lossless data compression technique that was created back in 1984 by Terry Welsh as an improvement to the popular LZ77 compression algorithm.

LZW is one of the most popular compression programs available and is the easiest dictionary-based algorithm to implement into an embedded design. It is the compression technique used to generate TIFF and GIF files.

It is also the compression technique used for UNIX’s Compress program. LZW was also the compression technique used by computer programs that claimed to “double your hard drive’s capacity!”

At the heart of the LZW compression program is the dictionary (Figure 1), which translates string patterns from the original data into a single token. The goal of the dictionary is to replace redundant data strings with single value tokens.

Unlike other dictionary-based programs, the real elegance of the LZW compression algorithm lies in the fact that the dictionary does not have to be attached and transmitted with the compressed data itself. The dictionary, which is dynamically constructed in RAM memory, is simply discarded after the compression process has completed.

The same elegance can be found in the LZW decompression algorithm, which dynamically re-constructs the dictionary as it converts the data back to its original form.

The LZW compression algorithm is designed to input data, accumulate it into a string, generate a dictionary that dynamically assigns tokens, and to output those tokens into a compressed format.

As each data value is input and concatenated to the string, the dictionary is searched. As long as the string is found in the dictionary, the process of concatenating another data value into the string will continue. Eventually, the next data value added to the string causes the search in the dictionary to fail.

At this point, the compress program outputs the dictionary token from the previous string to the compressed data output and generates a new token value for the newly created string. Figure 2 illustrates this process in detail.

The compression algorithm contains two variables, STRING and CHAR. The variable CHAR contains a single character from the input data stream. The variable STRING is an array of variable length strings from the input data. The STRING array is stored in the dictionary and used by the algorithm to search for recurring strings in the input data stream.

At the top of the flowchart, (box #1) reads in the first byte from the input data stream and places it into the STRING array. The next byte in the data stream is input and placed in CHAR (box #2).

The string array containing
(STRING + CHAR) is searched in the dictionary. If it is not found, the token for STRING is output to the compressed data buffer (box #5). Next, (box #6) adds (STRING + CHAR) to the dictionary and assigned a token to it. Then (box #7) places CHAR into STRING. If (STRING + CHAR) is found in the dictionary, CHAR is concatenated into STRING (box #4) and the process begins at (box #2).

This process continues until all the data has been brought into the compression program. The dictionary is then flushed of all the strings that were constructed during the compression process.

Example

Let’s explore the LZW data compression algorithm further by looking at a simplified example. You will see that even in this example in Table 1, a respectable compression ratio can be achieved.

In iteration #1 the first character from the input data is placed into STRING (“A”). The next character (“A”) is placed into CHAR. A search determines that STRING+CHAR (“AA”) is not in the dictionary, so the character (“A”) is sent to the Encoded Output and the string (“AA”) is added to the dictionary with a token value of 27. Then, the character in CHAR is transferred to STRING (“A”).

Iteration #2 begins by bringing the next character into CHAR (“A”). A search of the dictionary determines that STRING+CHAR (“AA”) is in the dictionary so CHAR is concatenated into STRING and the process moves to the next iteration.

Iteration #3 begins with the next character being received (“B”) and placed in CHAR. A dictionary search determines that STRING+CHAR (“AAB”) is not in the dictionary so the token from the previous search is sent to the Encoded Output (27) and CHAR (“B”) is placed into STRING (“A”).

In iteration #4 the next character is placed into CHAR (“A”). A search of the dictionary determines that STRING+CHAR (“BA”) is not in the dictionary, so the character in STRING is sent to the Encoded Output (“B”) and the string (“BA”) is added to the dictionary with a token value of 29. Then, the character in CHAR is transferred to STRING (“A”).

Iteration #5 begins with the next character being received (“B”) and placed in CHAR. A dictionary search determines that STRING+CHAR (“AABB”) is in the dictionary, so CHAR is concatenated into STRING and the process moves to the next iteration.

Iteration #6 brings in the next character (“B”). This is concatenated with STRING to form the new string (“AABB”). A dictionary search finds a match (token = 28) so the process moves to iteration #7. Here the next character is received (“B”) and is concatenated to STRING to yield (“AABB”).

A dictionary search determines that STRING+CHAR (“AABB”) is not in the dictionary, so the token value from the previous search (token = 28) is sent to the Encoded Output. Next, the STRING+CHAR (“AABB”) string is added to the dictionary with a token value of 30. Then, CHAR (“B”) is placed into STRING.

Iteration #8 begins with the next character being received (“B”) and placed into CHAR. A dictionary search determines that STRING+CHAR (“BB”) is not in the dictionary, so STRING (“B”) is sent to the Encoded Output. Next, the STRING+CHAR (“BB”) string is added to the dictionary with a token value of 31.

Table 1: A respectable compression ratio can be achieved.

<table>
<thead>
<tr>
<th>Iteration #</th>
<th>CHAR</th>
<th>STRING</th>
<th>STRING + CHAR</th>
<th>In Dictionary?</th>
<th>Encoded Output</th>
<th>Dictionary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>No</td>
<td>A</td>
<td>27 = AA</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>A</td>
<td>AA</td>
<td>Yes</td>
<td>27</td>
<td>28 = AAB</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>A</td>
<td>A</td>
<td>No</td>
<td>29</td>
<td>BA</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>A</td>
<td>AABB</td>
<td>Yes</td>
<td>30</td>
<td>AABB</td>
</tr>
<tr>
<td>5</td>
<td>A</td>
<td>B</td>
<td>B</td>
<td>No</td>
<td>31</td>
<td>BB</td>
</tr>
</tbody>
</table>

DONE

Table 2: The process starts with iteration #1, which places the first token from the compressed data (“A”) into CHAR.
token value of 31. Then, CHAR ("B") is placed into STRING.

In iteration #9 the next character is placed into CHAR ("B"). A search of the dictionary determines that STRING+CHAR ("BB") is in the dictionary, so CHAR is concatenated into STRING and the process moves to the next iteration.

The end of the data stream has been reached in iteration #10. So, the token from the previous match in the dictionary search is sent to the Encoded Output (31). In this example the data length was reduced from ten characters down to six. This results in a compression ratio of 40 per cent - not bad for a small example. In larger data with redundant strings the compression ratio would be much higher.

Data decompression
As shown in Figure 3, the decompression algorithm is responsible for reading in the LZW compressed data and converting it back to its original form. It does this through a novel iterative process, which recovers the original data by dynamically replicating the compression program’s dictionary.

The decompression program starts with the first entries of its dictionary initialized to all the characters in the original data. It then reads each character from the compressed data, which are merely pointers to the dictionary, and uses each pointer to retrieve uncompressed character strings from its dictionary and writes them to a decoded output buffer. It also builds its dictionary in the same way as the compression program.

Example
Let’s take the compressed data from the example above and decompress it using the algorithm shown in the flowchart in Figure 3.

As shown in Table 2, the process starts with iteration #1, which places the first token from the compressed data ("A") into CHAR. The token is used as a pointer into the dictionary where the element ("A") for that token is retrieved and placed into the variable ELEMENT. This variable is then sent to the Decoded Output. New STRING is set equal to the contents of Element.

Iteration #2 begins by moving New STRING into Last STRING. The next token from the compressed data is placed into CHAR (27). A dictionary search determines that the token in CHAR is not in the dictionary.

As a result, the variable ELEMENT is set equal to the contents of Last STRING plus the first character of Last STRING ("AA"). The contents of ELEMENT are then sent to the Decoded Output buffer. The contents of Last STRING plus the first character in ELEMENT are entered into the dictionary (token 27 = AA). Then, the contents of ELEMENT are placed into New String.

This process is continued from iteration #3 until all the tokens have been read and processed. In the end, the Decoded Output buffer will contain the original data without any loss in characters.

Multiple data buffers
In some applications it may be desirable to pack multiple data buffers into one compressed format. This is easily accomplished with a slight modification to the LZW compression and decompression algorithms.

Simply designate a unique token as a “pseudo” End of Record identifier. This token (shown in Figure 4) can be any value, as long as both the compression and decompression programs reserve the same token to represent the pseudo EOR identifier.

Dictionary structure
After studying the LZW compression and decompression algorithm you may have noticed something interesting about the dictionary. Each successive token generated by the algorithm is merely the previous token plus one extra character. This subtle behavior can be exploited to create a more efficient dictionary structure (Figure 5).

With some minor changes to the search algorithm a more efficient dictionary structure can be created.

Conclusion
As is true of any compression technique, it is important to make sure that the structure of the data is conducive to the LZW compression algorithm. Very impressive compression ratios can be achieved with the LZW algorithm, as long as the data is structured properly.

Another thing to consider is the execution time of the LZW algorithm. Most of the algorithm’s time is involved in building and searching strings in the dictionary. Some applications may not work well with the time required to compress the data. If the data is the right fit and your system can tolerate the time required to run the algorithm, the LZW compression algorithm might be the right fit for your application.