DATA SHEET LZS221-C

Version 6 Data
Compression Software

hi/fn

Hi/fn™ supplies two of the Internet's most important raw materials: compression and encryption. Hi/fn is also the world's first company to put both on a single chip, creating a processor that performs compression and encryption at a faster speed than a conventional CPU alone could handle, and for much less than the cost of a Pentium or comparable processor.

As of October 1, 1998, our address is:

Hi/fn, Inc.
750 University Avenue
Los Gatos, CA 95032
info@hifn.com
http://www.hifn.com
Tel: 408-399-3500
Fax: 408-399-3501

Hi/fn Applications Support Hotline:
408-399-3544

Table of Contents

Figures

THIS PAGE INTENTIONALLY BLANK

1 Product Description

The LZS221-C Data Compression Software Library provides a processor independent software implementation of the Hi/fn LZS® algorithm in a C source code format. The software is compatible with ANSI C.

Figure 1 on page 9 illustrates the compression speed of this library.

This library supports the simultaneous use of multiple compression and decompression histories. Each history is completely independent of other histories. In addition, this software is re-entrant.

LZS221-C is fully compatible with Hi/fn's data compression compressor chips along with the multi-history features. This library is also compatible with other members of the LZS221 software family. Files compressed or decompressed with hardware or software may be compressed or decompressed interchangeably with hardware or software.

Assembly language optimized implementations for some specific processors are also available. Consult Hi/fn for more information.

Features

- Hi/fn LZS compression format
- Multiple history support
- Adjustable compression speed vs. ratio
- High performance
- Cross compatible with other Hi/fn LZS compression software and hardware
- Interface similar to Version 4

New in version 6

- Special faster modes
- Flexible memory requirements
- Able to process fragments buffers

2 LZS221-C Files

The LZS221-C library is composed of several files. They are summarized below:

LZS.H - This header file contains the function prototypes and constant definitions. This header file should be included in all source modules that access the LZS221-C library. This file may be modified by the implementor. There are some compile-time switches that may be selected based on the characteristics of the processor and of the application. These switches may be implemented by modifying this file, or by using compiler options. These settings are described in detail in the *Compile-Time Options* section.

HIFNUTIL.H – This file contains function prototypes of functions inside HIFNUTIL.C. This file may be modified by the implementor.

HIFNDEFS.H – This file contains machine specific definitions used by LZS221-C, and the algorithm libraries. This file may be modified by the implementor, for example, to redefine non-machine specific constants such as u32b, and to define the switches needed to change the endianess and alignment.

HIFNUTIL.C - This file includes code which utilizes ANSI-C utilities that are required by LZS221-C, which may not be available in an embedded environment. Implementers may modify this file to redefine functions to call their own routines.

LZSI.H – This file includes internal function prototypes and constant definitions. This file must not be modified by the implementor.

LZSC.C - This source file contains the functions required for compression operations. This file must not be modified by the implementor.

LZSD.C - This source file contains the functions required for decompression operations. This file must not be modified by the implementor.

3 Function Summary

Functions related to data compression processing are:

LZS_C_SizeOfCompressionHistory - Returns amount of memory required for each compression history.

LZS_C_InitializeCompressionHistory - Initializes a compression history.

LZS_C_Compress - Compresses a block of data.

Functions related to data decompression are:

LZS_C_SizeOfDecompressionHistory - Returns amount of memory required for each decompression history.

LZS_C_InitializeDecompressionHistory - Initializes a decompression history.

LZS_C_Decompress - Decompresses a block of data.

4 Compile-Time Options

There are several user-selectable compile-time options available in the LZS.H and HIFNDEFS.H header files. These switches may be implemented by modifying this file, or by using compiler options.

Please note that no compiler options affect Hi/fn product cross compatibility.

4.1 HIFN_FAR

This constant (in the HIFNDEFS.H file) is used as a pointer type modifier for memory access. Suggested values are listed below. This constant can contain any value and should be based on the requirements of the compiler being used. For example, to access the type: unsigned char *, some compilers need the first example and other compilers need the second example.

[blank] - If left blank, then no modifier is used. This would produce “unsigned char *” as a memory pointer. This is the default.

__far - This would produce “unsigned char __far *” as a memory pointer. This may be useful for Intel target CPUs.

4.2 LZS_C_FOOTPRINT

This constant (in the LZS.H file) affects the size of the memory requirements per context. The performance of the object code and the compression ratio are affected in a minor way. Figure 2 summarizes the effects of LZS_C_FOOTPRINT on performance and history size.

LZS_C_FOOTPRINT_10 - This footprint setting is a fairly high speed, high compression ratio setting that takes up a modest amount of memory for its history. This is the default setting.

LZS_C_FOOTPRINT_20 - This setting is nearly identical to the LZS_C_FOOTPRINT_10 except that the history size of LZS_C_FOOTPRINT_10 varies widely with respect to the integer size of the platform that the code is compiled on. When using a 32 bit or greater platform this setting provides a substantial savings in history size over LZS_C_FOOTPRINT_10, with no loss in compression ratio and a moderate loss of speed.

LZS_C_FOOTPRINT_30 - This setting yields a smaller footprint, but it is slower than the LZS_C_FOOTPRINT_20 setting. It also has poorer compression ratios for performance settings 0-2.

LZS_C_FOOTPRINT_40 - This setting yields an even smaller footprint than the LZS_C_FOOTPRINT30 setting, and is a little bit slower (especially with small buffers). However it should yield the same compression ratio as the LZS_C_FOOTPRINT30.

LZS_C_FOOTPRINT_50 - This setting has the absolutely smallest per-history memory footprint, with the cost of having the worst speed and compression ratio.

Figure 2 shows that the memory footprints requirements change when

LZS_C_PERFORMANCE is less than or equal to 2 and when LZS_C_PERFORMANCE is greater than or equal to 3. There is a footprint size difference between LZS_C_PERFORMANCE is set to 2 and when LZS_C_PERFORMANCE is set to 3.

The default value of this compiler option is LZS_C_FOOTPRINT_10.

The system memory requirements are set to one of five footprint settings by defining the LZS_C_FOOTPRINT switch either inside the LZS.H file or by compiler option.

4.3 Byte Ordering

One of the following two constants (in the HIFNDEFS.H file) must be defined to the byte ordering used by the processor. The only valid values for this constant are the following:

HIFN_LITTLE_ENDIAN - Least significant bytes first. This is the default.

HIFN_BIG_ENDIAN - Most significant bytes first.

4.4 HIFN_ALIGNED

This constant (in the HIFNDEFS.H file), if defined, will produce a version of the library that defines type-aligned memory accesses. A type-aligned memory access restricts accesses to memory addresses that are evenly divisible by the size of the data being accessed. A u8b may reside at any address, a u16b only at even addresses, and a u32b only on a quad byte boundary. This is required for some RISC processors. The default is that HIFN_ALIGNED is not defined. Defining this constant may slow performance slightly.

4.5 LZS_C_PERFORMANCE

This constant (in the LZS.H file) specifies an compile time setting for controlling the amount of time that the Compress function will spend compressing the current buffer of data. Smaller values for the LZS_C_PERFORMANCE switch will force faster execution of the Compress function at the cost of compression ratio.

There is a footprint size difference between LZS_C_PERFORMANCE is set to 2 and when LZS_C_PERFORMANCE is set to 3. The memory footprints requirements change when LZS_C_PERFORMANCE is less than or equal to 2 and when LZS_C_PERFORMANCE is greater than or equal to 3. A value of 0 in the LZS_C_PERFORMANCE column of Figure 2 reflects the footprint size for LZS_C_PERFORMANCE settings of 0 to 2. A value of 6 in the LZS_C_PERFORMANCE column of Figure 2 reflects the footprint size for LZS_C_PERFORMANCE settings of 3 to 6.

The LZS_C_PERFORMANCE compile-time switch has seven possible value settings. The valid range for the LZS_C_PERFORMANCE switch is 0 through LZS_C_MAXIMUM_ PERFORMANCE_VALUE. The default value of this compiler option is LZS_C_MAXIMUM_ PERFORMANCE_VALUE.

Note: the value of LZS_C_MAXIMUM_ PERFORMANCE_VALUE for this version of code is 6.

5 Constants, Types, & Bits

In addition to the compile-time options described previously, there are many constants defined in the LZS221-C source code that are referred to in this document. A complete list of such constants is in the HIFNDEFS.H and LZS.H header files. See the function definitions in this document for further information concerning these constants.

LZS_C_DESTINATION_EXHAUSTED
LZS_C_DESTINATION_FLUSH
LZS_C_DESTINATION_MINIMUM
LZS_C_END_MARKER
LZS_C_FLUSHED
LZS_C_INVALID
LZS_C_SAVE_HISTORY
LZS_C_SOURCE_EXHAUSTED
LZS_C_SOURCE_FLUSH
LZS_C_UPDATE_HISTORY

Note: All unused bits in function return values must be ignored. All unused bits in input parameters must be set to zero.

u32b is a type definition which is defined to be a 32-bit unsigned data type for the target compiler.

u8b is a type definition which is defined to be a 8-bit unsigned data type for the target compiler.

All bits that are reserved must be written with zeros and ignored when read.

6 Performance

Figure 1 lists the approximate speed of compression and decompression. This performance is based on compressing a typical ASCII text file. The LZS _ C_PERFORMANCE is set to zero, and the LZS_C_FOOTPRINT variable is set to LZS_C_FOOTPRINT_10 constant.

Processor	compress (Kbytes/s)	decompress (Kbytes/s)
Pentium 200 MMX	5,020	6,374

Figure 1. Typical speed

The LZS_C_PERFORMANCE and LZS_C_FOOTPRINT settings control speed vs. compression ratio and history size trade-off within the LZS_C_Compress function. Figure 2 demonstrates how these parameters affect the overall performance of compression.

The LZS_C_PERFORMANCE and LZS_C_FOOTPRINT settings affect neither the decompression speed nor the decompression memory requirements.

These two examples use the standard text file of the U.S. Constitution in 1500 byte packet sizes running on a Pentium 200 MMX CPU. The code was compiled under Microsoft's Visual C++ v4.20 with full speed optimizations turn on using the "Pentium" processor model.

LZS_C_PERFORMANCE	LZS_C_FOOTPRINT	Compress speed (Kbytes/s)	Compression ratio	Approximate compress/decompress history size (Kbytes)
0	FOOOTPRINT_10	5,020	1.69	12/4 (32-bit compiler) 8/4 (16-bit compiler)
0	FOOOTPRINT_20	4,345	1.69	8/4 (16- or 32-bit compiler)
0	FOOOTPRINT_30	4,273	1.67	6/4 (16- or 32-bit compiler)
0	FOOOTPRINT_40	4,206	1.67	5/4 (16- or 32-bit compiler)
0	FOOOTPRINT_50	4,012	1.60	3.5/4 (16- or 32-bit compiler)
6	FOOOTPRINT_10	1,281	2.34	20/4 (32-bit compiler) 12/4 (16-bit compiler)
6	FOOOTPRINT_20	1,276	2.34	12/4 (16- or 32-bit compiler)
6	FOOOTPRINT_30	1,237	2.34	10/4 (16- or 32-bit compiler)
6	FOOOTPRINT_40	1,224	2.34	9/4 (16- or 32-bit compiler)
6	FOOOTPRINT_50	1,373	2.08	5.5/4 (16- or 32-bit compiler)

Figure 2. Effect of performance parameters

7 Hi/fn LZS Compression

The Hi/fn LZS compression algorithm compresses and decompresses data without sacrificing data integrity. Hi/fn LZS compression reduces the size of data by replacing redundant sequences of characters with tokens that represent those sequences. When the data is decompressed, the original sequences are substituted for the tokens in a manner that preserves the integrity of all data. Hi/fn LZS is "lossless" and differs significantly from "lossy" schemes, such as those used often for video images, which discard information that is deemed unnecessary.

The efficiency of data compression depends on the degree of redundancy within a given file. Compression ratios of up to 30:1 are possible, but an average compression ratio for mass storage applications is typically 2:1. For data communication applications, a compression ratio of 3:1 is more common. The compression ratio, CPU performance, and system resources can be adjusted to yield optimal system throughput. Refer to App-0022, "Data Compression Performance Analysis in Data Communications" for details.

8 Compression & Decompression Histories

This software requires a reserved block of memory in order to calculate and maintain compression information. This is referred to as a "history". The compression operation requires a compression history, and the decompression operation requires a separate decompression history.

Some applications may want to maintain multiple compression and decompression histories. For example a data communications product may associate a different history for each data channel. This may be used to maximize the redundancy in each individual history, which in turn maximizes the compression ratio that is obtained.

8.1 History Maintenance

Before a history may be used for the first time, it must be initialized. This is accomplished using the LZS_C_InitializeCompressionHistory or LZS_C_InitializeDecompressionHistory commands. This will place the history in a *start state*. A start state allows the history to be used when starting to process a new block of data. For multiple histories, each history must be initialized to the start state before it can be used for compression or decompression.

To properly finish compressing a block of data, a *flush* operation must be performed. A flush operation forces the compression algorithm to complete the compression of all the data it has read from the source buffer, and to append a unique end marker at the end of the compressed data. A flush operation guarantees that all the data read by the compression algorithm will be represented in the compressed data stream. A flush operation also places a compression history into a start state.

Sometimes, it is desirable to process a block of data in several smaller blocks (or sub-blocks). This allows the use of smaller source and destination buffers. The LZS_C_Compress function allows for this if both the LZS_C_SOURCE_FLUSH and LZS_C_DESTINATION_FLUSH flags are set to zero. It is important to note that when the LZS_C_Compress function returns in this condition, the compression history is not in a start state, but rather in a *continue* state. The LZS_C_Compress function can be called multiple times without requiring a flush operation. In order to properly terminate processing the complete block of data, the LZS_C_SOURCE_FLUSH or LZS_C_DESTINATION_FLUSH bit must be set to one in the LZS_C_Compress function call for the last sub-block of data. If this is not done during the last call to LZS_C_Compress, an alternative is to make an additional call to LZS_C_Compress with the size of the source buffer set to zero, and the LZS_C_SOURCE_FLUSH bit set to one. Note: This last call will produce destination data.

In some situations, you may need to set a compression history into a start state without regard to the data that has already been compressed. In this case, the LZS_C_Compress function can be called with the size of the source buffer set to zero, the size of the dest buffer to LZS_C_DESTINATION_MINIMUM, and the LZS_C_SOURCE_FLUSH bit set to one and the LZS_C_SAVE_HISTORY bit set to zero. Alternatively, the LZS_C_InitializeCompressionHistory function may be called (which is slightly slower).

9 LZS_C_SizeOfCompressionHistory

u32b HIFN_FAR LZS_C_SizeOfCompressionHistory(void);

This function must be called to determine the number of bytes required for one compression history. If multiple compression histories are to be used, simply multiply the value returned by this function by the number of compression histories desired.

Note: For informational purposes only, the approximate size of each compression history is provided in Figure 2. This is informational only, and subject to change. The LZS_C_SizeOfCompressionHistory function must be used to determine the actual byte count.

10 LZS_C_InitializeCompressionHistory

```
void HIFN_FAR LZS_C_InitializeCompressionHistory(
void HIFN_FAR *history                    /* Pointer to compression history */
);
```

This function must be called to initialize a compression history before it can be used with the LZS_C_Compress function. Each compression history must be initialized separately. Each history is typically only initialized once, although a compression history may be initialized at any time if desired.

If this function is called with a compression history that has been used previously, the history will be re-initialized to its beginning state. Any pending compression data within this compression history will be lost.

The *history parameter is a pointer to the memory previously allocated by the user for a compression history. The size of this allocated memory must be determined by the LZS_C_SizeOfCompressionHistory function.

11 LZS_C_Compress

```
u32b HIFN_FAR LZS_C_Compress(
u8b HIFN_FAR * HIFN_FAR *source,          /* Pointer to pointer to source buffer */
u8b HIFN_FAR *  HIFN _FAR *destination,   /* Pointer to pointer to destination buffer */
u32b  HIFN _FAR *sourceCount,             /* Pointer to source count */
u32b  HIFN _FAR *destinationCount,        /* Pointer to destination buffer size */
void  HIFN _FAR *history,                 /* Pointer to compression history */
u32b flags                                /* Special flags */
);
```

This function will compress data from the source buffer into the destination buffer. The function will stop when sourceCount bytes have been read from the source buffer or when destinationCount bytes (or slightly less than destinationCount bytes) have written to the destination buffer. A flush operation may occur under certain circumstances defined below.

The value of sourceCount will decrement and *source will increment for each byte that is read from the source buffer. The value of destinationCount will decrement and *destination will increment for each byte that is written to the destination buffer.

The valid range of sourceCount is 0 through 0x07FFFFFF. The valid range of destinationCount is LZS_C_DESTINATION_MINIMUM through 0x07FFFFFF. If this function is called with destinationCount less than LZS_C_DESTINATION_ MINIMUM, the function will immediately terminate without performing any compression and the return value will be LZS_C_INVALID.

If the source buffer exhausts (meaning all data has been read from the source buffer), then the LZS_C_SOURCE_EXHAUSTED flag in the return value will be set when the function returns. If the destination buffer exhausts (meaning all data has been written to the destination buffer), then the LZS_C_DESTINATION_EXHAUSTED flag in the return value will be set when the function returns. Both conditions may be set simultaneously.

If the LZS_C_SOURCE_FLUSH bit in the flags parameter is set and the source buffer exhausts (sourceCount reaches zero), then a flush operation will occur. If the LZS_C_DESTINATION_FLUSH bit in the flags parameter is set and the destination buffer exhausts (destinationCount less than LZS_C_DESTINATION_MINIMUM), then a flush operation will also occur. The value of destinationCount may not reach zero when the LZS_C_Compress function returns. This is due to the unknown amount of extra bytes that the compression engine needs to output during the flush operation.

If both LZS_C_SOURCE_FLUSH and LZS_C_DESTINATION_FLUSH bits are set, then when either source or destination buffers exhaust a flush operation will occur.

15	14	13	12	11	10	9	8
0	0	0	0	0	0	0	0
7	6	5	4	3	2	1	0
0	0	0	0	0	LZS_C_SAVE_ HISTORY	LZS_C_DEST_ FLUSH	LZS_C_SOURCE_ FLUSH

Figure 3. LZS_Compress flags parameter

The values of the flush bits cannot be changed between successive LZS_C_Compress function calls until the corresponding buffer is exhausted. That is, the LZS_C_SOURCE_FLUSH bit cannot change until after the LZS_C_SOURCE_EXHAUSTED flag is returned, and the LZS_C_DESTINATION_FLUSH bit cannot change until after the LZS_C_DESTINATION_EXHAUSTED flag is returned. This is independent of whether a flush operation actually occurs.

A flush operation will force any intermediate data out to the destination buffer, and will append an end marker to the destination buffer.

When the function returns after a flush operation occurs, both the *source and *destination pointers, as well as the sourceCount and destinationCount counters, will be updated. The LZS_C_SOURCE_EXHAUSTED and the LZS_C_DESTINATION_EXHAUSTED flags will be set appropriately. Also, the LZS_C_FLUSHED bit in the return value will be set to 1.

When the function returns without a flush operation having occurred, then the following values are returned.

If the source buffer exhausts, then the sourceCount counter will be 0, the *source pointer will point to 1 byte beyond the last byte processed in the source buffer, and the LZS_C_SOURCE_EXHAUSTED flag will be set to one in the return value.

If the source buffer does not exhaust, the *source pointer and sourceCount counter return values will be returned as the values the function was called with. The source buffer is still in use by the compression engine, and the original allocated source buffer will be used in the next function call. The actual pointer and counter values are stored in the compression history area, and the value of the *source and sourceCount calling parameters for the next function call are a "don't care". Also, the LZS_C_SOURCE_FLUSH bit must not change value in the next call.

If the destination buffer exhausts, then the destinationCount counter will be 0, the *destination pointer will point to 1 byte beyond the last byte processed in the destination buffer, and the LZS_C_DESTINATION_ EXHAUSTED flag will be set to one in the return value.

If the destination buffer does not exhaust, the *destination pointer and destinationCount counter return values will be returned as the values the function was called with. The destination buffer is still in use by the compression engine, and the original allocated destination buffer will be used in the next function call. The actual pointer and counter values are stored in the compression history area, and the value of the *destination and destinationCount calling parameters for the next function call are a "don't care". Also, the LZS_C_DESTINATION_FLUSH bit must not change value in the next call.

If the function terminates with both source and destination buffers exhausted then both the LZS_C_SOURCE_EXHAUSTED and LZS_C_DESTINATION_EXHAUSTED flags will be set in the return value and all counters and pointers will be updated.

Additional calls to the LZS_C_Compress function may be made to compress additional data. When more than one call to the LZS_C_Compress function is made, the compressed data (when appended together with the compressed data of the other function calls) will appear as if a single call were made to the LZS_C_Compress function.

The pseudocode in Figure 4 illustrates an example of how to call this function. If the LZS_C_SAVE_HISTORY bit of the flags parameter is set to zero, the Compression History will be cleared at the end of a flush operation. If this bit is set to one, the Compression History will NOT be cleared. This will allow a higher compression ratio for the next block to be compressed because it will continue to use the same history information. Note: Blocks must be decompressed in the same order as they were compressed if the Compression History has not been cleared between blocks during compression. If LZS_C_SOURCE_FLUSH and LZS_C_DESTINATION_FLUSH bits in the flags parameter are both zero, the LZS_C_SAVE_HISTORY bit will be ignored.

```
returnCode = LZS_C_DESTINATION_EXHAUSTED | LZS_C_SOURCE_EXHAUSTED;
flags = flagDefault& ~LZS_C_SOURCE_FLUSH & ~LZS_C_DEST_FLUSH;
sourceSize = 0; destSize = 0;
while (!(returnCode & LZS_C_FLUSHED))
{
        if (returnCode & LZS_C_SOURCE_EXHAUSTED)
        {
                Read a block of data into the source buffer;
                sourceSize += sourceCount;
                if (last block of data)
                        flags |= LZS_C_SOURCE_FLUSH;
        }
        if (returnCode & LZS_C_DESTINATION_EXHAUSTED)
        {
                Allocate a new destination buffer;
                destinationCount = COMP_BUFFER_SIZE;
        }
        returnCode = LZS_C_Compress(&source, &destination, &sourceCount,
                        &destinationCount, compHistory, flags, performance);
        if (returnCode & (LZS_C_DESTINATION_EXHAUSTED | LZS_C_FLUSHED))
        {
                destinationCount = COMP_BUFFER_SIZE - destinationCount;
                destSize += destinationCount;
                Write destination buffer to output device;
        }
}
```

Figure 4. LZS_C_Compress example pseudocode

The return value will be LZS_C_INVALID (zero) if the any of the calling parameters are invalid. The LZS_C_SOURCE_EXHAUSTED bit in the return value will be set to one if the function has been terminated by sourceCount reaching zero. The LZS_C_DESTINATION_EXHAUSTED bit in the return value will be set to one if the function has been terminated by destinationCount reaching (or almost reaching) zero. Both of these bits may be set simultaneously. The LZS_C_FLUSHED bit will be set in the return value if a flush operation has taken place. At termination *source and *destination pointers, and sourceCount, and destinationCount values may be updated depending on the conditions discussed above.

Note: For this version of the software, the value of LZS_C_DESTINATION_MINIMUM is 16. This value is specified here for information purposes only. This value may change in future versions. Do not write software that relies on a particular value of LZS_C_DESTINATION_MINIMUM.

15	14	13	12	11	10	9	8
x	x	x	x	x	x	x	x
7	6	5	4	3	2	1	0
x	x	x	x	x	LZS_C_ FLUSHED	LZS_C_DESTINATION_ EXHAUSTED	LZS_C_SOURCE_ EXHAUSTED

Figure 5. LZS_C_Compress return value

12 LZS_C_SizeOfDecompressionHistory

```
u32b HIFN_FAR LZS_C_SizeOfDecompressionHistory(void);
```

This function must be called to determine the number of bytes required for one decompression history. If multiple decompression histories are to be used, simply multiply the value returned by this function by the number of decompression histories desired.

Note: For informational purposes only, the approximate size of each decompression history is approximately 4K bytes. This is informational only, and subject to change. The LZS_C_SizeOfDecompressionHistory function must be used to determine the actual byte count.

13 LZS_C_InitializeDecompressionHistory

```
void HIFN_FAR LZS_C_InitializeDecompressionHistory(
void HIFN_FAR *history                      /* Pointer to decompression history */
);
```

This function must be called to initialize a decompression history before it can be used with the LZS_C_Decompress function. Each decompression history must be initialized separately. Each history is typically only initialized once, although a decompression history may be initialized at any time if desired.

The *history parameter is a pointer to the memory previously allocated by the user for a decompression history. The size of this allocated memory must be determined by the LZS_C_SizeOfDecompressionHistory function.

14 LZS_C_Decompress

```
u32b HIFN_FAR LZS_C_Decompress(
u8b HIFN_FAR *  HIFN _FAR *source,         /* Pointer to pointer to source buffer */
u8b  HIFN _FAR *  HIFN _FAR *destination, /* Pointer to pointer to destination buffer */
u32b  HIFN _FAR *sourceCount,              /* Pointer to source count */
u32b  HIFN _FAR *destinationCount,         /* Pointer to destination buffer size */
void  HIFN _FAR *history,                  /* Pointer to decompression history */
u32b flags                                 /* Special flags */
);
```

This function will decompress data from the source buffer into the destination buffer. The function will stop when sourceCount bytes have been read from the source buffer or when destinationCount bytes have been written to the destination buffer or if an end marker is encountered.

sourceCount will decrement and *source will increment when each byte is read from the source buffer. destinationCount will decrement and *destination will increment when each byte is written to the destination buffer.
The valid range of sourceCount is 0 through 0x07FFFFFF. The valid range of destinationCount is 0 through 0x07FFFFFF.

If the source buffer exhausts (meaning all data has been read from the source buffer), the LZS_C_SOURCE_ EXHAUSTED bit in the return value will be set to one. If destination buffer exhausts (meaning all data has been written to the destination buffer), the LZS_C_DESTINATION_ EXHAUSTED bit in the return value will be set to one. If an end marker has been detected, the LZS_C_END_MARKER bit in the return value will be set to one. More than one bit may be set in the return value.

If the function terminates due to end marker being detected, then all counters and pointers will be updated. In these cases *source and *destination pointers will point to the next bytes to be processed, sourceCount will indicate the number of bytes remaining in the source buffer to be processed, destinationCount will indicate the number of unused bytes (free space) in the destination buffer.

If the function terminates due to source buffer being exhausted, *source pointer will point to one byte beyond the last byte processed and sourceCount will be 0. In this case the *destination pointer and the destinationCount counter return values will be returned as the values the function was called with. The destination buffer is still in use by the decompression engine, and the original allocated destination buffer will be used in the next function call. The actual pointer and counter values are stored in the decompression history area, and the value of the *destination and destinationCount calling parameters for the next function call are a “don’t care”.

15	14	13	12	11	10	9	8
0	0	0	0	0	0	0	0
7	6	5	4	3	2	1	0
0	0	0	0	0	LZS_C_SAVE_ HISTORY	LZS_C_UPDATE_ HISTORY	0

Figure 6. LZS_C_Decompress flags parameter

If the function terminates due to the destination buffer being exhausted, *destination pointer will point to one byte beyond the last byte processed and destinationCount will be 0. In this case the *source pointer and the sourceCount counter return values will be returned as the values the function was called with. The source buffer is still in use by the compression engine, and the original allocated source buffer will be used in the next function call. The actual pointer and counter values are stored in the decompression history area, and the value of the *source and sourceCount calling parameters for the next function call are a “don’t care”.

If the function terminates with both source and destination buffers exhausted, then all counters and pointers will be updated.

Additional calls to the LZS_C_Decompress function may be made to decompress additional data. When more than one call to the LZS_C_Decompress function is made, the decompressed data (when appended together with the decompressed data of the other function calls) will appear as if a single call were made to the LZS_C_Decompress function.

The pseudocode in Figure 7 illustrates an example of how to call this function.

If it is desired to terminate processing a block of data prior to the end of the data block, simply call the LZS_C_InitializeDecompressionHistory function.

```
returnCode = LZS_C_DESTINATION_EXHAUSTED | LZS_C_SOURCE_EXHAUSTED;
flags = flagDefault;
sourceSize = 0; destSize = 0;
while (!(returnCode & LZS_C_END_MARKER))
{
        if (returnCode & LZS_C_SOURCE_EXHAUSTED)
        {
                Read a block of data into the source buffer;
                sourceSize += sourceCount;
        }
        if (returnCode & LZS_C_DESTINATION_EXHAUSTED)
                Allocate a new destination buffer;
        returnCode = LZS_C_Decompress(&source, &destination, &sourceCount,
                               &destinationCount, decompHistory, flags);
        if (returnCode & (LZS_C_DESTINATION_EXHAUSTED | LZS_C_END_MARKER))
        {
                destinationCount = (RAW_BUFFER_SIZE - destinationCount);
                destSize += destinationCount;
                Write destination buffer to output device;
        }
}
```

Figure 7. LZS_C_Decompress pseudocode example

Normally, the LZS_C_SAVE_HISTORY bit in the flags parameter should be set. This is required to ensure that the decompression history is properly updated between calls. The LZS_C_SAVE_HISTORY bit may be set to zero, if it is known that the compression history associated with the current decompression history was cleared. This will improve decompression speed when not maintaining history.

Note: Blocks must be decompressed in the same order as they were compressed if the Compression History has not been cleared between blocks during compression (i.e. the LZS_C_SAVE_HISTORY bit was set during LZS_C_Compress function calls).

If the LZS_C_UPDATE_HISTORY bit in the flags parameter is set to one, the source data is treated as if it were uncompressed data. The decompression history will be updated to reflect this data. The data in the source buffer will be moved into the destination buffer. This bit may only be set after a decompression history is initialized or after an end marker is detected. The sourceCount and destinationCount parameters must be set to the same value in the function call when the LZS_C_UPDATE_HISTORY bit is set. All counters and pointers will be updated when the function returns.

15	14	13	12	11	10	9	8
x	x	x	x	x	x	x	x
7	6	5	4	3	2	1	0
x	x	x	x	x	LZS_C_END_ MARKER	LZS_C_DESTINATION_ EXHAUSTED	LZS_C_SOURCE_ EXHAUSTED

Figure 8. LZS_C_Decompress return value

Note: If the compressed data stream used as source for the LZS_C_Decompress function has been corrupted (for example, due to a communication link error), memory outside the range of the decompression history could be accessed (read). Specifically, memory could be read up to 2 KBytes before the beginning of the decompression history, or up to 2 KBytes before the beginning of the destination buffer. If the compressed data stream has no errors, then memory outside the decompression history will not be accessed.