DATA SHEET MPPC-386

Version 6 Data Compression Software

hifn

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

Disclaimer

Hi/fn reserves the right to make changes to its products or to discontinue any semiconductor product or service without notice, and advises its customers to obtain the latest version of relevant information to verify, before placing orders, that the information being relied on is current.

Hi/fn warrants performance of its semiconductor products and related software to the specifications applicable at the time of sale in accordance with Hi/fn's standard warranty. Testing and other quality control techniques are utilized to the extent Hi/fn deems necessary to support this warranty. Specific testing of all parameters of each device is not necessarily performed, except those mandated by government requirements.

Certain applications using semiconductor products may involve potential risks of death, personal injury, or severe property or environmental damage ("Critical Applications").

HI/FN SEMICONDUCTOR PRODUCTS ARE NOT DESIGNED, INTENDED, AUTHORIZED, OR WARRANTED TO BE SUITABLE FOR USE IN LIFE-SUPPORT APPLICATIONS, DEVICES OR SYSTEMS OR OTHER CRITICAL APPLICATIONS.

Inclusion of Hi/fn products in such critical applications is understood to be fully at the risk of the customer. Questions concerning potential risk applications should be directed to Hi/fn through a local sales office.

In order to minimize risks associated with the customer's applications, adequate design and operating safeguards should be provided by the customer to minimize inherent or procedural hazards.

Hi/fn does not warrant that its products are free from infringement of any patents, copyrights or other proprietary rights of third parties. In no event shall Hi/fn be liable for any special, incidental or consequential damages arising from infringement or alleged infringement of any patents, copyrights or other third party intellectual property rights.

"Typical" parameters can and do vary in different applications. All operating parameters, including "Typicals," must be validated for each customer application by customer's technical experts.

The use of this product may require a license from Motorola. A license agreement for the right to use Motorola patents may be obtained through Hi/fn or directly from Motorola.

DS-0009-02 (9/98) © 1997-1998 by Hi/fn, Inc., including one or more U.S. patents No.: 4,701,745, 5,003,307, 5,016,009, 5,126,739, 5,146,221, 5,414,425, 5,414,850, 5,463,390, 5,506,580, 5,532,694. Other patents pending.

Table of Contents

Figures

THIS PAGE INTENTIONALLY BLANK

1 Product Description

The MPPC-386 Data Compression Software Library provides a highly optimized software implementation of the MPPC algorithm in 32-bit assembly source code for the Intel 386 family and above of processors.

Figure 2 on page 7 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.

MPPC-386 is fully compatible with Hi/fn's data compression compressor chips that support the MPPC algorithm. Files compressed or decompressed with MPPC hardware or software may be compressed or decompressed interchangeably with MPPC hardware or software.

Assembly language optimized implementations for other specific processors are also available. In addition, a C source code version is available that can be used to create a compression library for any processor. Consult Hi/fn for more information.

Features

- MPPC compression format
- Multiple history support
- High performance
- Cross compatible with other Hi/fn MPPC compression software and hardware
- Windows® NT compatible

2 LZS221-386 Files

The MPPC-386 library is composed of several MASM 6.11 compatible files and one C header file. These files are not user-modifiable. They are summarized below:

MPPC.H - This header file contains the function prototypes and constant definitions. This header file should be included in all source modules that access the MPPC-386 library.

MPPCC.ASM - This source file contains the functions required for the compression operations.

MPPCD.ASM - This source file contains the functions required for the decompression operations.

HIFNDEFS.H – This file contains machine specific definitions used by MPPC-386.

HIFNUTIL.ASM – This file includes code which utilizes memory utility functions that are required by MPPC-386.

3 Function Summary

All function names (and constants) must prepended with “MPPC_386_” to uniquely identify the version of this software library. In this data sheet, the “MPPC_386_” is missing to improve readability.

Functions related to data compression are:

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

InitCompressionHistory - Initializes a compression history.
Compress - Compresses a block of data.

Functions related to data decompression are:

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

InitDecompressionHistory - Initializes a decompression history.
Decompress - Decompresses a block of data.

4 Constants and Types

In addition to the compile-time options described previously, the following constants are defined in the MPPC.H header file. See the function definitions for further information concerning these constants.

Constant	Value
SAVE_HISTORY	0x04
INVALID	0x00
SOURCE_EXHAUSTED	0x01
DEST_EXHAUSTED	0x02
FLUSHED	0x04
RESTART_HISTORY	0x08
EXPANDED	0x10
SOURCE_MAX	8192

Note: The values listed are for this version of the software only. These values are listed here for information purposes only. These values may change in future versions. Do not write software that relies on a particular value of these constants.

Figure 1. Predefined constants

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.

u16b is a type definition which is defined to be a 16-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.

5 PERFORMANCE

The data presented in this section was generated by assembling MPPC-386.

Figure 2 lists the approximate speed of compression and decompression over a range of processors. This performance is based on compressing a typical ASCII text file. In this case a text file containing the U.S. Constitution was used.

Processor	Compress (Kbytes/s)	Decompress (Kbytes/s)
80486DX2-66	1383	2054
Pentium Pro @ 200MHz	8276	13738

Figure 2. Typical speed

6 Hi/fn MPPC Compression

The MPPC compression algorithm compresses and decompresses data without sacrificing data integrity. The MPPC algorithm 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. MPPC 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. Although very high compression ratios are possible, 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.

7 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. The decompression operation requires a 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.

7.1 History Maintenance

Before a history may be used for the first time, it must be initialized. This is accomplished using the InitCompressionHistory or InitDecompressionHistory 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. 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.

For this version of MPPC software, a flush operation must be performed on every call to the compression function.

8 SizeOfCompressionHistory

```
u16b SizeOfCompressionHistory(void);
```

This function must be called to determine the number of bytes required to be allocated 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 number of bytes required to be allocated for of each compression history is approximately 16 Kbytes. This is informational only, and subject to change. The SizeOfCompressionHistory function must be used to determine the actual byte count.

9 InitCompressionHistory

```
u16b InitCompressionHistory(
void *history                    /* Pointer to compression history */
);
```

This function must be called to initialize a compression history before it can be used with the Compress function. Each compression history must be initialized separately.

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 allocated for a compression history. The size of this allocated memory was determined by the SizeOfCompressionHistory function.

The return value will always be non-zero.

10 Compress

```
u16b Compress(
u8b * *source,                   /* Pointer to pointer to source buffer */
u8b * *dest,                     /* Pointer to pointer to destination buffer */
u32b *sourceCnt,                 /* Pointer to source count */
u32b *destCnt,                   /* Pointer to destination buffer size */
void *history                    /* Pointer to compression history */
u16b flags,                      /* Special flags */
u16b performance                 /* Performance parameter */
);
```

This function will compress data from the source buffer into the dest buffer. The function will stop when sourceCnt bytes have been read from the source buffer. A flush operation will take place after the source data has been processed.

sourceCnt will decrement and *source will increment for each byte that is read from the source buffer. destCnt will decrement and *dest will increment for each byte that is written to the dest buffer.

The valid range of sourceCnt is 0 through SOURCE_MAX.

The destination buffer (allocation size and destCnt parameter) must be large enough to hold all the compressed data. To ensure that the destination buffer is large enough to accommodate the worst case expansion, the destCnt parameter must be equal to or greater than the following formula:

(sourceCnt * 9/8) + 4

If this function is called with an invalid value of sourceCnt or destCnt, the function will immediately terminate without performing any compression and the return value will be INVALID.

If the data block expands during compression (meaning the number of bytes generated is greater than sourceCnt), then the EXPANDED flag in the return value will be set when the function returns. In this case, the destination data should be discarded, and the compression history is re-initialized automatically.

If the SAVE_HISTORY bit of the flags parameter is set to zero, the Compression History is automatically 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 was not cleared between blocks during compression.

The performance parameter is not used for this version of software. The value passed here is ignored.

The return value will be INVALID (zero) if the sourceCnt, destCnt, or flags calling parameters are invalid. The EXPANDED bit in the return value will be set to one if the function has been terminated by data expansion, and the function internally calls InitCompressionHistory. The FLUSHED and SOURCE_EXHAUSTED bits will be set if the compression operation was successful. The RESTART_HISTORY bit is information required by the decompression function. The value of the RESTART_HISTORY bit must be saved and passed to the flags parameter of the decompression function. If successful, the *source and *dest pointers, and sourceCnt, and destCnt values will be updated.

Note: If the EXPANDED bit is set to one, the InitDecompressionHistory function must be called for the corresponding decompression history.

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	SAVE_ HISTORY	0	Must be 1

Figure 3. Compress flags parameter

The pseudocode in Figure 4 illustrates an example of how to call this function. For a more detailed example, please refer to the example software supplied with this release.

```
Read a block of data into the source buffer;
returnCode = Compress(&source, &dest, &sourceCnt, &destCnt,
        compHistory, flags, performance);
Write dest buffer to output device;
```

Figure 4. Compress example pseudocode

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	EXPANDED	RESTART_ HISTORY	FLUSHED	0	SOURCE_ EXHAUSTED

Figure 5. Compress return value

11 SizeOfDecompressionHistory

```
u32b SizeOfDecompressionHistory(void);
```

This function must be called to determine the number of bytes required to allocate 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 number of bytes required to be allocated for each decompression history is approximately 8 Kbytes. This is informational only, and subject to change. The SizeOfDecompressionHistory function must be used to determine the actual byte count.

12 InitDecompressionHistory

```
u16b InitDecompressionHistory(
void *history                    /* Pointer to decompression history */
);
```

This function must be called to initialize a decompression history before it can be used with the Decompress function. In addition, this function must be called if the InitCompressionHistory function was called prior to this compressed data being produced. This would occur if the Compress function returned with the EXPANDED bit in the return value set to one for the corresponding compression operation, or if the Compress function was called with the SAVE_HISTORY bit in the flags parameter set to zero. In either of these two cases the InitDecompressionHistory function must be called prior to processing this compressed data.

The *history parameter is a pointer to the memory allocated for a decompression history. The size of this allocated memory was determined by the SizeOfDecompressionHistory function.

The return value will always be non-zero.

13 Decompress

```
u16b Decompress(
u8b * *source,          /* Pointer to pointer to source buffer */
u8b * *dest,            /* Pointer to pointer to destination buffer */
u32b *sourceCnt,        /* Pointer to source count */
u32b *destCnt,          /* Pointer to destination buffer size */
void *history           /* Pointer to decompression history */
u16b flags              /* Special flags */
);
```

This function will decompress data from the source buffer into the dest buffer. The function will stop when sourceCnt bytes have been read from the source buffer.

sourceCnt will decrement and *source will increment when each byte is read from the source buffer. destCnt will decrement and *dest will increment when each byte is written to the dest buffer.

The valid range of destCnt is 0 through SOURCE_MAX. The calling value of sourceCnt must be less than the calling value of destCnt. The value of destCnt must be equal to or greater than the actual number of raw data originally compressed. If this function is called with invalid values of sourceCnt or destCnt the function will immediately terminate without performing any compression and the return value will be INVALID.

If the value of destCnt is too small, the function will terminate with the DEST_EXHAUSTED bit set to one and the DECOMP_OK bits set to zero.

The RESTART bit in the flags parameter must contain the value returned in the RESTART bit from the compress function.

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 SAVE_HISTORY bit was set during Compress function calls).

The return value will be INVALID (zero) if the sourceCnt, or flags calling parameters are invalid. If the decompression operation is successful, and the destination buffer does not become full, the DECOMP_OK bits will be set to one. If successful, and the destination buffer is full (but does not overflow), The DECOMP_OK and the DEST_EXHAUSTED bits will be set to one. If the destination buffer overflows, only the DEST_EXHAUSTED bit will be set to one.

If successful, the *source and *dest pointers and sourceCnt and destCnt will be updated.

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	RESTART	Must be 1	0	0

Figure 6. Decompress flags parameter

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

```
Read a block of data from an input device;
returnCode = Decompress(&source, &dest, &sourceCnt, &destCnt,
        compHistory, flags);
Write destination buffer to packet memory;
```

Figure 7. Decompress example pseudocode

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	DECOMP_ OK	DEST_ EXHAUSTED	DECOMP_ OK

Figure 8. Decompress return value