Squish Developers Kit
Version 2.00

Created May 23rd, 1994

Documentation produced by Scott Dudley

TABLE OF CONTENTS

INTRODUCTION

This document serves as a reference guide for version 2.0 of the Squish Developers Kit. The primary purpose of this document is to describe the C Application Programming Interface (API) for Squish bases, also known as MsgAPI.

Information is included on rebuilding the MsgAPI source, compiling applications that use MsgAPI, and a description of each function within the MsgAPI. The standard MsgAPI distribution supports Turbo C, Borland C, WATCOM C, and Microsoft C.

This document does not provide a tutorial on using the MsgAPI, but those with a reasonable amount of programming experience should have little trouble in using MsgAPI.

In addition, this document includes information on the "feature DLL" interface that is supported by the OS/2 version of the Squish tosser/scanner program. These hooks allow third-party developers to manipulate messages when they are being processed by Squish. These hooks are called when Squish is tossing and packing messages.

Finally, this document also describes the physical layout of the Squish file format. Although use of the MsgAPI is recommended, this information should allow third-party developers to access the Squish base directly, without going through a set of API calls.

WARNING! THE SQUISH DEVELOPER'S KIT IS NOT A SUPPORTED PRODUCT. We would be interested in hearing of any problems that you have with this product, but we cannot provide technical support for MsgAPI users.

While we have tried to ensure that the material provided in this kit is correct, we also cannot guarantee that the code and documentation will function correctly on all systems.

Please read the following section for information on distribution terms and the product warranty.

COPYRIGHT AND DISTRIBUTION RESTRICTIONS

Distribution Policy

Although the MsgAPI code is copyrighted, you are granted a limited license to modify or usr MsgAPI in your own applications.

1) You may use the MsgAPI code as part of any type of application, including freeware, shareware, or commercial programs. No royalties or licensing fees are required.

2) This code must not be sold on its own. While it is permissible to sell an application that uses or contains the MsgAPI code, you may not charge any extra fee (above the base cost of the product) for just providing the user with a copy of the Squish Developers kit.

   Also, you may not charge any extra fee (above the base cost of the product) for providing Squish base support within your program.

3) If you use this code in your application, you must give credit for the MsgAPI code and indicate that "Squish" is a trademark of SCI Communications.

4) If you wish to identify your application as being compatible with the Squish message format, you must ensure that the original MsgAPI code is capable of reading and writing to the message bases that are created by your program. If the original MsgAPI code is unable to read the files created by your program, you may not label your program as Squish-compatible.

AS LONG AS THE ABOVE FOUR CONDITIONS ARE FOLLOWED, MSGAPI MAY BE INCORPORATED INTO ANY TYPE OF APPLICATION WITHOUT CHARGE, REGARDLESS OF THE NATURE OF THE APPLICATION (COMMERCIAL, SHAREWARE, OR FREEWARE).

No Warranty

BECAUSE THE SQUISH DEVELOPERS KIT (SDK) IS LICENSED FREE OF CHARGE, WE PROVIDE ABSOLUTELY NO WARRANTY. EXCEPT WHEN OTHERWISE STATED IN WRITING, SCI COMMUNICATIONS AND/OR OTHER PARTIES PROVIDE THE SDK "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF SQUISH, AND THE ACCURACY OF ITS ASSOCIATED DOCUMENTATION, IS WITH

YOU. SHOULD THE SDK OR ITS ASSOCIATED DOCUMENTATION PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

IN NO EVENT WILL SCI COMMUNICATIONS BE RESPONSIBLE IN ANY WAY FOR THE BEHAVIOUR OF MODIFIED VERSIONS OF THE SDK. IN NO EVENT WILL SCI COMMUNICATIONS AND/OR ANY OTHER PARTY WHO MAY MODIFY AND REDISTRIBUTE SDK AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY LOST PROFITS, LOST MONIES, OR OTHER SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS) THE SDK, EVEN IF SCI COMMUNICATIONS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY CLAIM BY ANY OTHER PARTY.

Author Contact Information

If you make any modifications to this code, and if you feel that the author would be interested in the changes, please forward the source code to the author.

You can contact the author at any of the addresses listed below:

FidoNet: 1:249/106
Internet: sjd@f106.n249.z1.fidonet.org
CompuServe: >INTERNET:sjd@f106.n249.z1.fidonet.org
BBS: (613) 634-3058 (V.32bis)

Surface mail:

777 Downing St.
Kingston, Ont.
Canada K7M 5N3

The author can also be reached through the FidoNet EchoMail conferences called MUFFIN (Maximus support) and TUB (Squish support).

Sending correspondence via electronic mail is strongly preferred, and the use of surface mail is discouraged. However, if you really have to send paper mail (and expect to receive a reply), please enclose a self-addressed, stamped envelope. (Users outside of Canada should include an international postal reply coupon instead of a stamp.)

DO NOT ATTEMPT TO CONTACT THE AUTHOR BY TELEPHONE! VOICE SUPPORT WILL NOT BE PROVIDED FOR THIS PRODUCT!

MSGAPI FUNCTION REFERENCE

This section describes all of the functions in the MsgAPI that can be used by application programs.

Initialization/Termination Functions

MsgOpenApi

```
sword MsgOpenApi(struct _minf *minf);
```

The MsgOpenApi function is used to initialize the MsgAPI. This function must be called before any of the other API functions are called, or else the results are undefined. This function serves to initialize any needed structures, and to prepare the message bases for use.

This function accepts one argument: a pointer to a structure containing information about the application. This structure contains the following information:

```
struct _minf
{
     /* The following fields are required for all
      * MsgAPI clients:
      */

     word req_version;
     word def_zone;
     word haveshare;

     /* The following fields are required when
      * req_version >= 1:
      */

     void OS2FAR * (MAPIENTRY *palloc)(size_t size);
     void (MAPIENTRY *pfree)(void OS2FAR *ptr);
     void OS2FAR * (MAPIENTRY *repalloc)(void OS2FAR *ptr,
                                         size_t size);

     void far * (MAPIENTRY *farpalloc)(size_t size);
     void (MAPIENTRY *farpfree)(void far *ptr);
     void far * (MAPIENTRY *farrepalloc)(void far *ptr,
                                         size_t size);
};
```

'req_version' indicates the MsgAPI revision level that the application is requesting. The compile-time revision level can always be accessed using the constant 'MSGAPI_VERSION'.

'def_zone' should contain a default FidoNet zone number.

Certain message systems, such as the FTSC-0001 *.MSG format, do not store zone information with each message. When the API encounters such a message and no zone is present, the specified zone will be used instead. A 'def_zone' of 0 indicates that nothing is to be inferred about the zone number of a message, and in that case, the API functions will return 0 as the zone number for any message with an unknown zone.

'haveshare' is automatically filled in by the internal API routines, and this flag indicates whether or not the DOS "SHARE.EXE" program is currently loaded. Note that SHARE must always be loaded to access Squish-format bases in a multitasking or network environment. This field is not used in the OS/2 version of the MsgAPI.

If 'req_version' is more than or equal to 1, the final six fields in the _minf structure must be provided. These fields are memory allocation hooks that MsgAPI will call whenever it needs to allocate memory. (If req_version is 0, or if one of the function pointers in this structure is NULL, then MsgAPI will use its own memory allocation routines.)

'palloc' is called to allocate a block of near memory. This function should behave in the same manner as the ANSI malloc() function. If this field is NULL, the MsgAPI will use its own malloc() function.

'pfree' is called to free a block of near memory. This function should behave in the same manner as the ANSI free() function. If this field is NULL, the MsgAPI will use its own free() function.

'repalloc' is called to reallocate a block of near memory. This function should behave in the same manner as the ANSI realloc() function. If this field is NULL, the MsgAPI will use its own realloc() function.

'farpalloc' is called to allocate a block of far memory. This function should behave in the same manner as the ANSI malloc() function, except that a far pointer should be returned. If this field is NULL, MsgAPI will use its own malloc() function.

'farpfree' is called to free a block of far memory. This function should behave in the same manner as the ANSI free() function, except that a far pointer should be accepted. If this field is NULL, MsgAPI will use its own free() function.

'farrepalloc' is called to reallocate a block of far memory. This function should behave in the same manner as the ANSI

realloc() function, except that a far pointer should be accepted and returned. If this field is NULL, MsgAPI will use its own realloc() function.

MsgOpenApi() returns a value of 0 if the initialization was performed successfully, and -1 if a problem was encountered.

MsgCloseApi

sword MsgCloseApi(void);

This function is used to deinitialize the MsgAPI. This function performs any clean-up actions which may be necessary, including the closing of files and releasing allocated memory. This function should be called before the application terminates.

MsgCloseApi returns a value of 0 if the API was successfully deinitialized, and -1 otherwise.

Area-Oriented Functions

MsgOpenArea

HAREA MsgOpenArea(byte *name, word mode, word type);

This function is used to open or create a message area. This function accepts three parameters:

'name' is the name of the message area to open. The contents of this string are implementation-defined. (See 'type' for more information.)

'mode' is the mode with which the area should be opened. Values for 'mode' are as follows:

MSGAREA_NORMAL	Open the message area in a normal access mode. If the area does not exist, this function fails.
MSGAREA_CRIFNEC	Open the message area in a normal access mode. If the area does not exist, the MsgAPI attempts to create the area. If the area cannot be created, this function fails.
MSGAREA_CREATE	Create the message area. If the area already exists, it is truncated or started anew with no messages. If the area cannot be created, this function

fails.

'type' specifies the type of message area to open. 'type' can have any of the following values:

MSGTYPE_SDM Star Dot MSG (SDM). This specifies a FTSC-0001 compatible access mode, and it instructs the MsgAPI to create and read Fido-compatible messages for this area. If MSGTYPE_SDM is specified, 'name' should contain the path to the *.MSG directory.

MSGTYPE_SQUISH Squish (*.SQ?) format. This specifies that the proprietary Squish message format is to be used for this area. 'name' should give the path and root name (eight characters) for the message area.

In addition, if the mask 'MSGTYPE_ECHO' is bitwise 'OR'ed with the 'MSGTYPE_SDM' value, the area in question will be treated as a FidoNet-style echomail area. This instructs the MsgAPI to keep high-water mark information in the 1.MSG file, and to stop the normal MsgAPI functions from writing to the first message in each area. Other message formats have a cleaner way of storing the high-water mark, so this mask is only required for *.MSG areas.

Other values for 'type' are currently reserved.

If this function succeeds in opening the message area, an HAREA handle is returned. This handle does not contain any information which can be used directly by the caller; all interaction should be performed through the MsgAPI functions only.

If this function fails, NULL is returned, and the global 'msgapierr' variable is set to one of the following values:

MERR_NOMEM Not enough memory for requested task

MERR_NOENT The area did not exist or could not be created.

MERR_BADF The message area is structurally damaged.

MsgCloseArea

sword MsgCloseArea(HAREA ha);

This function serves to "close" a message area. This function performs all clean-up actions necessary, such as

closing files, changing directories, and so on.

The function accepts one argument, which must be an HAREA handle returned by the MsgOpenArea function. The MsgCloseArea function should be called for each area opened by MsgOpenArea.

If the area was successfully closed, MsgCloseArea returns 0. Otherwise, a value of -1 is returned and msgapierr is set to one of the following:

MERR_BADH An invalid handle was passed to the function.

MERR_EOPEN Messages are still "open" in this area, so the area could not be closed.

MsgValidate

sword MsgValidate(word type, byte *name);

The MsgValidate function validates a particular message area, determining whether or not the area exists and if it is valid.

'type' is the type of the message area, using the same constants as specified for MsgOpenARea.

'name' is the name of the message area, using the same format as specified for MsgOpenArea.

MsgValidate returns the value 1 if the area exists and is valid. Otherwise. MsgValidate returns 0.

MsgGetHighWater

dword MsgGetHighWater(HAREA ha);

The MsgGetHighWater function returns the 'high water marker' for the current area. This number represents the highest message number that was processed by a message export or import utility.

'ha' is a message area handle, as returned by MsgOpenArea.

The high water marker is automatically adjusted when messages are killed.

The MsgGetHighWater function returns the high water mark number on success, or 0 on error and sets msgapierr to:

MERR_BADH

MsgSetHighWater

sword MsgSetHighWater(HAREA mh, dword hwm);

The MsgSetHighWater function sets the 'high water marker' for the current area.

'ha' is a message area handle, as returned by MsgOpenArea.

'hwm' is the new high water marker to use for the specified area.

The MsgGetHighWater function returns 0 on success, or -1 on error and sets msgapierr to:

MERR_BADH

MsgGetCurMsg

dword MsgGetCurMsg(HAREA ha);

The MsgGetCurMsg function returns the number of the last message accessed with MsgOpenMsg.

'ha' is a message area handle, as returned by MsgOpenArea.

MsgGetCurMsg returns the current message number on success, or 0 if there is no current message.

MsgGetNumMsg

dword MsgGetNumMsg(HAREA ha);

The MsgGetNumMsg function returns the number of messages in the current message area. On error, MsgGetNumMsg returns (dword)-1 and sets msgapierr to:

MERR_BADH

MsgGetHighMsg

dword MsgGetHighMsg(HAREA mh);

The MsgGetHighMsg function returns the number of the highest message in the specified area. On error, MsgGetHighMsg returns (dword)-1 and sets msgapierr to:

MERR_BADH

MsgLock

sword MsgLock(HAREA ha);

The MsgLock function 'locks' a message area for exclusive access. This function may enable buffering or otherwise improve performance, so it is advised that MsgLock be called if speed is a concern when accessing the area.

All of the MsgAPI functions automatically perform file locking and sharing internally, so this function is only required when high performance is necessary.

'ha' is a message area handle, as returned by the MsgOpenArea function.

MsgLock returns 0 on success. On error, MsgLock returns -1 and sets msgapierr to one of the following:

MERR_BADH

MsgUnlock

sword MsgUnlock(HAREA ha);

The MsgUnlock function unlocks a previously-locked message area.

'ha' is a message area handle, as returned by MsgOpenMsg.

MsgUnlock returns 0 on success, or it returns -1 on error and sets msgapierr to:

MERR_BADH

MsgInvalidHarea

sword MsgInvalidHarea(HAREA ha);

The MsgInvalidHarea function tests the given message area handle for validity. If the message area handle is invalid, this function returns TRUE and sets msgapierr to MERR_BADH. Otherwise, a value of FALSE will be returned. returned.

Message-Oriented Functions

MsgOpenMsg

```
HMSG MsgOpenMsg(HAREA mh, word mode, dword msgn);
```

This function "opens" a message for access, and it must be used to read from or write to a given message.

The function accepts three arguments:

'mh' is a message area handle, as returned by the MsgOpenArea function.

'mode' is an access flag, containing one of the following manifest constants:

MOPEN_CREATE	Create a new message. This mode should only be used for creating new messages.
MOPEN_READ	Open an existing message for reading ONLY.
MOPEN_WRITE	Open an existing message for writing ONLY.
MOPEN_RW	Open an existing message for reading AND writing.

'msgn' is the specified message number to open. If mode is either MOPEN_READ, MOPEN_WRITE or MOPEN_RW, the message number must currently exist in the specified area. If mode is set to MOPEN_CREATE, a value of 0 for 'msgn' indicates that a new message should be created, and assigned a number one higher than the current highest message. If 'msgn' is non-zero, but MOPEN_CREATE is set to the number of a currently-existing message, the specified message will be truncated and the new message will take its place.

For MOPEN_READ or MOPEN_RW, the following constants can also be passed in place of 'msgn':

MSGNUM_CUR	Open the last message which was accessed by MsgOpenMsg.
MSGNUM_PREV	Open the message prior to the last message accessed by MsgOpenMsg.
MSGNUM_NEXT	Open the message after the last message accessed by MsgOpenMsg.

The MsgAPI maintains the number of the last message opened by MsgOpenMsg, which is used when processing these constants. (See also MsgGetCurMsg.)

If the message was successfully opened, the MsgOpenMsg function will return a HMSG handle. Otherwise, a value of NULL is returned, and msgapierr will be set to one of the following:

MERR_NOENT
MERR_NOMEM
MERR_BADF
MERR_BADA
MERR_BADH

MsgCloseMsg

sword MsgCloseMsg(HMSG hmsg);

The MsgCloseMsg function serves to "close" a message which has been previously opened by MsgOpenMsg. All messages should be closed after use, or else data loss may result.

This function accepts a single argument, which is the message handle that was returned by MsgOpenMsg.

If the message was successfully closed, this function returns 0. Otherwise, a value of -1 is returned, and msgapierr is set to:

MERR_BADH

MsgReadMsg

dword MsgReadMsg(HMSG hmsg, dword ofs, dword bytes,
byte *text, dword cbyt, byte *ctxt);

The MsgReadMsg function is used to read a message from disk. This function can be used to read all parts of a message, including the message header, message body, and control information.

'hmsg' is a message handle, as returned by the MsgOpenMsg function. The message in question must have been opened with a mode of either MOPEN_READ or MOPEN_RW.

'msg' is a pointer to an XMSG (extended message) structure. The format of this structure is detailed in the "Squish File Format Specification" section, but it contains all of the message information that is found in the message header, including the to/from/subject fields, origination and arrival dates, 4D origination and destination addresses, and so forth. (See the appendices for specific information on the XMSG structure itself.) If the application wishes to

read the header of a given message, this argument should point to an XMSG structure. Otherwise, this argument should be NULL, which informs the API that the message header does not need to be read.

'ofs' is used for reading message text in a multiple-pass environment. This defines the offset in the message body from which the API should start reading. To start reading from the beginning of the message, a value of 0L should be given. Otherwise, the offset into the message (in bytes) should be given for this argument. If the application does not wish to read the message body, this argument should be set to 0L.

'bytes' represents the maximum number of bytes to read from the message. Fewer bytes may be read, but the API will read no more than 'bytes' during this call. (See 'text', and also this function's return value.) If the application does not wish to read the message body, this argument should be set to 0L.

'text' is a pointer to a block of memory, into which the API will place the message body. The message body will be read from the position specified by 'ofs', up to a maximum of 'bytes' bytes. If the application does not wish to read the message body, this argument should be set to NULL.

'cbyt' represents the maximum number of bytes of control information to read from the message.

'ctxt' is a pointer to a block of memory, into which the API will place the message control information. No more than 'cbyt' bytes of control information will be placed into the buffer. NOTE: unlike the message text functions, control information can only be read in one pass.

The text read by this function is free-form. The message body may or may not contain control characters, NULs, or any other sequence of characters. Messages are simply treated as a block of bytes, with no interpretation whatsoever.

In FidoNet areas, the message body consists of one or more paragraphs of text. Each paragraph is delimited by a hard carriage return, '\r', or ASCII 13. Each paragraph can be of any length, so the text should be wordwrapped onto logical lines before being displayed. If created by older applications, paragraphs may also contain linefeeds ('\n') and soft returns ('\x8d') at the end of each line, but these are optional and should always be ignored.

As an example, assume that the following stream of text was returned by MsgReadMsg():

"Hi!\r\rHow's it going? I got the new MsgAPI kit today!\r\rAnyhow, gotta run!"

The "\r" marks are carriage returns, so they indicate the end of a paragraph. Notice that the second paragraph is fairly long, so it might have to be wordwrapped, depending on the screen width. Your application might wordwrap the text to make it look like this, if using a window 40 characters wide:

```
Hi!

How's it going?  I got the new MsgAPI
kit today!

Anyhow, gotta run!
```

Paragraphs should always be wordwrapped by the application, regardless of the screen/window size. When parsing the message text, linefeeds and soft carriage returns should be simply skipped.

The 'message control information' has a somewhat more restricted format. The control information is passed to the application in the form of an ASCIIZ string. The control information is a variable-length string of text which contains information not found in the (fixed-size) message header.

The format of control information is given by the following regular expression:

(group)+<NUL>

A 'group' consists of a <SOH> and a control item.

<SOH> is the Start Of Header character, or ASCII 01. All control information strings must begin with an SOH, whether or not control items are present.

Following the <SOH> is a control item. A control item consists of a string which describes the type of control item, or it may consist of nothing.

At least one group must be present in each message. If a message has no extra control information, this field should consists of a <SOH> followed by a single <NUL>.

Although the control items are free-form, the following format is suggested:

<SOH>tag: value

where 'tag' is a descriptive identifier, describing the type of field that the item represents. 'value' is simply free-form text, which continues up until the next SOH or <NUL>.

The character set for the tag and value consists of those characters in the range 2-255, inclusive.

As an example, a message might have the following control information:

<SOH>CHARSET: LATIN1<SOH>REALNAME: Mark Twain<NUL>

The trailing <NUL> byte must be included in the read count given by 'cbyt'.

The return value for this function is the number of bytes read from the message body. If no characters were requested, this function returns 0.

On error, the function returns -1 and sets msgapierr to one of the following:

MERR_BADH
MERR_BADF
MERR_NOMEM

MsgWriteMsg

```
sword MsgWriteMsg(HMSG hmsg, word fAppend, PXMSG msg,
                  byte *text, dword textlen, dword totlen,
                  dword clen, byte *ctxt);
```

The MsgWriteMsg function is used to write the message header, body, and control information to a message.

'hmsg' is a message handle, as returned by the MsgOpenMsg function. The message must have been opened with a mode of MOPEN_CREATE, MOPEN_WRITE or MOPEN_RW.

'fAppend' is a boolean flag, indicating the state of the message body. If 'append' is zero, then the API will write the message body starting at offset zero. Otherwise, if 'append' is non-zero, the API will continue writing from the offset used by the last MsgWriteMsg call. This flag applies to the message body only; if no text is to be written to the body, this argument should be set to 0.

'msg' is a pointer to an XMSG structure. If this pointer is

non-NULL, then MsgWriteMsg will place the XMSG structure information into the message's physical header. To leave the header unmodified, NULL should be passed for 'msg'.

THIS 'msg' PARAMETER MUST BE PASSED THE **FIRST** TIME THAT MSGWRITEMSG() IS USED WITH A JUST-OPENED MESSAGE HANDLE!

'text' points to an array of bytes to be written to the message body. If no text is to be written, this argument should be NULL.

'textlen' indicates the number of bytes to be written to the message body in this pass of the MsgWriteMsg function. The text is free-format, and it can consist of any characters, including NULs and control characters. If the application does not wish to update the message body, a value of 0L should be passed for this argument.

'totlen' indicates the total length of the message to be written. This differs from 'textlen' in that the message may be written a piece at a time (using small 'textlen' values), but the total length of the message will not exceed 'totlen'. This parameter can be somewhat restrictive for the application; however, this value is required for optimal use of some message base types. The 'totlen' value does not have to be the exact length of the message to write; however, space may be wasted if this value is not reasonably close to the actual length of the message. The rationale behind this argument is that it gives the API writer the most flexibility, in terms of supporting future message base formats. If the application can provide this information to the API, then almost any message base format can be supported by simply dropping in a new API module or DLL.

To write text by making multiple passes, the FIRST pass should call MsgWriteMsg with 'append' set to 0, with the total length of the message in 'totlen', and the length of 'text' in textlen. Second and subsequent passes should set 'append' to 1, with the length of 'text' in textlen.

If the application does not wish to update the message body of an existing message, a value of 0L should be passed for this argument.

This argument MUST be specified during the first call to the MsgWriteMsg when using a mode of MOPEN_CREATE, even if the first call is not requesting any text to be written. However, this value will be stored internally, and ignored on the second and later calls.

When operating on a preexisting message (opened with MOPEN_WRITE or MOPEN_RW), it is an error to specify a length

in 'totlen' which is greater than the original length of the message.

'clen' specifies the total length of the control information, including the trailing NUL byte. To write no control information, a value of 0L should be passed for this argument.

'ctxt' is a pointer to the control information string. To write no control information, a value of 0L should be passed for this argument.

N.B. Several restrictions apply to writing control information:

First and foremost, control information can only be written once. If the control information is to be changed, the message must be read and copied to another message.

Secondly, control information MUST be written during or before MsgWriteMsg is called with information about the message body.

MsgWriteMsg returns a value of 0 on success, or -1 on error. If an error occurred, msgapierr will be set to one of the following values:

MERR_BADH
MERR_BADF
MERR_NOMEM
MERR_NODS

MsgKillMsg

sword MsgKillMsg(HAREA ha, dword msgnum);

The MsgKillMsg function is used to delete a message from the specified message area.

'ha' is a message area handle, as returned by MsgOpenArea.

'msgnum' specifies the message number to kill.

It is an error to kill a message which is currently open.

MsgKillMsg returns a value of 0 if the message was successfully killed, or it returns -1 on error and sets msgapierr to one of the following:

MERR_BADH
MERR_NOENT

MERR_BADF
MERR_NOMEM

MsgGetCurPos

dword MsgGetCurPos(HMSG hmsg);

The MsgGetCurPos function retrieves the 'current position' of a message handle. This position is where the MsgReadMsg would read text from the message body next.

'hmsg' is a message handle, as returned by MsgOpenMsg.

MsgGetCurPos returns the offset into the message on success, or (dword)-1 on error and sets msgapierr to:

MERR_BADH

MsgSetCurPos

sword MsgSetCurPos(HMSG hmsg, dword pos);

The MsgSetCurPos function sets the 'current position' in a message handle. This position is used by MsgReadMsg to read text from the message body.

'hmsg' is a message handle, as returned by MsgOpenMsg.

'pos' is the number of bytes into the message from which MsgReadMsg should start reading.

MsgSetCurPos returns 0 on success, or -1 on error and sets msgapierr to:

MERR_BADH

MsgGetTextLen

dword MsgGetTextLen(HMSG hmsg);

The MsgGetTextLen function retrieves the length of the message body for the specified message.

'hmsg' is a message handle, as returned by MsgOpenMsg.

MsgGetTextLen returns the length of the body on success. On error, it returns (dword)-1 and sets msgapierr to:

MERR_BADH

MsgGetCtrlLen

dword MsgGetCtrlLen(HMSG hmsg);

The MsgGetCtrlLen function retrieves the length of the control information for the specified message.

'hmsg' is a message handle, as returned by MsgOpenMsg.

MsgGetCtrlLen returns the length of the control information on success. On error, it returns (dword)-1 and sets msgapierr to:

MERR_BADH

MsgInvalidHmsg

sword MsgInvalidHmsg(HMSG hmsg);

The MsgInvalidHmsg function tests the given message handle for validity. If the message area handle is invalid, this function returns TRUE and sets msgapierr to MERR_BADH. Otherwise, a value of FALSE will be returned. returned.

UMSGID Translation Functions

MsgMsgnToUid

UMSGID MsgMsgnToUid(HAREA ha, dword msgnum);

The MsgMsgnToUid function converts a message number to a 'unique message ID', or UMSGID. This function can be used to maintain pointers to an 'absolute' message number, regardless of whether or not the area is renumbered or packed. The MsgMsgnToUid function converts a message number to a UMSGID, and the MsgUidToMsgn function converts that UMSGID back to a message number.

'mh' is the message area handle, as returned by MsgOpenArea.

'msgnum' is the message number to convert.

MsgMsgnToUid returns a UMSGID on success; otherwise, it returns 0 and sets msgapierr to:

MERR_BADH
MERR_BADF
MERR_NOENT

MsgUidToMsgn

dword MsgUidToMsgn(HAREA ha, UMSGID umsgid, word type);

The MsgUidToMsgn function converts a UMSGID to a message number.

'ha' is the message area handle, as returned by MsgOpenArea.

'umsgid' is the UMSGID, as returned by a prior call to MsgMsgnToUid.

'type' is the type of conversion to perform. 'type' can be any of the following values:

UID_EXACT	Return the message number represented by the UMSGID, or 0 if the message no longer exists.
UID_PREV	Return the message number represented by the UMSGID. If the message no longer exists, the number of the preceding message will be returned.
UID_NEXT	Return the message number represented by the UMSGID. If the message no longer exists, the number of the following message will be returned.

If no valid message could be found, MsgUidToMsgn returns 0 and sets msgapierr to one of the following:

MERR_BADH
MERR_NOENT

CtrlInfo Manipulation Functions

MsgGetCtrlToken

byte *MsgGetCtrlToken(char *szCtrlInfo, char *szTag);

The MsgGetCtrlToken function finds a specified control information tag in a message's control information buffer.

'szCtrlInfo' is the control information buffer for a message, using the format described for MsgReadMsg.

'szTag' is the name of the tag to be extracted from the control information buffer. (For example, tags such as "FMPT ", "MSGID:" or "REPLY:" can be specified.)

If the tag is found in szCtrlInfo, the function allocates a new block of memory and copies the tag to that new block. The address of the new block is returned to the caller. (The returned address should be freed later with a call to MsgFreeCtrlToken.)

If the tag is not found in szCtrlInfo, this function returns NULL.

For example, given a szCtrlInfo buffer that initially contains the following text:

<SOH>REALNAME: John Doe<SOH>FMPT 24<SOH>TOPT 6<SOH>

A call to MsgGetCtrlToken(szBuf, "FMPT ") would return the string "FMPT 24". After the call, szCtrlInfo would contain:

<SOH>REALNAME: John Doe<SOH>TOPT 6<SOH>

MsgFreeCtrlToken

```
void MsgFreeCtrlToken(byte *szCtrlToken);
```

The MsgFreeCtrlToken function frees a control token that was returned by a call to MsgGetCtrlToken. This function must be called to release memory that is allocated by the MsgGetCtrlToken function.

'szCtrlToken' is the return value of a prior call to MsgGetCtrlToken.

MsgRemoveToken

```
void MsgRemoveToken(byte *szCtrlInfo, byte *szTag);
```

The MsgRemoveToken function removes a specified control token from a control information buffer. The existing text in the buffer is shifted over such that no gaps are left in the buffer.

'szCtrlInfo' is a control information buffer, using the format specified in the description for MsgReadMsg.

MsgGetNumKludges

```
word MsgGetNumKludges(byte *szCtrlInfo);
```

The MsgGetNumKludges function returns the number of tags contained in the control information buffer.

'szCtrlInfo' is a control information buffer, using the format specified in the description for MsgReadMsg.

For example, given a control information buffer of the following format:

<SOH>TAG1: ABC<SOH>TAG2: DEF<SOH>TAG3: GHI<SOH>

MsgGetNumKludges would return a value of 3, since three separate tags exist within the control information buffer.

MsgCvt4D

```
void MsgCvt4D(byte *szCtrlInfo, NETADDR *orig,
              NETADDR *dest);
```

The MsgCvt4D function converts FidoNet-style control information (namely, the FMPT, TOPT and INTL tags) into a proper 4D address. Messages are also un-gaterouted, if necessary. After each tag is converted, it is removed from the control information.

Most normal applications will not need to use this function. MsgCvt4D is usually only used by tosser/scanner programs. (The MsgAPI automatically handles 4D address conversions when writing to *.MSG areas.)

'szCtrlInfo' specifies the buffer containing the original control information for the message.

'orig' points to a NETADDR structure containing the two-dimensional origination address of the message. (This information is normally extracted from the 2D header of a FTS-0001 packet.)

'dest' points to a NETADDR structure containing the two-dimensional destination address of the message. (This information is normally extracted from the 2D header of a FTS-0001 packet.)

On output, the 'orig' and 'dest' parameters are updated with zone and point information from the control information, if necessary.

For example, if the following parameters were given as input:

```
orig = {0, 249, 106, 0}  /* zone and point fields are 0 */
dest = {0, 254, 1, 0}    /* zone and point fields are 0 */

szCtrlInfo = <SOH>FMPT 4<SOH>TOPT 6<SOH>INTL: 2:254/1
```

1:249/106<SOH>REALNAME: John Doe<SOH>

A call to MsgCvt4D would result in the parameters being updated as follows:

orig = {1, 249, 106, 4}
dest = {2, 254, 1, 6}
szCtrlInfo = <SOH>REALNAME: John Doe<SOH>

MsgCreateCtrlBuf

```
char *MsgCreateCtrlBuf(char *szKludgeText,
                       char **pszEndText,
                       unsigned *puiLength);
```

The MsgCreateCtrlBuf function is used to convert an array of FidoNet-style "kludge lines" (as per FTS-0001) into a control information buffer. Most application programs will not need to use this function.

MsgCreateCtrlBuf is normally used by tosser/scanner programs that need to directly manipulate FTS-0001 *.PKT files. Most applications will not need to use this function.

'szKludgeText' points to the buffer containing the beginning of an FTS-0001 style message body. MsgCreateCtrlBuf will extract all of the control information from this buffer, up until the first non-blank and non-kludge line is encountered.

'pszEndText' should be a pointer to a 'char *' variable. When MsgCreateCtrlBuf returns, the given variable will point to the first byte of the text following the kludge information. (In most cases, this will be the beginning of the FTS-0001 style message body.)

'puiLength' is a pointer to a variable that contains the length of the message buffer pointed to by 'szKludgeText'. On return, this variable will be updated to reflect the real length of the message body, not counting the kludge lines at the beginning of the text.

This function returns a pointer to the newly-created control information buffer. This buffer should be freed with a call to MsgFreeCtrlBuf.

MsgCvtCtrlToKludge

```
char *MsgCvtCtrlToKludge(char *szCtrlInfo);
```

The MsgCvtCtrlToKludge function converts a body of control information lines to an equivalent set of FTS-0001 kludge lines.

MsgCvtCtrlToKludge is normally used by tosser/scanner programs that need to directly manipulate FTS-0001 *.PKT files. Most applications will not need to use this function.

'szCtrlInfo' is a buffer of control information, using the format given in the MsgReadMsg description.

This function returns a pointer to a new block of memory containing the FTS-0001 style kludge lines. This buffer should be freed with a call to MsgFreeCtrlBuf.

MsgFreeCtrlBuf

```
void MsgFreeCtrlBuf(byte *szCtrlBuf);
```

The MsgFreeCtrlBuf function is used to free the control buffers which are allocated by the MsgCvtCtrlToKludge and MsgCreateCtrlBuf functions.

'szCtrlBuf' is the return value of a prior call to MsgCvtCtrlToKludge or MsgCreateCtrlBuf.

Squish-Specific Functions

SquishHash

```
dword SquishHash(byte *szTxt);
```

The SquishHash function is used to calculate a hash for a "To:" field in a message.

'szTxt' is a pointer to the To: field of a message.

The algorithm used to calculate this hash is described in the "Index File Format" subsection of the Squish File Format Specification (contained later in this document).

SquishSetMaxMsg

```
dword SquishSetMaxMsg(HAREA ha, dword dwMaxMsg,
                      dword dwSkipMsg, dword dwMaxDays);
```

The SquishSetMaxMsg function is used to set the "maximum messages", "skip messages" and "maximum age" parameters in a Squish-format base.

'ha' is a Squish area handle, as returned by MsgOpenMsg. 'ha' must be a handle for a Squish-format message area.

'dwMaxMsg' is the maximum number of messages to allow in this area.

'dwSkipMsg' is the number of messages to skip (at the beginning of the base) before automatically deleting messages.

'dwMaxDays' is the maximum age (in days) of messages to be kept in the message base.

API Changes from MsgAPI Version 0

1) Renamed the msg.ftsc_date field to msg.__ftsc_date. Most of the developers whoa re currently using this field should actually be using the binary dates. See the comments for the __ftsc_date field in MSGAPI.H for more information.

2) For compatibility with Windows, the MSG and MSGH names have been changed. The old-style ones will still be there for compatibility, but the API defaults to using HAREA instead of MSG*, and HMSG instead of MSGH*. If you wish to turn off the old definitions because they conflict with your source files or the Windows headers, define MSGAPI_NO_OLD_TYPES before including msgapi.h.

3) Renamed the InvalidMsgh and InvalidMsg functions to MsgInvalidHarea and MsgInvalidHmsg, making them more consistent with the other API names. Also fixed the documented return codes for these functions.

4) A few slight changes were made to the XMSG structure. Notably, now only 9 reply links are supported, and the field that used to hold the tenth reply link is now used for storing a message's UMSGID.

BUILDING MSGAPI APPLICATIONS

Source File Requirements

Any code which uses the MsgAPI must include the following line at the top of each source file:

```
#include "msgapi.h"
```

If the MsgAPI source is not installed in the current directory, you should add the MsgAPI directory to your compiler's "include path" before trying to compile an application.

Programmers who are using MsgAPI with Windows should make sure to define the "MSGAPI_NO_OLD_TYPES" constant before including the msgapi.h header. Older versions of the MsgAPI used typedefs which were incompatible with Windows, and this macro must be defined to stop MsgAPI from using the old typedef names.

For information on using the MsgAPI function calls, please see the section entitled "MsgAPI Function Reference". This section contains a complete list of function calls in the API, including formal parameters, notes on operation, and return codes.

If you are using TC++ or BC++, make sure that your application is compiled in "C mode". MsgAPI was not designed for use with C++. (In the TC++ IDE, set Options/Compiler/C++ to "CPP extension only". For the command-line compiler, make sure that you are NOT using the "-P" switch.)

Also, the distribution TC++/BC++ and WC libraries have been compiled without overlay support. If you wish to use the WATCOM or Borland overlay managers, you will have to recompile the MsgAPI source with overlay support.

Compiling Source Files

No special requirements are necessary for compiling source files for the DOS version of MsgAPI. However, make sure to select the LARGE memory model when compiling your application. If you wish to create an application using a different memory model, you will have to recompile the MsgAPI code.

For compiling OS/2 applications that use MsgAPI, make sure that the "OS_2" macro is defined on the compiler command-line. The MsgAPI uses a different set of calling conventions under OS/2, so all of your code must be compiled with the OS_2 macro defined.

Linking Your Application

This package already includes pre-made libraries for a number of compilers, all built using the large memory model.

Under DOS, the MsgAPI library is statically linked with each application. To link with the MsgAPI routines, simply specify the required library file when linking. For example:

TLINK (Borland C, DOS):

```
tlink \bc\lib\c0l myapp,myapp,nul,bc_dos_l \bc\lib\cl
```

TLINK (Turbo C, DOS):

```
tlink \tc\lib\c0l myapp,myapp,nul,tc_dos_l \tc\lib\cl
```

WLINK (16-bit DOS):

```
wlink file myapp lib wc_dos_l name myapp
```

WLINK (32-bit DOS):

```
wlink file myapp lib wc_dos_f name myapp
```

LINK (DOS):

```
link myapp,myapp,nul,mc_dos_l;
```

If using the TC++ or BC++ IDE, simply declare the required library as part of your project file. If you are using TC++ or BC++ from the command line, you can specify the name of the library after your source module, like this:

```
tcc -ml myapp.c tc_dos_l.lib
```

Under OS/2, the MsgAPI code is distributed as a DLL (dynamic link library). To use this code in your application, simply link with the DLL16.LIB import library (for 16-bit code) or the DLL32.LIB import library (for 32-bit code). 16-bit OS/2 applications should also be compiled in the large memory model.

After ensuring that MSGAPI.DLL and/or MSGAPI32.DLL are on your LIBPATH, your application should be ready to run.

If you wish to use the default libraries and memory models, you do not need any of the *.c files in the source directory. However, you must still retain all *.h header files, since they are required when compiling your application.

Building the Sample Applications

The SAMPLES subdirectory contains a number of sample programs that use the MsgAPI. The commands given below can be used to make the samples using the supported compilers:

Compiler	OS	Command
WATCOM C/16	DOS	wmake -u -f makefile.wcd
WATCOM C/16	OS/2	wmake -u -f makefile.wco
WATCOM C/32	DOS	wmake -u -f makefile.w3d
WATCOM C/32	OS/2	wmake -u -f makefile.w3o
Microsoft C	DOS	nmake -f makefile.mcd
Microsoft C	OS/2	nmake -f makefile.mco
Turbo C	DOS	make -fmakefile.tcd
Borland C	DOS	make -fmakefile.bcd

BUILDING THE MSGAPI SOURCE

The MsgAPI sources have been tested with the following compilers:

WATCOM C/16 9.5	(DOS and OS/2)
WATCOM C/32 9.5	(DOS and OS/2)
Microsoft C 6.0	(DOS and OS/2)
Turbo C 2.0	(DOS only)
Borland C++ 3.1	(DOS only)

In addition, either the Microsoft Macro Assembler or the Borland Turbo Assembler is required to build the MsgAPI source.

Other versions of these compilers (or compilers made by different vendors) may also work, but only the compilers above have been tested. The code works with both 16-bit and 32-bit compilers. (However, if you add support for a compiler from a different vendor, you will probably have to modify h\compiler.h.)

To recompile the MsgAPI source, simply edit the MAKEFILE within the SRC directory. Sample definitions have been provided for all of the supported compilers.

The makefile included in this package should work with almost any make utility. This makefile has been successfully used under Borland's MAKE, Microsoft's NMAKE, WATCOM's WMAKE, and Vadura's DMAKE.

Warning! WMAKE users must use the "-u" command-line option when running WMAKE.EXE. By default, WMAKE uses a non-standard line continuation character, so the makefile will not be processed correctly unless the "-u" option is used.

By default, the makefile is set up to compile for WATCOM C/16 9.5. To select another compiler, comment out all of the WC/16 definitions and uncomment those that pertain to your compiler.

Having made the appropriate modifications, simply type "make" to rebuild the appropriate msgapi*.lib or msgapi*.dll file.

Under DOS, the "MODEL=" directive at the beginning of the makefile can also be used to instruct the compiler to build a MsgAPI library for one of the following memory models. MODEL should be set to one of the following values:

s	Small
m	Medium
c	Compact
l	Large

The files in the FEATURES subdirectory contains everything that is required to write third-party extensions for Squish.

This developers kit includes the source for two sample feature DLLs. These DLLs are not part of the standard Squish distribution, nor are they supported in any manner. They are simply included to provide examples and to give a demonstration of feature DLL programming.

Sample Feature DLLs

KILLRCAT.DLL

The first sample DLL is "KillrCat", a message filtering program that selectively deletes messages based on a set of search criteria. The following can be added to the top of your SQUISH.CFG for a demonstration of KillrCat's capabilities:

```
; The Feature and/or Feature32 keywords tell Squish to load
; KILLRCAT and/or KILLRC32.

Feature   KillrCat
Feature32 KillrC32

; Keyword:
;
; KillrCat      <where> <echo_tag>      <text>
;
; <where>:      t = to
;               f = from
;               s = subject
;               b = body
;
; Any combination of the above flags are permitted.
;
; <echo_tag> is the tag of the echo area.
;
; <text> is the text to find.

KillrCat  b    sysop249       Node Diff announcement
KillrCat  s    sysop249       Nodediff announcement
KillrCat  sb   users249       Latest Versions'n Stuff
KillrCat  sb   users249       Fidonews!
KillrCat  b    netmail        File request generated by AMAX
```

When KillrCat find a message that matches the specified criteria, it diverts the message from its usual area and places it in BAD_MSGS. If you don't want to see these messages at all, fix KILLRCAT.C and use the "delete message" bitmask in the return code.

MSGTRACK.DLL

The other sample DLL is a message bouncer. This bouncer requires a version 7 nodelist to work properly. Try adding the following to your SQUISH.CFG for a demonstration:

```
Feature   Msgtrack
Feature32 Msgtra32

; Path and name of nodelist

Msgtrack Nodelist   <filename>

; Delete bounced messages
;Msgtrack Kill
```

When packing netmail, this bouncer will automatically return messages to the originating node if the destination is not listed in the nodelist.

<filename> should be the name of the NODEX.DAT file from your version 7 nodelist.

The optional "Msgtrack Kill" line instructs the message bouncer to delete messages after they have been bounced. (By default, Squish just marks a bounced message as "Sent" and "Orphan", but the message is left in the netmail area.)

Feature DLL Programming Interface

FEATURES.LZH includes a sample TEMPLATE.C file that can be used as a template for writing feature DLLs.

A feature DLL must start with the following lines:

```
#define INCL_DOS
#include <os2.h>
#include "sqfeat.h"
```

os2.h is the standard OS/2 include file, as distributed in the OS/2 toolkit. (For 16-bit feature DLLs, the OS/2 1.x toolkit is required. For 32-bit feature DLLs, an OS/2 2.x toolkit is required.)

sqfeat.h is the feature DLL include file. This contains definitions and typedefs necessary to write a Squish feature DLL.

Each feature DLL must contain a number of functions. These functions are used as hooks when Squish is performing certain actions.

All of the hook functions must be exported by name, and they all must use the pascal calling convention. The "FEATENTRY" keyword automatically defines all of these attributes.)

The following functions must be present in every feature DLL:

FeatureInit

This function is responsible for performing feature-specific initializations. FeatureInit() is called as soon as the Feature or Feature32 keyword is encountered in SQUISH.CFG.

This function is passed a pointer to a _feat_init structure. The structure contains the following fields:

struct_len — The length of the _feat_init structure

szConfigName — The name of the DLL. This should be filled in by the FeatureInit() function. This name is be used for adding feature-specific configuration options to SQUISH.CFG.

When Squish encounters an unknown configuration line, it first checks to see if the first word matches any of the feature DLLs, as given in the szConfigName field. If a match is found, the configuration line is passed to the DLL's FeatConfig function.

pfnLogMsg — A pointer to the Squish logging routine. This field is filled in by Squish itself. If the feature DLL needs to add lines to the Squish log, this function pointer can be used to access the system log function.

The FeatInit() function should save a copy of the pfnLogMsg field in a global variable, such that the other Feat...() functions can create log entries when necessary.

ulFlag — This flag contains a bitmask that Squish uses to optimize calls made to the feature DLL. FeatInit() should initialize this field before returning to the caller.

Any or all of the following flags can be combined using the bitwise OR operator ("|"):

FFLAG_NETSENT — Only call FeatNetMsg when packing a message that does not have the SENT bit set.

FFLAG_NETTOUS — Only call FeatNetMsg when packing a message which is addressed to one of our addresses, as given in SQUISH.CFG.

FFLAG_NETRECD — Only call FeatNetMsg when packing a message that does not have the RECEIVED bit set.

FFLAG_NETNOTTOUS — Only call FeatNetMsg when packing a message that is NOT addressed to one of our addresses, as given in SQUISH.CFG.

If this field is given a value of 0, the FeatNetMsg function will be called before packing every netmail message.

FeatureConfig

This function is called whenever a feature-specific configuration line is encountered. Feature DLLs can use this function to process application-specific configuration information from SQUISH.CFG.

This function is passed a pointer to a _feat_config structure. The structure contains the following fields:

struct_len — This field contains the length of the _feat_config structure.

szConfigLine — This field contains the line just read from SQUISH.CFG. The first word on this line matches the szConfigName value that was placed in the _feat_init structure by FeatInit.

ppszArgs — This field contains a standard argv-style variable, pointing to a tokenized version of szConfigLine. The first word on szConfigLine can be accessed as ppszArgs[0]; the second word can be accessed as ppszArgs[1]; and so on. If there are 'n' arguments, ppszArgs[0] through ppszArgs[n-1] will contain pointers to each word on the line, and ppszArgs[n] will be NULL.

The FeatureConfig function should extract any relevant

information and store it in a global variable that can be accessed by other functions in the DLL.

FeatureNetMsg

This function is called whenever a NetMail message is about to be packed. This function is only used on non-ArcmailAttach systems.

This function is passed a pointer to the _feat_netmsg structure. The structure contains the following fields:

struct_len — This field contains the length of the _feat_netmsg structure.

ulAction — This field controls the action taken by Squish when FeatureNetMsg returns. The contents of this field should be filled out by the feature DLL.

This field can be set to a combination of any of the following values, using the bitwise OR operator ("|"):

FACT_NONE — No action is taken

FACT_KILL — The message is deleted upon returning to Squish, before the message is packed.

FACT_SKIP — The message is left alone and is not packed.

FACT_HIDE — Do not pass this message to other installed features. (By default, all installed features are called with a pointer to the current message. However, if the FACT_HIDE flag is set, other DLLs will not be called with this message during the current run.)

FACT_RWMSG — Rewrite the current message. This flag should be used if the DLL modifies the message header in pMsg.

pszAreaTag — This field contains the name of the current

netmail area.

ha — This is a MsgAPI HAREA handle for the current netmail area.

hmsg — This is a MsgAPI HMSG handle for the current message.

ulMsgNum — This contains the message number of the current message.

pMsg — Pointer to the message header of the current message.

pszCtrl — Pointer to the control information for the current message.

pszMsgTxt — Pointer to the message text for the current message.

us — A NETADDR-type structure giving Squish's primary address.

FeatureTossMsg

This function is called whenever an EchoMail message is about to be tossed.

This function is passed a pointer to the _feat_toss structure. The structure contains the following fields:

struct_len — Length of the _feat_toss structure.

ulTossAction — This field specifies the action to be taken when Squish returns from FeatureTossMsg.

This field can contain any of the following options, combined using the bitwise OR operator ("|"):

FTACT_NONE — No action is taken.

FTACT_KILL — Delete the message without attempting to toss it.

FTACT_AREA — Toss to the new area tag specified in szArea.

FTACT_HIDE — Do not pass this message to any other installed features. (See the comments for

FACT_HIDE in FeatureNetMsg for more information.)

FTACT_NSCN — Do not scan this message to anyone else (if performing a one-pass toss/scan).

szArea — Area tag for this message. This field can be changed to cause the message to be redirected to another area, but to ensure that Squish recognizes the change, the FTACT_AREA flag should be set (above).

szPktName — Name of the packet file that contains the current message.

pMsg — Pointer to the message header for the current message.

pszCtrl — Control information for the current message.

pszMsgTxt — Message text of the current message.

FeatureScanMsg

This function is currently not supported by Squish. However, it must be present for Squish to load the DLL.

FeatureTerm

This function is called when Squish is about to finish execution. This function should perform feature-specific clean-up operations.

Feature16Bit

This function is never actually called by Squish, but it must be present in all 16-bit feature DLLs.

Feature32Bit

This function is never actually called by Squish, but it must be present in all 32-bit feature DLLs.

Building Feature DLLs

The following commands can be used to build a sample 16-bit DLL using WATCOM C/16 9.5 or above, where "template" is the name of the source file for the feature:

```
[C:\] wcc /bd /dOS_2 /ml /zu template.c

[C:\] wlink sys os2 dll initi debug all name template.dll
file template.obj lib os2 opt manyauto, protmode, verb,
modname=template
```

The following commands can be used to build a sample 32-bit DLL using WATCOM C/32 9.5 or above:

```
[C:\] wcc386 /bd /dOS_2 /fo=templa32 template.c

[C:\] wlink sys os2v2 dll initi debug all name templa32.dll
file templa32.obj lib os2386 opt verb, manyauto,
modname=templa32
```

SQUISH FILE FORMAT SPECIFICATION

This section describes the physical file layout of a Squish message base. This is intended as a reference for developers who are writing their own Squish-compatible programs. For an overview of the Squish message base, see the section of SQUISH.PRN entitled "Using Squish-Format Message Areas".

While the Squish MsgAPI library provides a standardized interface to Squish and *.MSG bases for C programmers, authors who use other languages may need to access Squish bases directly. This section describes the implementation details of the Squish file format.

Squish Philosophy

A standard Squish base consists of two files: a message data file and a message index file. Both files have the same prefix, but the data file has an extension of ".sqd", while the index file has an extension of ".sqi".

From an overall point of view, the Squish data file consists of messages stored in a doubly-linked list. The Squish data file includes a header that contains pointers to the first and last frames in the area, in addition to other area-specific information.

In the data file, a "frame" is used to hold an individual message. A frame consists of a frame header (which contains links to the prior and next messages), followed by the optional message header, control information and message body fields.

This "linked list of frames" approach is ideal for a BBS message base. Almost all message base access is sequential, starting from a particular offset, and reading or writing until the end of the message base is reached. Since each frame header contains the offset of the prior and next messages, no disk accesses are required to find the preceding or following messages.

The index file is a flat array of Squish Index (SQIDX) records. The index file is used primarily for performing random access look-ups by message number.

Unlike other message base formats, the Squish base is only loosely based on the concept of "message numbers". While all messages have a message number, these numbers can change at any time. By definition, the message numbers in a Squish base always range from 1 to the highest message in the area. Consequently, there are no "gaps" in message numbers, so a Squish message area never needs to be renumbered.

While this makes it easy to scan through all of the messages in an area, this also makes it difficult to find one specific message. Consequently, the concept of a "unique message identifier" or (UMSGID) is introduced.

When a message is created, it is assigned a 32-bit UMSGID. These identifiers are unique and NEVER CHANGE. Unique message numbers are never "renumbered", so once a UMSGID of a message is obtained, it can always be used to find the current message number of the given message, no matter how many messages have been added or deleted in the interim.

Squish Data Types

The following integral types are used in the Squish file format definitions:

Type	Size	Description
char	1 byte	A one-byte unsigned character.
word	2 bytes	A two-byte unsigned integer.
sword	2 bytes	A two-byte signed integer.
dword	4 bytes	A four-byte unsigned integer.
FOFS	4 bytes	A four-byte unsigned integer. This type is used to store offsets of frames within the Squish data file.
UMSGID	4 bytes	A four-byte unsigned integer. This type is used to store unique message identifiers for Squish messages.

The types above are stored in the standard Intel "backwords" format, with the least significant byte being stored first, and the most significant byte being stored last.

A two-byte integer containing 0x1234 would be stored as follows:

Offset	Value
0	0x34
1	0x12

A four-byte integer containing 0x12345678 would be stored as follows:

Offset	Value
0	0x78
1	0x56
2	0x34

3 0x12

The Squish file format also uses a number of abstract data types:

The SCOMBO type is used for describing a message date/time stamp. This structure has the following format:

Name	Type	Ofs	Description
date	word	0	DOS bitmapped date value. This field is used to store a message date. The first five bits represent the day of the month. (A value of 1 represents the first of the month.) The next four bits indicate the month of the year. (1=January; 12=December.) The remaining seven bits indicate the year (relative to 1980).
time	word	2	DOS bitmapped time value. This field used to store a message time. The first five bits indicate the seconds value, divided by two. This implies that all message dates and times get rounded to a multiple of two seconds. (0 seconds = 0; 16 seconds = 8; 58 seconds = 29.) The next six bits represent the minutes value. The remaining five bits represent the hour value, using a 24-hour clock.
	Total:	4 bytes	

The NETADDR type is used for describing a FidoNet network address. This structure has the following format:

Name	Type	Offset	Description
zone	word	0	FidoNet zone number.
net	word	2	FidoNet net number.
node	word	4	FidoNet node number.
point	word	6	FidoNet point number. If the system is not a point, this field should be assigned a value of zero.
	Total:	8 bytes	

In addition, to describe an array of a given type, the "type[n]" notation is used. For example, "char[6]" represents an array of six contiguous characters. Likewise, "UMSGID[12]" represents an array of twelve UMSGID types.

Data File Format

The Squish data file consists of two major sections:

1) A fixed-length area header, stored at the beginning of the file.

2) A variable-length heap that comprises the rest of the file. This part of the file is used for storing message text.

The area header stores pointers to the head and tail of two major "chains" of messages; the message chain and the free chain. The message chain is used to find all active messages in an area. The free chain is used for storing the locations of deleted messages, such that space can be reused at a later point in time.

The Squish data file always contains a copy of the following _sqbase structure at file offset 0:

Name	Type	Offset	Description
len	word	0	Length of the _sqbase structure.
reserved	word	2	Reserved for future use.
num_msg	dword	4	Number of messages in this Squish base. This should always be equal to the value of the high_msg field.

high_msg	dword	8	Highest message number in this Squish base. This should always be equal to the value of the num_msg field.
skip_msg	dword	12	When automatically deleting messages, this field indicates that the first skip_msg messages in the area should not be deleted. (If max_msg=50 and skip_msg=2, this means that the writing program should start deleting from the third message whenever the total message count exceeds 50 messages.)
high_water	dword	16	The high water marker for this area, stored as a UMSGID. This field is used in EchoMail areas only. This contains the UMSGID of the highest message that was scanned by EchoMail processing software.
uid	dword	20	This field contains the UMSGID to be assigned to the next message created in this area.
base	char[80]	24	Name and path of the Squish base, as an ASCIIZ string, not including the extension. This field is optional and will not necessarily be filled out by all applications. (If this field is not supported, it should be initialized to ASCII 0.)
begin_frame	FOFS	104	Offset of the first frame in the message chain.
last_frame	FOFS	108	Offset of the last frame in the message chain.

free_frame	FOFS	112	Offset of the first frame in the free chain.
last_free_frame	FOFS	116	Offset of the last message in the free chain.
end_frame	FOFS	120	Offset of end-of-file. Applications will append messages to the Squish file from this point.
max_msg	dword	124	Maximum number of messages to store in this area. When writing messages, applications should dynamically delete messages to make sure that no more than max_msgs exist in this area.
keep_days	word	128	Maximum age (in days) of messages in this area. This field is not normally used by applications. However, it is used by SQPACK when performing a message area pack.
sz_sqhdr	word	130	Size of the SQHDR structure. For compatibility with future versions of the Squish file format, applications should use this value as the size of the SQHDR structure, instead of using a hardcoded "sizeof(SQHDR)" value.
reserved	char[124]	132	Reserved for future use.
	Total:	256 bytes	

To examine the messages in a Squish base, the application needs to follow the message chain. To do this, start with the begin_frame and/or end_frame fields. These fields contain the offsets of the first and last frames (respectively) in the message base.

A frame in the message chain consists of a Squish Frame Header structure (SQHDR), followed by the XMSG message header, message control information, and message body.

A frame in the free chain consists of a SQHDR structure only. A free frame does not necessarily contain a message.

A SQHDR structure always has the following format:

Name	Type	Ofs	Description
id	dword	0	The frame identifier signature. This field must always be set to a value of 0xAFAE4453.
next_frame	FOFS	4	Frame offset of the next frame, or 0 if this is the last frame.
prev_frame	FOFS	8	Frame offset of the prior frame, or 0 if this is the first frame.
frame_length	dword	12	Amount of space ALLOCATED for the frame, not including the space used by the SQHDR itself.
msg_length	dword	16	Amount of space USED in this frame, including the size of the XMSG header, the control information, and the message text. This field does NOT include the size of the SQHDR itself.
clen	dword	20	Length of the control information field in this frame.
frame_type	word	24	This field can contain one of the following frame type values: 0 Normal frame. This frame contains an XMSG header, followed by the message control information and the message body. Normal frames should only be encountered when processing the normal message chain. 1 Free frame. This frame has been deleted, but it can be reused. The amount of available space in the frame is given by

			the frame_length field. Free frames should only be encountered when processing the free chain.
			2 LZSS frame. This frame type is reserved for future use.
			3 Frame update. The frame is being updated by another task. This is only a transient frame type; it indicates that the frame should not be manipulated by another task.
			All other frame types are reserved for future use.
reserved	word	26	Reserved for future use.
	Total:	28 bytes	

For a normal frame type, the XMSG header immediately follows the Squish frame header. The XMSG structure has the following format:

Name	Type	Ofs	Description
attr	dword	0	Message attributes. This is a combination of any of the MSG* attributes. (See below.)
from	char[36]	4	Name of the user who originated this message.
to	char[36]	40	Name of the user to whom this message is addressed.
subject	char[72]	76	Message subject.
orig	NETADDR	148	Originating network address of this message.
dest	NETADDR	156	Destination network address of this message. (Used for netmail areas only.)

date_written	SCOMBO	164	Date that the message was written.
date_arrived	SCOMBO	168	Date that the message was placed in this Squish area.
utc_ofs	sword	172	The message writer's offset from UTC, in minutes. Currently, this field is not used.
replyto	UMSGID	174	If this message is a reply, this field gives the UMSGID of the original message. Otherwise, this field is given a value of 0.
replies	UMSGID[9]	178	If any replies for this message are present, this array lists the UMSGIDs of up to nine reply messages.
umsgid	UMSGID	214	The UMSGID of this message. THIS FIELD IS ONLY VALID IF THE MSGUID BIT IS SET IN THE "ATTR" FIELD. Older Squish programs do not always set this field, so its contents can only be trusted if the MSGUID bit is set.
__ftsc_date	char[20]	218	FTS-0001 compatible date. Squish applications should not access this field directly. This field is used exclusively by tossers and scanners for preserving the original ASCII message date. Squish applications should use the binary dates in date_written and date_arrived to retrieve the message date.
	Total:	238 bytes	

Any of the following bitmasks can be used in the XMSG "attr" field:

Attribute	Value	Description
MSGPRIVATE	0x00000001	The message is private.
MSGCRASH	0x00000002	The message is given a "crash" flavour when packed. When both MSGCRASH and MSGHOLD are both enabled, the message is given a "direct" flavour.
MSGREAD	0x00000004	The message has been read by the addressee.

MSGSENT	0x00000008	The message has been packed and prepared for transmission to a remote system.
MSGFILE	0x00000010	The message has a file attached. The filename is given in the "subj" field.
MSGFWD	0x00000020	The message is in-transit; it is not addressed to one of our primary addresses.
MSGORPHAN	0x00000040	The message is orphaned. The message destination address could not be found in the nodelist.
MSGKILL	0x00000080	The message should be deleted from the local message base when it is packed.
MSGLOCAL	0x00000100	The message originated on this system. This flag must be present on all locally-generated netmail for Squish to function properly.
MSGHOLD	0x00000200	The message should be given a "hold" flavour when packed. When combined with the MSGCRASH flag, the message is given a "direct" flavour.
MSGXX2	0x00000400	Reserved for future use.
MSGFRQ	0x00000800	The message is a file request. The filename is given in the "subj" field.
MSGRRQ	0x00001000	A receipt is requested. (Not supported by Squish.)
MSGCPT	0x00002000	This message is a receipt for an earlier MSGRRQ request.
MSGARQ	0x00004000	An audit trail is requested. (Not supported by Squish.)
MSGURQ	0x00008000	This message is an update request. The filename is given in the "subj" field.
MSGSCANNED	0x00010000	This echomail message has been scanned out to other systems.

MSGUID	0x00020000	The "uid" field contains a valid UMSGID for this message.

Index File Format

The index file provides random access capability for a Squish base. Given a message number, the index file can be used to quickly find the frame offset for that message.

Similarly, given a UMSGID, the index file can also be used to find the message number and/or the frame offset for the message.

The Squish index file is an array of Squish Index (SQIDX) structures. Each SQIDX structure corresponds to an active message. For a base containing 'n' messages, there are at least 'n' SQIDX structures. (There may also be extra SQIDX frames at the end of the index file, but these will be initialized with invalid values, as described below.)

The SQIDX for the first message is stored at offset 0.
The SQIDX for the second message is stored at offset 12.
The SQIDX for the third message is stored at offset 24.
(and so on)

The Squish Index structure (SQIDX) has the following format:

Name	Type	Ofs	Description
ofs	FOFS	0	Offset of the frame for this message. A value of 0 is used to indicate an invalid message.
umsgid	UMSGID	4	Unique message ID for this message. A value of 0xffffffff is used to indicate an invalid message. The umsgid field must always be greater than the umsgid field of the preceding SQIDX structure. UMSGIDs are assigned serially, so this will normally be the case. (A binary search is performed on the index file to translate UMSGIDs, so the umsgid field of the SQIDX headers must always be stored in ascending order.)
hash	dword	8	The low 31 bits of this field contain a hash the "To:" field for this message. (See below for the hash function.) The high bit is set to 1 if the MSGREAD flag is enabled in the corresponding XMSG

header.

Total: 12 bytes

The following hash function is used to calculate the "hash" field of the SQIDX structure. All variables are 32-bit unless otherwise noted:

Set "hash" to a value of 0

For each 8-bit character "ch" in the To: field, repeat:

- Shift "hash" left by four bytes.
- Convert "ch" to lowercase
- Increment the hash by the ASCII value of "ch"

- Set "g" to the value of "hash"
- Perform a bitwise AND on "g", using a mask of 0xf0000000.

- If "g" is non-zero:

  - Perform a bitwise OR on "hash" with the value of "g".
  - Shift "g" right by 24 bits.
  - Perform a bitwise OR on "hash" with the value of "g".

Perform a bitwise AND on "hash" with a value of 0x7fffffff.

The following C function can be used to calculate such a hash:

```
#include <ctype.h>

unsigned long SquishHash(unsigned char *f)
{
     unsigned long hash=0;
     unsigned long g;
     char *p;

     for (p=f; *p; p++)
     {
          hash=(hash << 4) + (unsigned long)tolower(*p);

          if ((g=(hash & 0xf0000000L)) != 0L)
          {
               hash |= g >> 24;
               hash |= g;
          }
     }
```

```
        /* Strip off high bit */

        return (hash & 0x7fffffffLu);
    }
```

The SquishHash function is derived from the hashpjw() function by P.J. Weinberger.

The following procedure can be used to find the frame offset or UMSGID of a particular message number:

- Read the Squish data header and ensure that the message number is less than or equal to the value given by "high_msg".

- Subtract one from the message number.

- Multiply the result by 12 (the size of the SQIDX structure).

- The product is the required offset in the SQIDX file. Seek to that offset and read the SQIDX header.

- The "fofs" field of the SQIDX structure indicates the offset of the message frame. The "umsgid" field indicates the UMSGID for the message in question.

To find a message number and/or frame offset of a particular UMSGID, a simple binary search can be performed on the index file. Since the SQIDX structures are (by definition) stored in ascending order by the "umsgid" field. Conventional binary search techniques can be used to find the SQIDX structure for a particular UMSGID value.

Having found the SQIDX structure, the offset of the message can be obtained from the SQIDX "fofs" field. In addition, the message number can be determined from the location of the SQIDX within the index file. (The SQIDX at offset 0 is for message 1; the SQIDX at offset 12 is for message 2; and so on.)

Message Links

Like with any doubly-linked list, inserting a message into a Squish message chain requires more work than just writing a single header.

First of all, when writing the SQHDR for the new message, the "prev_frame" and the "next_frame" fields must be set to the frame offsets of the previous and next messages in the chain.

In addition, the "next_frame" field of the PREVIOUS message must be set to the offset of the message being inserted. Likewise,

the "prev_frame" field of the NEXT message must be set to the offset of the message being inserted.

Similar arguments apply to deleting a message. The prior and next messages must be linked together to remove the current message from the chain.

Beyond that, if the message is being inserted or deleted is at the beginning or end of the chain, the begin_frame/last_frame or free_frame/last_free_frame pointers must be updated (for the message and free chains, respectively).

Reading Messages

Once the offset of a message frame is known, the application can follow these steps to read the message:

- Seek to the specified frame offset.
- Read the SQHDR for the frame, making sure to read "sz_sqhdr" bytes (as given in the _sqdata base header).
- Validate the "id" field. If the contents of this field are not equal to the predefined constant (see "id", above), the Squish base is damaged.
- Validate the "frame_type" field. If the type is equal to "normal frame", proceed. If the type is equal to "frame update", another application is processing the base, so the message should be skipped. Any other value is an error.
- Read the XMSG header.
- Use the "clen" field of the SQHDR to determine the length of the control information. Read this many bytes into an array.
- Use the "msg_length" field of the SQHDR to determine the total length of the frame. Subtract the size of the XMSG header and the "clen" value, and use this result as the length of the message text. Read this many bytes into an array.

To read the next or prior message in a chain, simply use the "next_frame" or "prev_frame" fields (respectively) to obtain the frame offset (FOFS) of the indicated frame. A FOFS of zero indicates that the end of the chain has been reached.

Writing Messages

To write a new message to a Squish base, an application should follow this procedure:

- Obtain exclusive access to the Squish base. (See the section entitled "Concurrency" for more information.)

- Re-read a copy of the Squish base header.

- Scan the messages in the "free_frame" chain to see if any frame is large enough to accommodate the new message. Ensure that the message header has a type of "free frame". If so, unlink the message from the free chain, adjusting the free_frame and last_free_frame pointers if necessary. (See "Message Links", above.)

- If no free frame was found, allocate a frame at the end of the file, using the "end_frame" offset from the base header. Increment the "end_frame" value by the amount of space allocated.

- Link the new SQHDR frame into the end of the message chain, adjusting the begin_frame and last_frame pointers in the base header (if necessary).

- Set the "frame_type" field of the new SQHDR to "frame update" and write at the appropriate offset.

- Copy the "uid" field from the base header into the XMSG header of the message to be written. Also set the MSGUID flag in the "attr" field of the XMSG header. Then increment the "uid" field in the base header.

- Increment the num_msg and high_msg values in the base header.

- Rewrite the Squish base header.

- Relinquish exclusive access to the Squish base.

- Take the offset of the new SQHDR, and add the "sz_sqhdr" value from the base header. Seek to that offset and write the XMSG header.

- Write the message's control information.

- Write the message body.

- Set the frame type of the new message to "normal".

- Seek to the appropriate offset in the Squish index file, and

write a SQIDX header for the new message. Make sure

- Check the "high_msg" count of the message area to ensure that it is less than or equal to max_msg. If not, delete messages (while skipping the first skip_msg messages) until the total message count is less than high_msg.

Deleting Messages

An application should follow this procedure to delete an existing message:

- Obtain the frame offset of the message to be deleted. (If given a message number, the frame offset for a message can be found in the n'th record of the index file.)

- Read the frame header into memory.

- Obtain exclusive access to the message base.

- Update the header pointed to by the next_frame link to skip over the message being deleted.

- Update the header pointed to by the next_frame link to skip over the message being deleted.

- Shift the index file to remove the just-deleted message from the message area. The global pointers in the Squish header (num_msg, high_msg, and so on) should also be updated.

- Append the just-deleted header to the chain of "free" messages. The frame should be inserted at the end of the free list. The procedure used for inserting a free frame is similar to that used in "Writing a message", except that the free list is manipulated (instead of the message list).

- Relinquish exclusive access to the message base.

Concurrency Considerations

For Squish applications to operate properly in a multitasking or networked environment, certain precautions must be taken when manipulating a Squish base. While there are few restrictions on reading message information, a certain set of conventions must be followed when writing to a Squish base.

The procedure for obtaining exclusive access to the message base (as used in the descriptions above) is as follows:

- Attempt to lock the first byte of the Squish data file. If the lock fails, wait for one second and try again. If all

ten locks fail, the base is locked by another process and cannot be modified.

- Re-read the Squish base header. Old copies of the header may no longer be current, so this step must always be performed when obtaining exclusive access to the Squish base.

An application should not write to the Squish base until performing the steps above.

Once exclusive access has been obtained, the application can perform any necessary modifications. When the application no longer needs to write to the message base, the following steps can be performed to relinquish exclusive access:

- Write the updated Squish base header. This revised header should include any modifications which were made while exclusive access was in effect.

- Unlock the first byte of the Squish data file.

As long as all Squish applications follow these conventions for accessing the Squish base, concurrency should not be a problem.

###