MsgAPI Documentation, Revision 0

Function Reference

```
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's accepts one arguments: a pointer to a structure containing information about the application. The contents of this structure as as follows:

```
struct _minf
{
    word req_version;
    word def_zone;
    word _haveshare;
};
```

`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.

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

```
sword MsgCloseApi(void);
```

This function is used to denationalize 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.

```
MSG * 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 ORed 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, a handle is returned in the form of a pointer to a MSG structure. This structure does not contain any information which can be used 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.

`sword MsgCloseArea(MSG *mh);`

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 a MSG* 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.

`MSGH * MsgOpenMsg(MSG *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 last message 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 pointer to a MSGH structure. 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

`sword MsgCloseMsg(MSGH *msgh);`

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

`dword MsgReadMsg(MSGH *msgh, XMSG *msg, 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.

`msgh' 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 below, 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

`sword MsgWriteMsg(MSGH *msgh, word append, XMSG *msg, byte *text, dword textlen, dword totlen, dword clen, byte *ctxt);`

The MsgWriteMsg function is capable of writing the message header, body, and control information to a message.

\`msgh' 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.

\`append' 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 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 make 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

`sword MsgKillMsg(MSG *mh, dword msgnum);`

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

`mh' is a message area pointer, 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

`sword MsgLock(MSG *mh);`

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.

`mh' is a message area pointer, 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

`sword MsgUnlock(MSG *mh);`

The MsgUnlock function unlocks a previously-locked message area.

`mh' is a pointer to a message area, as returned by MsgOpenMsg.

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

MERR_BADH

`sword MsgSetCurPos(MSGH *msgh, 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.

`msgh' 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

`dword MsgGetCurPos(MSGH *msgh);`

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.

`msgh' 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

`UMSGID MsgMsgnToUid(MSG *mh, 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

`dword MsgUidToMsgn(MSG *mh, UMSGID umsgid, word type);`

The MsgUidToMsgn function converts a UMSGID to a message number.

`mh' 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

`dword MsgGetHighWater(MSG *mh);`

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.

`mh' 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

`sword MsgSetHighWater(MSG *mh, dword hwm);`

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

`mh' 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

`dword MsgGetTextLen(MSGH *msgh);`

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

`msgh' 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

`dword MsgGetCtrlLen(MSGH *msgh);`

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

`msgh' 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

`dword MsgGetCurMsg(MSG *mh);`

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

`mh' 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.

`dword MsgGetNumMsg(MSG *mh);`

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

`dword MsgGetHighMsg(MSG *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

`sword MsgValidate(word type, byte *name);`

The MsgValidate function validates a particular message area, and determines whether or not the area exists and 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.

`sword InvalidMsgh(MSGH *msgh);`

The InvalidMsgh function tests the given message handle for validity. If the message handle is valid, a value of 1 is returned. Otherwise, InvalidMsgh returns 0 and sets msgapierr to MERR_BADH.

`sword InvalidMh(MSG *mh);`

The InvalidMh function tests the given message area handle for validity. If the message handle is valid, a value of 1 is returned. Otherwise, InvalidMsgh returns 0 and sets msgapierr to MERR_BADH.