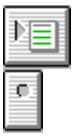
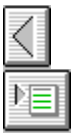


Message Compiler Outline

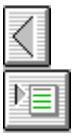
Utilities for the Windows NT SDK



Message Compiler (MC.EXE)

The Message Compiler (MC.EXE) converts message text files into binary files suitable for inclusion in a resource script (.RC file). The Resource Compiler is then used to places the messages into a resource for inclusion in an application or DLL. Applications that perfom event logging typically use an independent resource-only DLL that contains the messages, rather than carry the messages in the application image. See "Message Compiler Source Files" for complete information on the Message Compiler source-file format. For more information on event logging, see the event logging overviews in the Win32 API documentation.

Contents



Using the Message Compiler

The Message Compiler utility has the following command-line syntax:

Syntax

MC [**-v**] [**-w**] [**-s**] [**-h** *dir*] [**-r** *dir*] *filename*[**.MC**]...

Parameters

-v
Generates verbose output to stderr.

-w
Generates a warning message whenever an insert escape sequence is seen that is a superset of the type supported by the OS/2 MKMSGF utility. These are any escape sequences other than **%0** and **%***n*. This option is useful for converting MKMSGF message files to MC format.

-s
Adds an extra line to the beginning of each message that is the symbolic name associated with the message identifier.

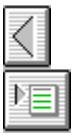
-c
Marks all messages as user-defined by setting the Customer code flag (the 29th bit of the 32-bit message). This bit can be used to determine if a message has come from the system or from an application.

-h *dirs*
Specifies the target directory of the generated include file. The include-file name is the base name of the .MC file with a .H extension.

-r *dir*
Specifies the target directory of the generated Resource Compiler script (.RC file). The script file name is the base name of the .MC file with a .RC extension.

filename[**.MC**]
Specifies one or more input message files that is compiled into one or more binary resource files, one for each language specified in the Message Compiler source files.

The Message Compiler reads the source file and generates a C/C++ include file containing definitions for the symbolic names. For each **LanguageId** statement, MC generates a binary file containing a message table resource. It also generates a single RC script file that contains the appropriate Resource Compiler statements to include each binary output file as a resource with the appropriate symbolic name and language type.



Message Compiler Source Files

Message Compiler source files (default extension .MC) are converted into binary resource files by the Message Compiler (MC.EXE). The binary resources are then passed to the Resource Compiler which puts them in the resource table for an application or DLL. For applications performing event logging, the messages are typically placed in a DLL that contains nothing but the message table. This DLL is registered by the application as the source of message text for the events that it logs. The application then uses the event logging APIs or the **FormatMessage** API to retrieve and use the message text.

Messages are defined using ASCII text in a text file. The Message Compiler source-format supports multiple versions of the same message text, one for each national language supported by your application. The Message Compiler automatically assigns numbers to each message, and generates a C/C++ include file for use by the application to access a message using a symbolic constant. The purpose of the message text file is to define all of the messages needed by an application, in a format that makes it easy to support multiple languages with the same image file.

General Syntax

The general syntax for lines in the message source file is:

*keyword***=***value*

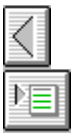
Spaces around the equal sign are ignored, and the value is delimited by white space (including line breaks) from the next *keyword***=***value* pair. Case is ignored when comparing against keyword names. The *value* portion can be a numeric integer constant using C/C++ syntax; a symbol name that follows the rules for C/C++ identifiers; or a file name that follows the rules for then base name of a file with the FAT file system (8 characters or less, with no period characters).

Comments

Comment lines are allowed in the source file. Place a semicolon (;) at the beginning of the line, and follow it with the C++ line-comment delimiter (//). For example:

```
;//This is a comment.
```

The leading semicolon prevents the comment line from being processed. The emitted comment line will begin with //.



Header Section

The overall structure of a message text file consists of a header which defines names and language identifiers for use by the message definitions in the body of the file. The header contains zero or more of the following statements:

MessageIdTypedef = [*type*]
SeverityNames = (*name***=***number*[**:***name*]**)**
FacilityNames = (*name***=***number*[**:***name*]**)**
LanguageNames = (*name***=***number***:***filename***)**

These keywords have the following meaning:

MessageIdTypedef = *type*

Gives a typedef name that is used in a type cast for each message code in the generated include file. Each message code appears in the include file with the format:

#define *name* **((***type*) **0x***nnnnnnnn***)**

The default value for *type* is empty, and no type cast is generated. It is the programmer's responsibility to specify a typedef statement in the application source code to define the type. The type used in the typedef must be large enough to accommodate the entire 32-bit message code.

SeverityNames = (*name***=***number*[**:***name*]**)**

Defines the set of names that are allowed as the value of the **Severity** keyword in the message definition. The set is delimited by left and right parentheses. Associated with each severity name is a number that, when shifted left by 30, gives the bit pattern to logical-OR with the **Facility** value and **MessageId** value to form the full 32-bit message code.

The default value of this keyword is:

```
SeverityNames=(
    Success=0x0
    Informational=0x1
    Warning=0x2
    Error=0x3
    )
```

Severity values occupy the high two bits of a 32-bit message code. Any severity value that does not fit in two bits is an error. The severity codes can be given symbolic names by following each value with **:***name*

FacilityNames = (*name***=***number*[**:***name*]**)**

Defines the set of names that are allowed as the value of the **Facility** keyword in the message definition. The set is delimited by left and right parentheses. Associated with each facility name is a number that, when shift it left by 16 bits, gives the bit pattern to logical-OR with the **Severity** value and **MessageId** value to form the full 32-bit message code.

The default value of this keyword is:

```
FacilityNames=(
    System=0x0FF
    Application=0xFFF
    )
```

Facility codes occupy the low order 12 bits of the high order 16-bits of a 32-bit message code. Any facility code that does not fit in 12 bits is an error. This allows for 4,096 facility codes. The first 256 codes are reserved for use by the system software. The facility codes can be given symbolic names

by following each value with **:***name*

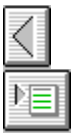
LanguageNames = (*name***=***number***:***filename***)**

Defines the set of names that are allowed as the value of the **Language** keyword in the message definition. The set is delimited by left and right parentheses. Associated with each language name is a number and a file name that are used to name the generated resource file that contains the messages for that language. The number corresponds to the language identifier to use in the resource table. The number is separated from the file name with a colon.

The initial value of **LanguageNames** is:

```
LanguageNames=(English=1:MSG00001)
```

Any new names in the source file which don't override the built-in names are added to the list of valid languages. This allows an application to support private languages with descriptive names.



Message Definitions

Following the header section is the body of the Message Compiler source file. The body consists of zero or more message definitions. Each message definition begins with one or more of the following statements.

MessageId = [*number*|**+***number*]
Severity = *severity_name*
Facility = *facility_name*
SymbolicName = *name*

The **MessageId** statement marks the beginning of the message definition. A **MessageID** statement is required for each message, although the value is optional. If no value is specified, the value used is the previous value for the facility plus one. If the value is specified as *+number* then the value used is the previous value for the facility, plus the number after the plus sign. Otherwise, if a numeric value is given, that value is used. Any **MessageId** value that does not fit in 16 bits is an error.

The **Severity** and **Facility** statements are optional. These statements specify additional bits to OR into the final 32-bit message code. If not specified they default to the value last specified for a message definition. The initial values prior to processing the first message definition are:

```
Severity=Success
Facility=Application
```

The value associated with **Severity** and **Facility** must match one of the names given in the **FacilityNames** and **SeverityNames** statements in the header section.

The **SymbolicName** statement allows you to associate a C/C++ symbolic constant with the final 32-bit message code.

The constant definition in the generated include file has the format:

//
// message text
//

#define *name* **((***type***) 0x***nnnnnnnn***)**

The comment before the definition is a copy of the message text for the first language specified in the message definition. The *name* is the value given in the **SymbolicName** statement. The *type* is the type name specified in the **MessageIdTypedef** statement. If no type was specified, the cast is not generated.

The following comment appears in each header file to explain the bit-fields in the 32-bit message:

```
//  Values are 32 bit values, laid out as follows:
//
//   3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
//   1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
//  +---+-+-+-----------------------+-------------------------------+
//  |Sev|C|R|     Facility          |               Code            |
//  +---+-+-+-----------------------+-------------------------------+
//
```

The meanings of the fields in the message code are:

Sev
The severity code:

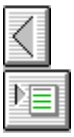
Bits	Meaning
00	Success
01	Informational
10	Warning
11	Error

C
The Customer code flag.

R
A reserved bit.

Facility
The facility code.

Code
The status code for the facility.



Language-Specific Message Text Definitions

After the message definition statements, you specify one or more message text definitions.

Syntax

Language=*language_name*
messagetext
.

Each message begins with a **Language** statement that identifies the binary output file for this message. The first line of the message text begins with the next line. The message text is terminated by a line containing a single period at the beginning of the line, immediately followed by a new line. No spaces are allowed around the terminating period. Within the message, blank lines and white space are preserved as part of the message.

You can specify several escape sequences for formatting the message when the message text is used by the application or an event viewer. The percent sign character (**%**) begins all escape sequences.

%0
Terminates a message text line without a trailing newline. This can be used to build up long lines or to terminate the message without a trailing newline, which is useful for prompt messages.

%*n*[**!***printf-format-specifier***!**]
Identifies an insert. Each insert refers to a parameter used in a call to the **FormatMessage** API. **FormatMessage** returns an error if the message text specifies an insert that was not passed to **FormatMessage**.

The value of *n* can be between 1 and 99. The **printf** format specifier must be enclosed in exclamation marks. It is optional and defaults to !s! if not specified.

The **printf** format specifier can contain the ***** specifier for either the precision or width components. When specified, they consume inserts numbered *n*+1 and *n*+2 for their values at run time. MC prints a warning message if these inserts are specified elsewhere in the message text.

Any character following a percent sign other than a digit is formatted in the output message without the percent sign.

You can specify the following additional escape sequences:

%%
Generates a single percent sign in the formatted message text.

%
Generates a hard line break when it occurs at the end of a a line. Useful when **FormatMessage** is supplying normal line breaks so the message fits in a certain width.

%r
Generates a hard carriage return, without a trailing newline character.

%b
Generates a space character in the formatted message text. This can be used to insure there are the appropriate number of trailing spaces in a message text line.

%.
Generates a single period character in the formatted message text. This can be used to get a period at the beginning of a line without terminating the message definition.

%!
Generates a single exclamation point in the formatted message text. This can be used to specify an

exclamation point immediately after an insert.