Programmatic Support for Windows NT Services

Note that wherever this document refers to Windows NT, the comments apply to both Windows NT 4, Windows 2000, and later versions, unless stated otherwise.
You can find the latest version of this document on-line at .

LOGGER is a general-purpose library for logging text messages to one or more destinations, including `stdout`, operating systems files or the Windows NT Event Log. It also includes a set of macro definitions which are useful for embedding debug statements into compiled code. These debug statements can be preprocessed out of the release version of code.

1 Copyright and Distribution

2 Effectivity

This article applies to the following versions of software:

Software	Version	Platform
LOGGER	2.1	Windows NT 4 (Service Pack 4+)

Table 1: Effectivity

You can check the version of **LOGGER** by running the following shell command against it (new in version 2.1):

```
find "***" logger.dll
```

You will see output similar to the following:

```
*** LOGGER version 2.10 Copyright (C) 1998 - 2000 Nick Rozanski (##############)
```

LOGGER should work on Windows 95 / 98 (apart from the Event Log functionality) but I have not tested it on these platforms. Support for the Sybase functionality is being phased out (I no longer work for Sybase).

3 Pre-Requisites

3.1 Windows NT

To run **LOGGER** on Windows NT you require the following files somewhere in your PATH at runtime.

- `logger.dll` - **LOGGER** library
- `msvcrt.dll` - Microsoft C++ run-time library

At compile/link time, you will need the following:

- `logger.h` - **LOGGER** header file
- `logger.lib` - **LOGGER** library definitions

All of these components are available from my website.

3.2 Linux

To run **LOGGER** on Linux you require the following file in a directory which is searched for library files.

- `liblogger.so` - **LOGGER** library

Library files are usually placed in `/usr/lib`, but check `/etc/ld.so.conf` for your configuration.

At compile/link time, you will need the following:

- `logger.h` - **LOGGER** header file
- `liblogger.so` - **LOGGER** library definitions

You should supply the flag `-llogger` to `gcc` to build a program which uses the logger.

4 Background

LOGGER allows you to define up to `LOGGER_MAX_LOGGERS` (30) different loggers, although typically you may only need one. Each logger has a different destination, such as `stdout`, a file or the Windows NT Event Log. You define loggers using `LoggerConfigure`.

When you write a message, **LOGGER** sends it to *each* of the loggers you have defined, in turn, if it meets the filter criteria for that logger

LOGGER includes (some of) the following elements in the message:

1. the host (computer) name
2. the application
3. the current date and time
4. the message class (eg `DEBUG` or `INFO`)
5. the message severity
6. the thread id
7. the source file
8. the line number within the source file
9. the function name
10. the text of the message

Some of these parameters (such as the hostname) may be omitted. See the definition of `LoggerWriteMessage` for details.

On Windows NT, the **LOGGER** DLL uses Win32 "Critical Sections", so is thread-safe.

5 Synopsis

LOGGER exports the following functions.

5.1 LoggerConfigure

`LoggerConfigure` allows you to configure a logger, that is, define its destination.

You must call `LoggerConfigure` at least once before calling `LoggerWriteMessage`. Call this function multiple times to get messages written to multiple destinations.

The function prototype is as follows.

```
int LoggerConfigure
(
    LOGGER_ID  LoggerId,
    char      *Hostname,
    char      *Application,
    int        Destination,
    void      *DestDetails1,
    void      *DestDetails2,
    int       *ErrorPtr,
    char      *ErrorMsgPtr
)
```

`LoggerConfigure` takes the following parameters.

LoggerId

The identifier of the logger to configure. Identifiers start at zero and must be less than `LOGGER_MAX_LOGGERS` (30). To use the "default" logger, set this parameter to `LOGGER_DEFAULT_LOGGER`. To get the next unused logger, call the function `LoggerGetUnusedLogger()`. It is not necessary to define loggers contiguously.

Hostname

Defines the "hostname" part of logged messages. This can be the null string, which causes it to be ignored.

Application

Defines the "application" part of logged messages. This can be the null string, which causes it to be ignored.

Destination

Defines the destination for this logger, one of the following.

- LOGGER_FMTONLY - construct the message and write it to memory location DestDetails1 (which must point to a static buffer at least LOGGER_BUFFERSIZE bytes in size)
- LOGGER_ANSI_STDOUT - use ANSI 'C' functions to write to stdout
- LOGGER_ANSI_FILENAME - use ANSI 'C' functions to write to the file named (*DestDetails1). LoggerConfigure will open the file in create or append mode depending on the value of *DestDetails2.
- LOGGER_ANSI_FILEPTR - use ANSI 'C' functions to write to the file which has previously been opened by the calle using file pointer (*DestDetails1)
- LOGGER_WIN32_CONSOLE - use Win32 functions to write to console with handle (*DestDetails1)
- LOGGER_WIN32_FILENAME - use Win32 functions to write to the file named (*DestDetails1). LoggerConfigure will open the file in create or append mode depending on the value of *DestDetails2.
- LOGGER_WIN32_FILEHANDLE - use Win32 functions to write to the file which has previously been opened by the caller, using file handle (*DestDetails1)
- LOGGER_WIN32_EVENTLOG - write to the Windows NT Event Log on the computer named (*DestDetails1)
- LOGGER_SYBASE_SRVLOG - write to the Sybase Open Server error log (this requires that the Sybase Open Server libraries can be found in %PATH%)
- LOGGER_UNIX_SYSLOG - write to the Linux system logger (syslogd)

DestDetails1

The meaning of DestDetails1) depends on the value of _CODE(Destination as follows. (W means Windows only; L means Linux only; WL means both.)

OS	Destination	DestDetails1	Meaning
WL	LOGGER_FMTONLY	char*	the address of a buffer into which formatted messages will be written; must have static scope
WL	LOGGER_ANSI_STDOUT	*ignored*	
WL	LOGGER_ANSI_FILENAME	char*	a null-terminated string specifying the pathname of the file to write to
WL	LOGGER_ANSI_FILEPTR	FILE*	previously opened by caller for writing, specifying the file to append to
W	LOGGER_WIN32_CONSOLE	HANDLE*	previously opened by caller for writing using GetStdHandle) or _CODE(AllocConsole
W	LOGGER_WIN32_FILENAME	char*	a null-terminated string specifying the pathname of the file to write to
WL	LOGGER_WIN32_FILEHANDLE	HANDLE*	previously opened by caller for writing, specifying the file to append to
W	LOGGER_WIN32_EVENTLOG	char*	a null-terminated string specifying the computer name on which the message should be logged; NULL or the empty string to log to this computer
WL	LOGGER_SYBASE_SRVLOG	*ignored*	
L	LOGGER_UNIX_SYSLOG	*ignored*	

DestDetails2

For LOGGER_ANSI_FILENAME and LOGGER_WIN32_FILENAME, DestDetails2), if not _CODE(NULL, should be a pointer to a boolean value ((int)0 or 1). If non-zero, the output file will be truncated by LoggerConfigure before it starts writing messages (new in version 2.1).

DestDetails2 is ignored for other destination types.

ErrorPtr

If LoggerConfigure fails, and ErrorPtr is not NULL, LoggerConfigure sets (*ErrorPtr) to an integer containin the ANSI or Win32 error code.

ErrorMsgPtr

If LoggerConfigure fails, and ErrorMsgPtr is not NULL, LoggerConfigure writes an error message into (*ErrorMsgPtr).

5.2 LoggerSetFilter

`LoggerSetFilter` defines a *filter* for a configured logger (new in version 2.1). Only messages which meet the filter criteria are logged in subsequent calls to `LoggerWriteMessage`. Messages which do not meet all of the criteria for that logger are silently discarded by `LoggerWriteMessage`. By default, all messages are logged.

Note that a message must meet all criteria defined by `LoggerSetFilter` before it is logged by a logger. For example, if criteria are specified for message class and for source file, only messages of that class *and* for that source file are logged.

The function prototype is as follows.

```
void LoggerSetFilter
(
        LOGGER_ID LoggerId,
        int   FilterAll,
        int   MsgClass,
        int   MsgSeverity,
        int   ThreadId,
        char *SourceFile,
        char *FuncName
)
```

`LoggerSetFilter` takes the following parameters.

LoggerId

The identifier of the logger to configure. This must have already been configured using `LoggerConfigure`.

FilterAll

If true (non-zero) then all filters for the logger are turned off (ie all messages are logged). In this case the remaining parameters are ignored.

MsgClass

`MsgClass` defines the class(es) of the message to be logged. `MsgClass` is a bitwise OR of one or more of the following values.

- `LOGGER_BARE_FILTER` - if set, log messages of MsgClass `LOGGER_BARE`
- `LOGGER_INFO_FILTER` - if set, log messages of MsgClass `LOGGER_INFO`
- `LOGGER_WARN_FILTER` - if set, log messages of MsgClass `LOGGER_WARN`
- `LOGGER_ERROR_FILTER` - if set, log messages of MsgClass `LOGGER_ERROR`
- `LOGGER_DEBUG_FILTER` - if set, log messages of MsgClass `LOGGER_DEBUG`
- `LOGGER_AUDIT_SUCCESS_FILTER` - if set, log messages of MsgClass `LOGGER_AUDIT_SUCCESS`
- `LOGGER_AUDIT_FAILURE_FILTER` - if set, log messages of MsgClass `LOGGER_AUDIT_FAILURE`

To log messages of all classes, set this parameter to `LOGGER_ALL_CLASSES_FILTER`.

MsgSeverity

If `MsgSeverity` is not negative, then only messages of this severity or greater will be logged. To ignore message severity, set this parameter to `-1`.

ThreadId

If `ThreadId` is not negative, then only messages from this thread will be logged. To ignore thread id, set this parameter to `-1`.

SourceFile

If `SourceFile` is `NULL` or an empty string, then the source file is ignored. Otherwise, only messages from this source file will be logged. The rule here is inclusion, so it is not necessary to include a complete path name.

FuncName

If `FuncName` is `NULL` or an empty string, function name is ignored. Otherwise, only messages for this function will be logged.

5.2.1 LoggerSetFilter Example

For example, to log only debug and error messages from the file `..\myprogram.c` for logger `1`, make the following call:

```
LoggerSetFilter(
        1,                                          // LoggerId
        0,                                          // FilterAll
        LOGGER_DEBUG_FILTER|LOGGER_ERROR_FILTER,    // MsgClass
        -1,                                         // MsgSeverity
        -1,                                         // ThreadId
        "myprogram.c",                              // SourceFile
        NULL                                        // FuncName
)
```

```
// later ...
LoggerWriteMessage( ... ,"oldprogram.c", ...)             // message discarded
LoggerWriteMessage(LOGGER_INFO, ...)                      // message discarded
LoggerWriteMessage(LOGGER_DEBUG, ... ,"myprogram.c", ...) // message logged
```

5.3 LoggerGetUnusedLogger

`LoggerGetUnusedLogger` retrieves the id of the smallest unused logger. If there are no unused loggers it returns `LOGGER_NO_UNUSED_LOGGER`.

The function prototype is as follows.

```
LOGGER_ID LoggerGetUnusedLogger();
```

5.4 LoggerWriteMessage

`LoggerWriteMessage` formats the supplied message and writes it to all configured loggers. The function prototype is as follows.

```
void LoggerWriteMessage
(
    int  MsgClass,
    int  MsgSeverity,
    int  ThreadId,
    char SourceFile[],
    int  LineNumber,
    char FuncName[],
    char MsgText[],
    ...
)
```

`LoggerWriteMessage` takes the following parameters.

MsgClass

`MsgClass` defines the class of the message, that is how it should be viewed by the recipient. `MsgClass` takes one of the following values.

- `LOGGER_BARE` - bare message (no other elements added to it)
- `LOGGER_INFO` - this is an informational message
- `LOGGER_WARN` - this is a warning message
- `LOGGER_ERROR` - this is an error message
- `LOGGER_DEBUG` - this is a debug message
- `LOGGER_AUDIT_SUCCESS` - this is an audit (success) message
- `LOGGER_AUDIT_FAILURE` - this is an audit (failure) message

MsgSeverity

`MsgSeverity` is an application-specific integer. It has no particular meaning to **LOGGER**, although by convention severities are greater than zero, and a larger severity means a more severe error.

ThreadId

`ThreadId` is the identifier of the calling thread. This supports programs which use NT native threads, and those which implement their own threading.

If set to `-1`, it is ignored; if set to `-2`, the value returned by the Win32 call `GetCurrentThreadId` is used.

SourceFile, LineNumber, FuncName

These parameters define where in the source the message came from. They represent the source file (eg `__FILE__`), line number (eg `__LINE__`), and function name respectively. The source file and function can be the empty string, in which case they will be ignored; similarly a line number less than zero will be ignored.

MsgText, ...

This is the text of the message. It may include `sprintf` conversion specifiers such as `%s`, `%d` etc. These must match the remaining arguments to the function, as if they were being passed to `sprintf`.

5.4.1 LoggerWriteMessage Example

A typical call to `LoggerWriteMessage` might look as follows.

```
void LoggerWriteMessage
(
    LOGGER_ERROR,                       // MsgClass
```

```
        4,                                  // MsgSeverity,
        -2,                                 // ThreadId (NT native thread id)
        __FILE__,                           // SourceFile
        __LINE__,                           // LineNumber
        "myfunc",                           // FuncName
        "failed to open file %s, error=%d", // MsgText
        myfilename,                         // 1st variable argument %s
        errorcode                           // 2nd variable argument %d
    )
```

It is much easier to make these calls using the **LOGGER** macros.

5.5 LoggerMarkUnused

`LoggerMarkUnused` marks a logger as unused. Messages are no longer sent to that target.

The function prototype is as follows.

```
void LoggerMarkUnused(LOGGER_ID LoggerId);
```

`LoggerMarkUnused` takes the following parameters.

LoggerId

The identifier of the logger to configure. Identifiers start at zero and must be less than `LOGGER_MAX_LOGGERS` (30). To mark the "default" logger unused, set this parameter to `LOGGER_DEFAULT_LOGGER`.

5.6 LoggerSetDebugLevel

`LoggerSetDebugLevel` sets the current debug level. The function prototype is as follows.

```
int LoggerSetDebugLevel(int DbgLvl);
```

This is a global value which can be used as required by callers. The **LOGGER** routines themselves do not pay heed to this value. It is only used by the **LOGGER** macros.

5.7 LoggerGetDebugLevel

`LoggerGetDebugLevel` returns the current debug level. The function prototype is as follows.

```
int LoggerGetDebugLevel();
```

6 Macros

The **LOGGER** header file `logger.h` contains a number of macros useful for including debug messages in compiled code. These macros are described below.

LOGGER_SET_DEBUG_LEVEL(*d*)

`LOGGER_SET_DEBUG_LEVEL(`*d*) sets the debug level, at run time, to *d*, which is one of the following:

- a negative value means all debug, information and error messages are logged
- a value of zero means all information and error messages are logged
- a positive value means error messages only are logged

LOGGER_LOG_DEBUG(*m*)
LOGGER_LOG_DEBUG1(*m*,*p1*)
LOGGER_LOG_DEBUG2(*m*,*p1*,*p2*)
LOGGER_LOG_DEBUG3(*m*,*p1*,*p2*,*p3*)
LOGGER_LOG_DEBUG4(*m*,*p1*,*p2*,*p3*,*p4*)

If the macro `_DEBUG` is defined, these macros log a debug message (ie make a call to `LoggerWriteMessage`). (`_DEBUG` is set by compilers, including MS Visual C++, to indicate a debug build.) *m* is the text of the message. It may include `sprintf` conversion specifiers such as `%s`, `%d` etc. These must match the remaining arguments to the macro, as if they were being passed to `sprintf`.

If `_DEBUG` is undefined, these macros do nothing. This effectively removes all debug code from Release executables.

LOGGER_LOG_INFO(*m*)
LOGGER_LOG_INFO1(*m*,*p1*)
LOGGER_LOG_INFO2(*m*,*p1*,*p2*)

LOGGER_LOG_INFO3(*m*,*p1*,*p2*,*p3*)
LOGGER_LOG_INFO4(*m*,*p1*,*p2*,*p3*,*p4*))

These macros log an informational message (ie make a call to `LoggerWriteMessage`). They are unaffected by the value of `_DEBUG`.

LOGGER_LOG_ERROR(*m*)
LOGGER_LOG_ERROR1(*m*,*p1*)
LOGGER_LOG_ERROR2(*m*,*p1*,*p2*)
LOGGER_LOG_ERROR3(*m*,*p1*,*p2*,*p3*)
LOGGER_LOG_ERROR4(*m*,*p1*,*p2*,*p3*,*p4*))

These macros log an error message (ie make a call to `LoggerWriteMessage`). They are unaffected by the value of `_DEBUG`

6.1 Macro Examples

Some example macro invocations are given below.

```
LOGGER_LOG_ERROR2("failed to open file %s, error=%d",myfilename,errorcode)

LOGGER_LOG_DEBUG4("parameters are %s, %d, %d, %c",filename,filesize,length,openmode)
```

The second of these two macros will preprocess out to a null statement in release code.

7 Bugs

- `LOGGER_ANSI_FILENAME` sometimes causes a core dump if **LOGGER** is unable to create the file (eg if the directory does not exist).
- On NT, `LOGGER_SYBASE_SRVLOG` causes a (non-fatal) stack corruption in some cases. This seems to be a problem with the use of `GetProcAddress` to find the entry point for `srv_log`, but as I do not work for Sybase any longer I can't investigate this. It is likely that support for `LOGGER_SYBASE_SRVLOG` will be withdrawn in the future.

8 Version History

Version 2.0 (30 November 1998)

First release version.

Version 2.1 (30 June 2000)

Simplified build; added filtering functionality and ability to truncate log files.

 url: http://localhost.rozanski.org.uk/index.php?page=logger last changed: 7 March 2017