Cross-platform event logging in Object Pascal

Micha¨el Van Canneyt

June 24, 2007

1 Introduction

Often, a program which works in the background or sits in the windows system tray needs to write some diagnostic log message to some log. Both Linux and Windows NT or higher provide a system logging facility: A service or program which collects messages from various client applications, and writes them in a structured manner to some set of files, which can be viewed at any later time. In this article, a component is presented which can be used in an application to log messages to the system log - or in the case of Windows 95 - to a file. The component works in Delphi, Kylix and is distributed standard with Free Pascal. Before presenting the component itself, the log facility of Linux and Windows is explained.

2 Logging under Linux

The system log facility under Linux offers a slightly simpler interface than under Windows, so this one will be presented first. The system log under linux is handled by the syslogd daemon, which is started at system boot. It listens on a designated Unix socket (normally /dev/log) for incoming connections and log messages. It then writes any received messages to a set of files, normally located under /var/log. Optionally, log messages can be forwarded to a remote host on the network.

The C library contains a logging API which connects to the socket on which the system log daemon listens and writes messages to the sockets, or, if this fails, to the console. Essentially, the logging process requires 3 calls:

```
Procedure openlog(Ident : pchar; Option : Integer; Facility: Integer);cdecl;

Procedure closelog;cdecl;

Procedure syslog(priority : Integer; Fmt: Pchar);cdecl;varargs;
```

The first two of these calls are quite straightforward:

openlog will open the logging facility, which basically means opening the connection to the socket the syslog daemon listens on. This must be called prior to any attempt to write messages to the log facility. The first argument `Ident`, is a string which identifies the program that is logging. It will be prepended to any message written to the log files. The `Option` parameter can be used to set various options for the logging process. Finally, the `Facility` option determines the class of the messages

that are written to the log. The system log process (called syslogd) will use this to determine where the log messages are written. this is determined by the system configuration (usually in /etc/syslog.conf). A detailed description of all options and possible values for the `Facility` argument can be found in the `syslog` page in section 3 of the linux manual pages.

closelog will close the logging facility. This should be called after logging is finished.

Depending on how many messages are written, it is preferable to open the log before writing a message, and to close it after the message was written: Keeping the connection open consumes system resources.

The actual logging is done using the `syslog` call. As can be seen in the above declaration, this function accepts a variable number of arguments. The first argument is a priority. According to the manual pages, this can be one of

LOG_ALERT action must be taken immediately.

LOG_CRIT critical conditions.

LOG_ERR error conditions.

LOG_WARNING warning conditions.

LOG_NOTICE normal, but significant, condition.

LOG_INFO informational message

LOG_DEBUG debug-level message

The second argument is a format string, which will be used by the system to format the arguments that follow it, in a manner similar to the standard Delphi `Format` function. It can be followed by an arbitrary number of arguments. A description of the actual format string can be found e.g. in the `sprintf` manual page.

The following small program does nothing useful, but demonstrates the logging process:

```
program testlin;

uses Libc,SysUtils;

var
  i : Integer;
  prefix : ansistring;

begin
  i:=getpid;
  prefix:=format('testlog[%d] ',[i]);
  openlog(pchar(prefix),LOG_NOWAIT,LOG_DEBUG);
  for i:=1 to 10 do
    syslog(log_info,'This is message number %d',i);
end.
```

3 Logging under Windows

Under Windows NT, 2000 and XP, there is also a system log API. It is similar to the linux API, but requires a bit more setup. The system log can be viewed using the 'Event viewer'

program - the location of the log files is not public as under linux, only the event viewer should be used to view and manipulate the system logs. The messages are divided in separate 'logs', the default 'Application log', 'System log' and 'Security log'. More can be defined if need be, but the above should be available by default on all systems.

As under linux, 3 calls must be used in the logging process:

```
function OpenEventLog(lpUNCServerName, lpSourceName: PChar): THandle; stdcall;
function CloseEventLog(hEventLog: THandle): BOOL; stdcall;

function ReportEvent(hEventLog: THandle; wType, wCategory: Word;
  dwEventID: DWORD; lpUserSid: Pointer; wNumStrings: Word;
  dwDataSize: DWORD; lpStrings, lpRawData: Pointer): BOOL; stdcall;
```

The first two calls are quite similar to their linux counterparts:

OpenEventLog Creates a connection to the event log, returning a handle which must be used in subsequent calls. The `lpSourceName` argument is identical in purpose to the `Ident` argument under linux: it identifies the program to the logging system, and is displayed in the event viewer (more on this will be said later on). The `lpUNCServerName` can be used to perform remote logging: it identifies the name of a computer on the network to which events must be logged. This is different from linux, where the system log daemon decides whether messages should be forwarded to another machine.

CloseEventLog closes the event log again. The handle obtained through the `OpenEventLog` must be passed to this function.

The actual event logging call is slightly more complicated than its Linux counterpart, as it has more options. Fortunately, some of them can be ignored for most practical purposes:

hEventLog This is the handle obtained through the `OpenEventLog` call.

wType This is the event type. This can be one of several pre-defined constants. For each type of event, the event viewer will display a different icon in the event log.

wCategory This parameter determines the category of the message. It can be any value. This value will be displayed in the overview of the event log viewer, and can be mapped to a string.

dwEventID The event ID. The event ID will also be displayed in the event viewer. As well as the Category ID, it can be mapped to a string to be displayed in the event viewer.

lpUserSid This is a pointer to the current user's security profile. It can be ignored for most purposes.

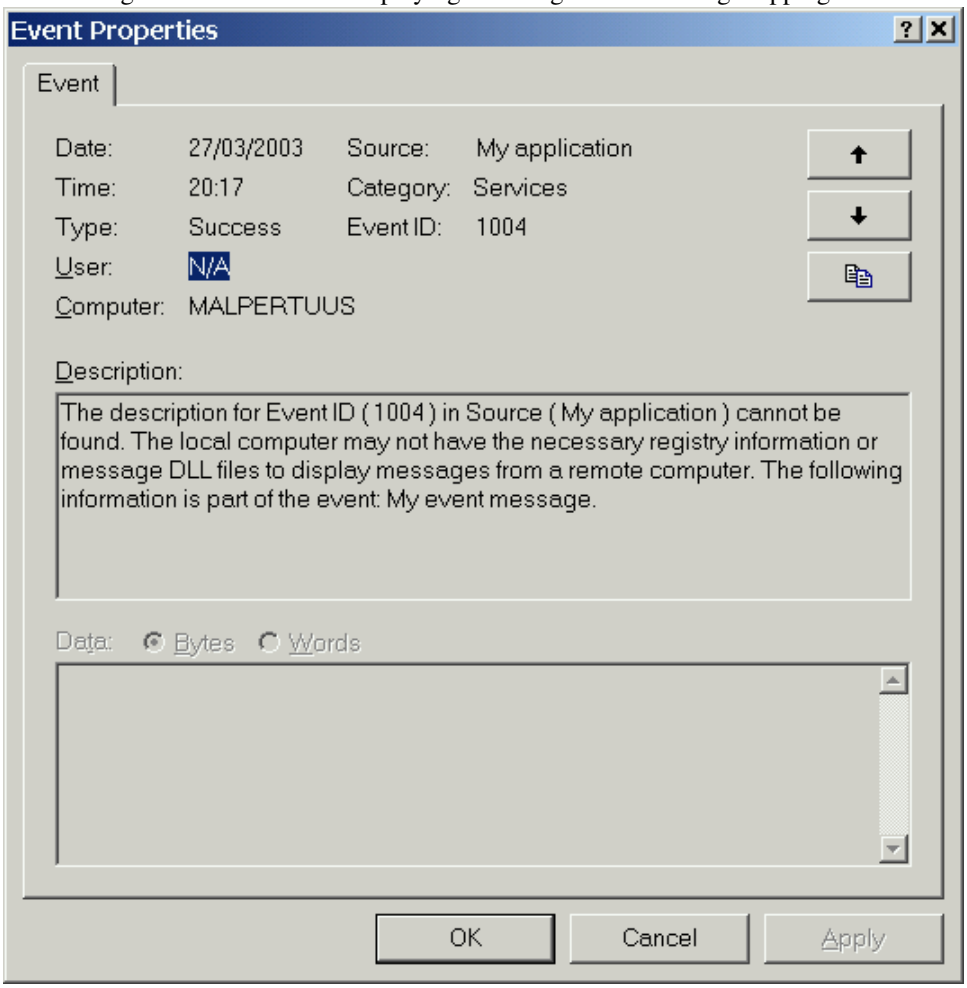
wNumStrings This is the number of strings that is pointed to by the `lpStrings` argument.

dwDataSize This is the size (in bytes) of the binary data pointed to by the `lpRawData` argument.

lpStrings A pointer to an array of null-terminated strings - or `Nil`. The strings will be merged into placeholders found in the message corresponding to the EventID - more about this follows.

lpRawData A pointer to a buffer containing binary data - or `Nil`. The size of this buffer must be specified in the `dwDataSize` argument.

Figure 1: Event viewer displaying a message without string mappings



.

The event type can be one of the following pre-defined constants:

EVENTLOG_SUCCESS Success event.

EVENTLOG_ERROR_TYPE Error event.

EVENTLOG_WARNING_TYPE Warning event.

EVENTLOG_INFORMATION_TYPE Information event.

EVENTLOG_AUDIT_SUCCESS Audit success event.

EVENTLOG_AUDIT_FAILURE Audit failure event.

Thus far, the Windows interface does not differ substantially from the one on Linux. It can be used without further preparations if need be.

However, there is a small caveat. The event viewer expects to find mappings for the category and EventID to a descriptive string. If these mappings are not present, the messages logged will look as in picture **??**: a standard error message, followed by the strings passed to the `ReportEvent` function.

4 Creating message tables

To get rid of the ugly error message in the event viewer, mappings from event categories or event IDs to strings must be created. This can be done using a message file. A message file consists of numbered messages, which can be compiled to a resource file, and which can be linked in a message library or even in an ordinary executable. The location of this library or executable should be registered in the Windows registry. The Event viewer will then extract the messages from the resources, and use them while displaying logged events.

For each category used when logging events, a message with MessageID corresponding to the category must be present in the message file. Likewise, for each event ID, a message with the same ID must be found. The category names should be short and descriptive. The event ID messages can be longer.

A partial message file would look like this:

```
; Categories (1-4)
MessageId=1
SymbolicName=ECInfo
Language=English
Information
.

MessageId=2
SymbolicName=ECWarning
Language=English
Warning
.

; Message Definitions (1000-1004)
MessageId=1000
Language=English
Error: %1.
.
```

The file format is quite simple. It consists of a series of keywords identifiying messages and their properties. The most important ones are

MessageID this is the ID of the message, and starts a new message definition.

Language the language of the message. A mapping of language names to ID's can be defined as well.

SymbolicName is not important for the compiled messages, but can be used to create a pascal (or C) unit with constants that identify the messages, as follows:

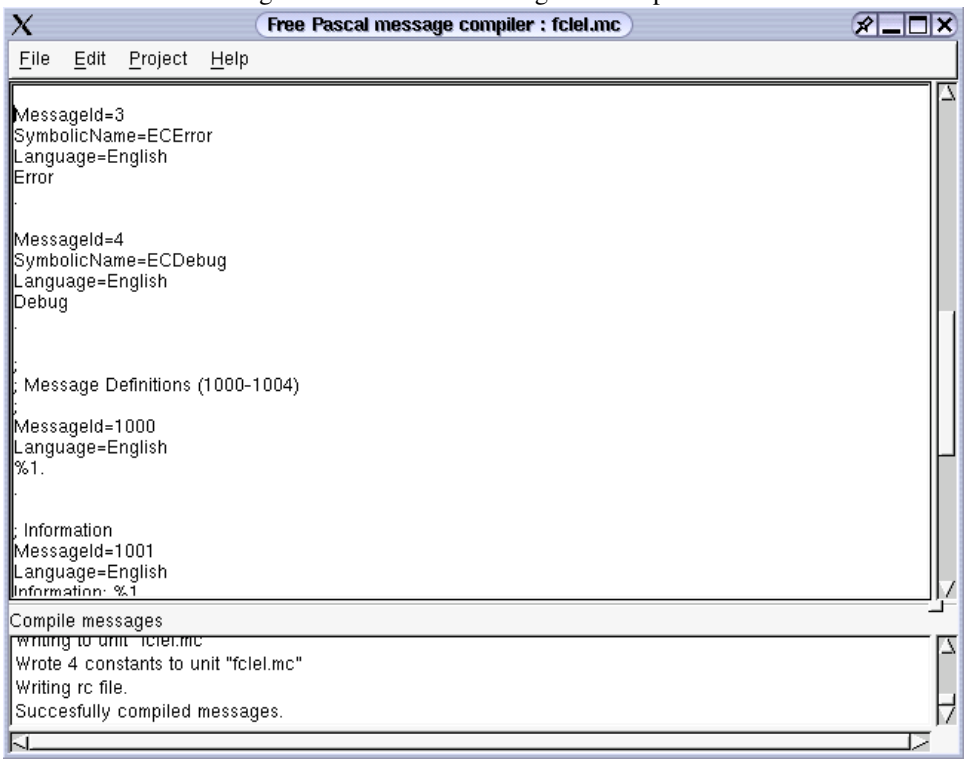
```
ECInfo    = 1;
ECWarning = 2;
```

After the keywords, the message text follows. It is ended by a line containing just a period (.) character. The file can also contain comments, they start with a semicolon. A full reference of the message file format can be found on the MSDN website.

Unfortunately, Delphi does not have a message file compiler, but the Borland resource compiler can create message tables in resource files.

Free Pascal has a simple command-line message file compiler, and a GUI version as well. Starting from a message file (traditionally using extension .mc), it can create 3 files:

Figure 2: Free Pascal message file compiler



1. A compiled message file. (extension .msg)
2. A resource script file (extension .rc) to create a resource file (.res) which can be linked in a library or executable.
3. A pascal unit containing constants which correspond to the SymbolocName identifier found in the mesage file.

The GUI version of the message compiler can be seen in figure **??**

After the .rc resource script and .msg compiled message file have been created, the borland resource compiler (brcc32) or the freely available windres tool can be used to create a .res resource file, which can be linked into a binary using the `{$R}` directive.

To use the created message tables (linked in some binary), they must be registered with the event log system. This can be done by creating some new key and some entries in the registry, providing amonst others the path to the binary containing the messages. The new registry key should be located under

```
\SYSTEM\CurrentControlSet\Services\EventLog\Application\
```

and should have the same name as the identifier used when opening the event log. In this key, the following entries should be created:

CategoryCount The number of categories used by the application.

EventMessageFile Path to the executable or library containing the message table with event messages.

CategoryMessageFile Path to the executable or library containing the message table with category descriptions.

TypesSupported Bitmask of supported event types. This can be constructued by 'Or'-ing the various used event types together.

Using these entries in the registry, the event viewer will be able to show descriptive strings for the event categories used by the logging program.

The various event IDs refer to messages which will be displayed in the actual log message. They can contain placeholders of the form `%N` when `N` is a digit starting at 1. The placeholders will be replaced by the corresponding strings passed in the `lpStrings` argument of the `ReportEvent` call.

5 The `TEventLog` component

In order to simplify the handling of logging in an application, the `TEventLog` component was written. It can be dropped on a form in Delphi or Kylix, and can be used in e.g. the Lazarus IDE when working with Free Pascal. The component encapsulates the logging API of Windows and Linux, (and OS/2 when using Free Pascal) and presents a unified interface. For Windows programs, it also takes care of registering message files. It can also write the log messages to a file instead of the system log. (in fact, for DOS programs developed with Free Pascal, this is the default).

The component offers the following methods which can be used to write to the log file or sytem event log:

```
Procedure Log(EventType: TEventType; Msg: String);
Procedure Log(EventType: TEventType; Fmt: String; Args: Array of const);
Procedure Log (Msg : String);
Procedure Log (Fmt : String; Args : Array of const);
```

The `Log` variant without `EventType` parameter uses the `DefaultventType` property to determine what type the message has. The variant with the `Fmt` and `Args` parameters uses the Delphi `Format` function to format the message.

The `EventType` parameter determines the type of the log message. It is converted to a system-dependent type and can be one of the following:

etCustom A custom type message. The actual used message type depends on the `CustomLogType` property or the `OnGetCustomEvent` event handler.

etInfo An informational message.

etWarning A warning.

etError An error.

etDebug A debug message.

For convenience and ease of use, each of these types has a corresponding method:

```
Procedure Warning (Msg: String);
Procedure Warning (Fmt: String; Args: Array of const);
Procedure Error (Msg: String);
Procedure Error (Fmt : String; Args : Array of const);
Procedure Debug (Msg : String);
Procedure Debug (Fmt : String; Args : Array of const);
Procedure Info (Msg : String);
Procedure Info (Fmt : String; Args : Array of const);
```

The TEventLog component has the following design-time properties:

Identification This property is used when opening the sysem log, to identify the program.

LogType Determines the type of logging: ltFile means that all log messages are sritten to a log file. The default value of ltSystem tells the component to write to the system log.

Active Setting this property to True opens the log. This must be done before using any of the logging methods. Setting it to False will close the log again.

DefaultEventType Is the event type used by the Log call if no type is provided.

FileName Is the filename to use when writing log messages to file. If no filename is specified, a name will be chosen. This name is system dependent.

TimeStampFormat Can be used to specify the timestamp used when writing to a file log.

CustomLogType Specified the type of log message used when 'etCustom' is specified. The meaning of this value is system dependent.

EventIDOffset Used on Windows only: When converting an event ype to an event ID for the windows system log, the event ID is calculating by adding this property's value to the ordinal value of the EventType parameter. It is 1000 by default.

To give more control over the logging process in Windows, custom messages can be used in combination with the following events to determine the Category, Event ID and Message type:

OnGetCustomCategory

OnGetCustomEventID

OnGetCustomEvent

To support the use of message tables, the following call is introduced:

```
Function RegisterMessageFile(AFileName : String) : Boolean;
```

It will create the necessary entries in the registry to register a message file. The FileName argument is the name of the executable or library that contains the message tables. If it is left empty, the name of the current executable is used. The fclel.res files that comes with the component contains a message table with definitions for all needed categories and events. It can be linked into the program, and will be used by the event viewer to display message sent by the TEventLog component. The RegisterMessageFile does nothing on other platforms.

The usage of this component is simple: drop it on a form. When logging is needed, set the active property to True, and call the Log method or one of it's variants. A small demo program is provided on the disk accompagnying this issue. It should compile under Delphi 6 or Kylix. Free Pascal delivers a similar program which uses the gtk toolkit on Windows and Linux. The result on Windows - with the event log showing the sent messages - can be seen in figure **??**. The result under Linux - compiled using Free Pascal, can be seen in figure **??**. The top X-term shows the messages as they appear in the system's messages log file.

Figure 3: Event log in action on Windows

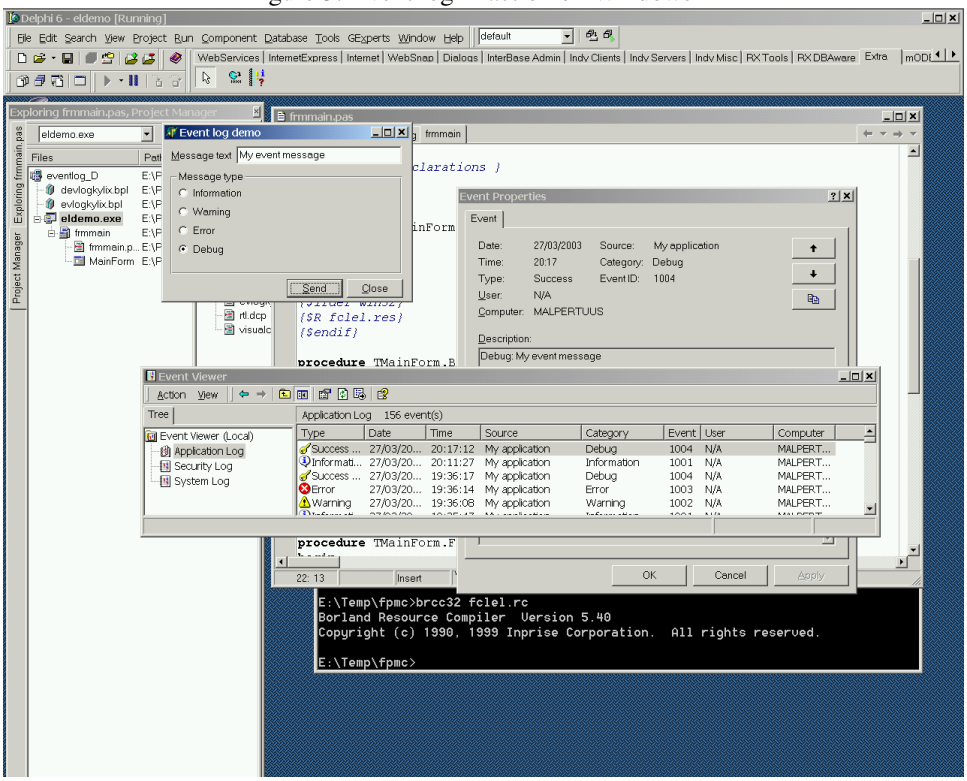
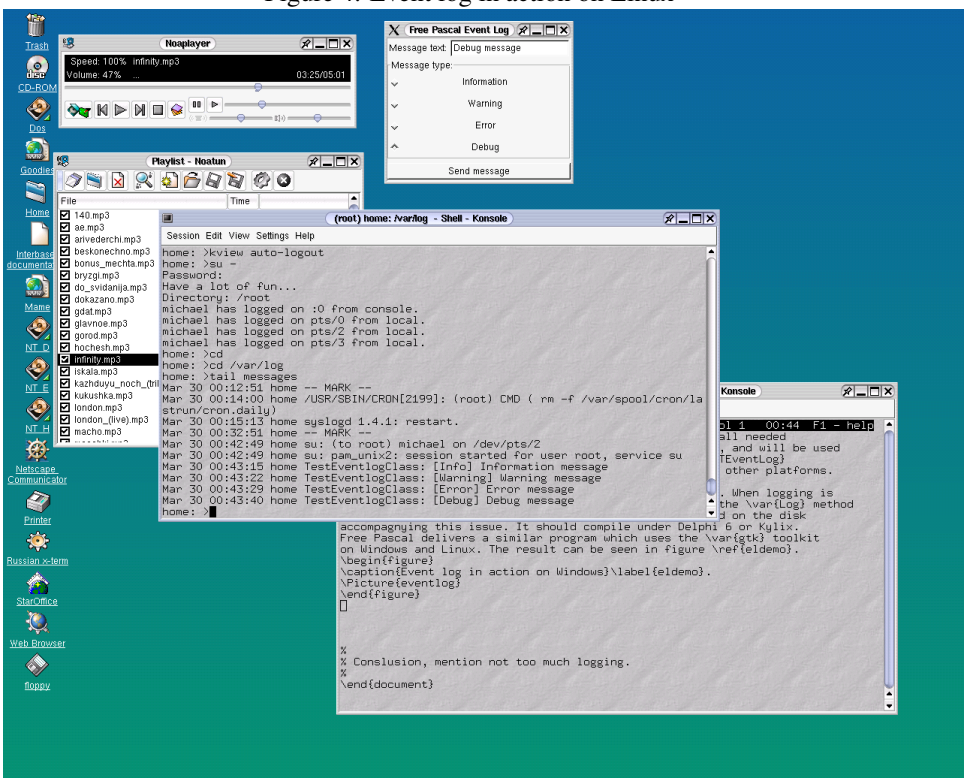


Figure 4: Event log in action on Linux



6 Conclusion

While not all details of the sytem log on Linux and Windows have been discussed - the configuration has not been dealt with, and the Windows API to retrieve log messages from the log files has also been left untouched - the article has shown that using the system log doesn't have to be difficult. Indeed, the API is quite simple and resemblant on all platforms. The proposed component is amply sufficient for simple logging of messages, for instance to report failure of services, or successful execution of scheduled tasks. However, the ease of use offered by the component should not be abused: Having too much log messages make the log difficult to read and decipher, which may hide the important messages which should be noted in the first place.