NAME

selog − selective logging

LIBRARY

Selective logging core library (libselog, −lselog)
Replacement err/warn implementation (libselog_err, −lselog_err)
Replacement syslog implementation (libselog_syslog, −lselog_syslog)

SYNOPSIS

#include <stdarg.h>
#include <stdbool.h>
#include <stddef.h>

#include <selog.h>

typedef struct selog_selector selog_selector**[1];**
typedef struct selog_buffer selog_buffer**[1];**

/ initialization */*
selog_selector *sel* = SELINIT(*name*, *level*);
int selog_open(**const char ***_config_**, const char *const** _spelling_**[]);**

/ simple output */*
void selog(**selog_selector** *sel*, **const char *** *fmt*, **...);**
void selogerr(**selog_selector** *sel*, **const char *** *fmt*, **...);**

/ complex messages */*
bool selog_on(**selog_selector** *sel*);
void selog_bufinit(**selog_buffer** *buf*, **selog_selector** *sel*);
void selog_prep(**selog_buffer** *buf*, **selog_selector** *sel*);
void selog_add(**selog_buffer** *buf*, **const char *** *fmt*, **...);**
void selog_addv(**selog_buffer** *buf*, **const char *** *fmt*, **va_list** *ap*);
void selog_addopt(**selog_selector, selog_buffer, const char *** *fmt*, **...);**
void selog_write(**selog_buffer** *buf*);

/ selector inspection */*
const char ***selog_name(selog_selector** *sel*);
const char ***selog_level(selog_selector** *sel*);
unsigned int SELOG_LEVEL(**selog_selector** *sel*);
unsigned int SELOG_EXITVAL(**selog_selector** *sel*);
unsigned int selog_parse_level(**const char ***_str_**, size_t** _len_**);**

/ debug macros */*
void selassert(**selog_selector** *sel*, **bool** *pred*, **const char *** *fmt*, **...);**
void seltrace(**selog_selector** *sel*, **const char *** *fmt*, **...);**

DESCRIPTION

Selog is a library of routines that unifies error reporting, activity logging, and debug tracing. It allows programmers to give their users flexible control over which messages are written and where they are written to.

This manual describes the programmer's interface to selog, and the auxiliary libraries that can be used to replace the standard **err**(3) and **syslog**(3) interfacses with implementations based on selog. For an overview of selog, details of how users can configure it, and how the output channels behave, see **selog**(7).

SELECTORS

Message output using selog is controlled by message selectors, and optional parts of messages are controlled by option selectors. A message selector classifies the messages it controls according to a "category" and a "level" which are both defined by the programmer. Multiple selectors can have the same category or level. Option selectors are distinguished from message selectors by their level setting.

As well as their category and level, selectors also contain a word of enable flags that is filled in at run-time based on the user's configuration. For message selectors this determines which channels messages should be written to, if any. For option selectors it determines whether the option is on or off. Once a selector's enable word has been initialized, the run-time test to skip disabled messages is very

efficient, so it is reasonable to leave debugging messages compiled in to production code.

Finally, selectors also have a linked list pointer which is used to chain all the configured selectors together. When selog is reconfigured this chain is used to reset all the selectors.

Categories

A selector's category is a string that names the kind of messages controlled by the selector, or that names the optional message part it controls. The selog configuration uses category names to determine which selectors are enabled, so they should be meaningful to users.

For example, a server might have a "connection" selector for messages related to open and closed connections, and an "interface" option to determine if the server's IP address is logged as well as the client's on multi-homed machines.

A good name for message selectors is often based on the part of the program that emits the messages. This is especially true for debugging messages.

Levels

A selector's level defines whether it is an option selector or a message selector, and for message selectors defines whether it is for error reporting, activity logging, or debugging. The possible values are as follows, in order from lowest to highest.

SELOG_TRACE
: For programer-oriented debug messages, i.e. for people working on the program's source code. The **seltrace()** function is intended for use with *SELOG_TRACE* selectors.

SELOG_DEBUG
: For user-oriented debug messages, i.e. for people dealing with configuration problems.

SELOG_OPTION_OFF or **SELOG_OPTION**
: For option selectors that are off by default.

SELOG_VERBOSE
: For activity log messages that are not emitted by default. Otherwise equivalent to *SELOG_INFO*.

SELOG_OPTION_ON or **SELOG_DEFAULT**
: For option selectors that are on by default.

SELOG_INFO
: The normal level for activity log messages.

SELOG_NOTICE
: For more important activity log messages. Not normally used, unless the user might want different messages in the same category to be written to different output channels. Consider using different categories instead.

SELOG_WARNING or **SELOG_WARN**
: For errors that should be fixed but which do not harm functionality.

SELOG_ERROR
: For errors that cause degraded functionality.

SELOG_CRITICAL or **SELOG_CRIT**
: For serious errors. The difference between *SELOG_ERROR* and *SELOG_CRITICAL* is similar to the difference between *SELOG_INFO* and *SELOG_NOTICE*.

SELOG_ALERT
: For syslog compatibility; not normally used. Intended to bring something to the immediate notice of the system administrator.

SELOG_EMERGENCY or **SELOG_EMERG**
: For syslog compatibility; not normally used. Intended for use when the system is catastrophically broken, so should be reserved for fundamental code like the kernel.

SELOG_FATAL(*status*)
: Similar to *SELOG_CRIT* except that after the message is written the *status* code is passed to **exit**(3). Selog does not implement its own exit hook: you should use **atexit**(3).

SELOG_EXIT
Equivalent to *SELOG_FATAL(0)*.

SELOG_ABORT
For failed internal consistency checks. Selog calls **abort**(3) after writing the message. The **selassert()** function is intended for use with *SELOG_ABORT* selectors.

Selog's levels are a superset of **syslog**(3)'s severities. The extensions mostly add more flexibility for non-error conditions. Syslog is rather over-endowed with error levels. Whether an error is severe enough to merit a "alert" or "emergency" level depends more on the purpose of the system and the policy preferred by the system administrator than on static properties of a program. Selog allows the sysadmin to choose how to handle messages based on the program and category, instead of the facility and severity as with syslog. Selog's greater flexibility allows sysadmins to express their filtering policy with fewer levels than syslog.

Defining selectors

Selectors are typically defined as static variables with file scope. (It is possible to define selectors dynamically, though you should be careful to re-use selectors and avoid creating and initializing new selectors if performance matters.) The *SELINIT()* macro is provided to initialize selectors correctly. The first parameter is the category and the second is the level. For example,

selog_selector log_conn **= SELINIT("***connection***",** SELOG_INFO**);**
selog_selector opt_iface **= SELINIT("***interface***",** SELOG_OPTION**);**

The macro initializer hides the detail that *selog_selector* is an array of one struct. This trick allows you to pass it by reference to functions without an explicit *&* address-of operator, similar to the standard *jmp_buf* type. In C++, *selog_selector* is a normal structure type, and C++ features are used to implement implicit pass-by-reference.

Software that creates selectors dynamically (such as selog's Lua interface) can turn a string into a numerical level using **selog_parse_level()**. The first argument is a pointer to the string and the second is its length. The string does not have to be nul-terminated. The function returns *SELOG_NOLEVEL* if the string is not a valid level.

Accessor functions

The following can be used to inspect a selector at run time.

selog_name(*sel***)**
Function that returns the selector's category.

selog_level(*sel***)**
Function that returns the selector's level as a string, suitable for use in log message preambles. See *selog_bufinit()* and *selog_prep()* below.

SELOG_LEVEL(*sel***)**
Macro that returns the selector's numeric level.

SELOG_EXITVAL(*sel***)**
Macro that returns the exit status of a *SELOG_FATAL()* selector.

selog_on(*sel***)**
Returns true if the selector is enabled. Has the side-effect of initializing the selector's flag word if necessary. Implemented as both a macro and a function; the macro may evaluate *sel* more than once. In C++ you can treat *sel* as a boolean value which has the same effect as calling *selog_on()*.

SIMPLE MESSAGE OUTPUT

This section describes the functions that allow you to write messages that are simple enough to format in one step, similar to the **printf**(3) and **syslog**(3) functions. These functions are defined with macro wrappers that perform the *selog_on()* test in-line for speed, therefore they may evaluate the selector more than once.

selog(*sel***,** *fmt***, ...)**
The usual message output function. Checks that the selector is enabled using *selog_on(sel)* and if so writes the formatted message to the relevant channel(s). The format string is interpreted the same way as by **printf**(3).

selogerr(*sel***,** *fmt***, ...)**
Equivalent to *selog()* with *": "* and *strerror(errno)* appended to the message.

seltrace(*sel***,** *fmt***, ...);**
The same as *selog(),* except with an extended preamble of the form
"*file***:***line func***() *category level***: "**
This macro is intended for use with *SELOG_TRACE* selectors. It does not have a function equivalent.

selassert(*sel***,** *pred***,** *fmt***, ...);**
If the predicate is false, *selassert()* writes the message to the relevant channel(s) and calls **abort**(3). The message preamble includes trace information and the stringified predicate expression. This macro is intended for use with *SELOG_ABORT* selectors. It does not have a function equivalent.

COMPLEX MESSAGES

This section describes functions that are used to compose messages in stages. For example, it is often awkward to format a message in one step if it contains optional parts. These facilities are also useful if formatting a message requires extra work that should be skipped if its selector is disabled. The pattern to follow is:

- Check that the message's selector is enabled using *selog_on().*
- Initialize a *selog_buffer* variable using *selog_prep()* or *selog_bufinit().*
- Call *selog_add()* or one of its related functions to append each part of the message to the buffer.
- When the message is complete, call *selog_write()* which writes it to the appropriate channel(s).

For example, *selogerr(sel, fmt, ...)* is equivalent to

```
if (selog_on(sel)) {
        selog_buffer buf;
        selog_prep(buf, sel);
        selog_add(buf, fmt, ...);
        selog_add(buf, ": %s", strerror(errno));
        selog_write(buf);
}
```

Like the *selog_selector* type, the *selog_buffer* type is an array of one struct. This trick allows you to pass it by reference to functions without an explicit *&* address-of operator. In C++, *selog_buffer* is a normal structure type, and C++ features are used to implement implicit pass-by-reference.

selog_prep(*buf***,** *sel***);**
The normal buffer initialization function. It is equivalent to:

```
selog_bufinit(buf, sel);
selog_add(buf, "%s ", selog_name(sel));
selog_add(buf, "%s: ", selog_level(sel));
```

selog_bufinit(*buf***,** *sel***);**
Initialize the buffer without adding any message text. This function does not have a built-in *selog_on()* guard; the selector argument is stored in the buffer for use by *selog_write().*

If you use *selog_bufinit(),* you should add your own message preamble in the style of *selog_prep().* This might be in order to add extra metadata such as the name of the function that emitted the message, or less metadata if the selector's category and level are redundant. The information should be ordered from less specific to more specific, for example, see *seltrace()* above, and the "MESSAGE FORMAT" section of **selog**(7). The preamble should not include information that is added by channels, such as the timestamp, host name, program name, etc.

selog_add(*buf***,** *fmt***, ...);**
Append the formatted string to the buffer.

selog_addv(*buf*, *fmt*, *ap*);
: Append the formatted string to the buffer, getting the arguments from a *va_list* like **vsnprintf**(3).

selog_addopt(*sel*, *buf*, *fmt*, ...);
: Add an optional part of a message to the buffer, if the selector is enabled.

selog_write(*buf*);
: Write the message to the relevant channel(s), determined by the selector that was passed to *selog_prep()*.

INITIALIZATION

Selog should be initialized soon after the program starts by calling **selog_open**(*config*, *spelling*);

The configuration string should be obtained from the user by a command-line option or a configuration file setting. If the user does not provide a configuration then the program may wish to provide a default to override the built-in default described in **selog**(7). If the program calls another selog function before *selog_open()*, then selog will initialize itself with its built-in default. In any case, if the **SELOG_CONFIG** environment variable is set, it overrides any other configuration string. Selog keeps a pointer to the configuration for later use by *selog_on()* to initialize selectors. (This implies that the program must not free or overwrite the memory pointed to by *config*.)

The *selog_open()* function scans the configuration string and opens the channels it specifies. It also checks that all the categories mentioned in the string are listed in the *spelling* array. You should ensure that the array contains selog's built-in categories as well as all the categories defined by your program. You can disable this check by passing a NULL *spelling* pointer.

Selog can be reconfigured by calling *selog_open()* again. The old channels are closed and all selectors are reset before the new configuration is installed. Therefore reconfiguration is not seamless. Note also that any *SELOG_CONFIG* environment variable setting still takes precedence.

The return value of *selog_open()* is 0 on success. If it encounters an error it sets *errno* and returns -1, and it also reports the error using the **log_config** selector. Because selog is not fully initialized at this point, the messages it controls can only be written to the standard error stream. However selog's filtering features do work.

STANDARD LIBRARY REPLACEMENTS

Selog comes with two auxiliary libraries that can be used to add selog's channel configuration features to programs that were not written to use selog. They can be used by re-linking the program with the relevant library, or less permanently by running the program with **LD_PRELOAD** set to the library's file name. In either case selog's replacement implementation of the functions "interposes" on the standard C library's implementation. Programs manipulated in this way do not call *selog_open()* so you must specify non-default configuration using the **SELOG_CONFIG** environment variable.

These libraries may also be useful for programs that mostly use selog, but which also depend on other code that calls the legacy APIs.

selog_err
: This is a replacement implementation of the 4.4BSD **err**(3) functions. It defines two selectors, **{err, FATAL}** used by the *err()* functions, and **{warn, ERROR}** used by the *warn()* functions. (Somewhat confusingly, the selog levels that have similar names to the functions have different meanings from the levels that correspond to the functions' actions.) If the program calls *err_set_file()* then the library just emits a diagnostic using the selector **{err_set_file, DEBUG}**. The library does not call *selog_open()* so relies on selog's default configuration.

selog_syslog
: This is a replacement implementation of the traditional **syslog**(3) functions. It defines eight selectors corresponding to the syslog severity levels, **{syslog, DEBUG}** up to **{syslog, EMERG}**. The selector is determined by the first *pri* argument to *syslog()*. The library does not do anything with facilities encoded in the *pri* argument of *syslog()* or with calls to *setlogmask()* and if either occurs the library emits a diagnostic using its **{syslog, DEBUG}** selector. The library implements the *ident* and *facility* arguments to **openlog()** and the **LOG_PID**, **LOG_PERROR**, and **LOG_CONS** options (though these can be overridden by the **SELOG_CONFIG** environment variable). It behaves as if **LOG_NDELAY** is always set.

DIAGNOSTICS

This section lists the built-in selectors used by selog itself. Selectors are written **{***category***,** *LEVEL***}** which is an abbreviated form of the usual selector initializer **SELINIT(“***category***”, SELOG_***LEVEL***);**

When documenting a program that uses selog, you should list the slectors it defines in a similar manner, and direct users to **selog**(7) for documentation of the configuration syntax.

{log_config, ERROR}
: This is used by **selog_open()** to report syntax errors. Because it is used before selog is fully initialized, the messages it controls can only be written to the standard error stream. However selog’s filtering features do work.

{error, FATAL}
{warning, ERROR}
{err_set_file, DEBUG}
: These are used by the **selog_err** library (see above).

{syslog, DEBUG}
{syslog, INFO}
{syslog, NOTICE}
{syslog, WARNING}
{syslog, ERROR}
{syslog, CRITICAL}
{syslog, ALERT}
{syslog, EMERGENCY}
: These are used by the **selog_syslog** library (see above).

ENVIRONMENT

SELOG_CONFIG
: Overrides the configuration string provided to *selog_open().*

EXIT STATUS

When the program uses a selector that is initialized using the **SELOG_FATAL(***status***)** macro, selog exits the program with the given status code.

COMPATIBILITY

The selog library requires some C99 features, in particular variadic macros. The <selog.h> header should work equally well in C99 and C++ programs.

SEE ALSO

abort(3), **atexit**(3), **err**(3), **exit**(3), **printf**(3), **selog**(7), **syslog**(3), **vsnprintf**(3).

AUTHOR

Written by Tony Finch <dot@dotat.at> <fanf2@cam.ac.uk>
at the University of Cambridge Computing Service.
Source available from <http://dotat.at/prog/selog>