This Document describes the Application Interface of Slog

DESIGN DOCUMENT

Document ID:

Slog

Slog API Guide

TABLE OF CONTENTS

1 Overview

This Slog module provides rich set of features for debugging/logging/tracing/error reporting.

It provides below list of features

- Traces the events module wise
- Almost 10 different trace levels for each module.
- Compile time and runtime enable/disable traces for every module & every level.
- Reduce foot print of target binary by removing debug prints directly from binary.
- Configurable support for Logging to console, Logging to buffer, etc…
- Timestamp all the events
- Origin (File path & line no) of the event
- Filtering of events based on the category & level.

2 Event Details

Events can be viewed as a structure with 3 attributes: Mask (or event category), level (or priority) and message. The events are represented by a 32 bit value, with the first 16 Bits representing the index in the memory chunk with respect to the charTab(concept explained in detail in Slog_designdocument). The lower 16 bits represent the mask value ored with the level.

While the diags bits selected in the 'mask' signify the "category" of the event (e.g. Entry/Exit, Analysis, Info), the 'level' field allows you to assign a "priority" or "detail level" to the event relative to other events in that category. There are four event levels defined by EventLevel.

Event Categories

Diags_ALL: Mask of all diagnostics categories, including both logging and asserts

Diags_ALL_LOGGING:Mask of all logging diagnostic categories (does not include asserts)

Diags_ANALYSIS:Analysis (e.g., benchmark) event

Diags_ASSERT: Assert checking

Diags_ENTRY: Function entry

Diags_EXIT: Function exit

Diags_INFO: Informational event

Diags_INTERNAL: Internal diagnostics

Diags_LIFECYCLE: Object life-cycle

Diags_STATUS: Warning or error event

Diags_USER1: User defined diagnostics

Diags_USER2: User defined diagnostics

Diags_USER3: User defined diagnostics

Diags_USER4: User defined diagnostics

Diags_USER5: User defined diagnostics

Diags_USER6: User defined diagnostics

Diags_USER7: Alias for informational event

Diags_USER8: Alias for analysis event

Event Levels

Four event levels are defined

Diags_Level1

Diags_Level2

Diags_Level3

Diags_Level4

3 Configurations

3.1 How to configure a logger?

1) Update the logger macro in the file Config.h file to the logger being used.
2) Update the config.c file to include appropriate logger header file.(LoggerBuf.h, LoggerSys.h)
3) Update all the application files to include appropriate logger header file and do extern LoggerX_Object loggerObj. (X is Buf/Sys)
4) If LoggerSys, *Uncomment* the statements

   LoggerSys_Object LoggerSys_Object__table__V = {1};
   LoggerSys_Object* loggerObj = (LoggerSys_Object*)&LoggerSys_Object__table__V;

   and *comment* the statements
   LoggerBuf_Object* loggerObj;
   LoggerBuf_Module_State LoggerBuf_state = {0x20,0,0};
   This statement sets the level of filter required. See Slog_Userguide for details
   LoggerBuf_Module_State* LoggerBuf_module = &LoggerBuf_state;
5) In case of loggerBuf, *comment* the statements

   LoggerSys_Object LoggerSys_Object__table__V = {1};
   LoggerSys_Object* loggerObj = (LoggerSys_Object*)&LoggerSys_Object__table__V;

   and *uncomment* the statements
   LoggerBuf_Object* loggerObj;
   LoggerBuf_Module_State LoggerBuf_state = {0x20,0,0};
   This statement sets the level of filter required. See Slog_Userguide for details
   LoggerBuf_Module_State* LoggerBuf_module = &LoggerBuf_state;
6) If LoggerBuf is used, following additional steps should also be followed
   a. Inside main, the first statement must be

      LoggerBuf_Instance_init();
   b. Moreover, it should end with

      LoggerBuf_Instance_finalize(loggerObj,0); to avoid any memory leaks.

3.2 How to configure the timestamp display format?

Update the TIMEDISPFORMAT to appropriate value from below.

Mode for printing the timestamp;

0 -> in Hexadecimal;

1 -> in Decimal;

2 -> in MilliSec

3 -> Human readable Date/Time format

3.3 How to configure the timestampproxy?

Slog provides two timestampproxies that can be attached to the logger viz. TimestampNull & TimestampStd.

If the macro TIMESTAMPPROXY is defined, TimestampStd is attached, TimestampNull otherwise.

3.4 How to remove the runtime control over a module's runtime mask?

Update the value of Diags_setMaskEnabled to 0, if the control is to be prevented,

Change to 1 otherwise.

3.5 How to enable the System_printf formats like %f, %$S & %$F?

In slog, supports for formats like %f, %$S & %$F can be disabled to save the code space. However, if required, they can be enabled by defining the macro PRINTEXTEND in Config.h file.

3.6 How to enable filtering of the events based on their level?

Update the value of LoggerBuf_filterByValue in Config.h file appropriately.

If want to disable, Set it to 0, else 1.

3.7 Can I disable the logger?

Change the value of Module__LOGDEF macro to 0 for disabling.

Update it to 1, if Logger is required.

3.8 What is Text_isLoaded macro in Config.h file?

In order to save memory footprint of the program, Slog allows the user to prevent the loading of event messages & module names to be loaded into the memory. This can be done by setting the value of Text_isLoaded macro to 0. If 1, normal working is expected.

3.9 How to use Log_write in the code?

In order to use Log_write, Log_error, Log_info, Log_warning, certain events are required to be defined before the call to any of the above API occurs. The createEvent API is responsible for the creation of events. So, the following lines should be included in the code before calling the above mentioned APIs.

Text_Error = createEvent("Error", Diags_STATUS, Diags_ERROR);

Log_L_error = createEvent("ERROR: %$F%$S", Diags_STATUS, Diags_ERROR);

Log_L_construct = createEvent("<-- construct: %p('%s')", Diags_LIFECYCLE, 0);

Log_L_create = createEvent("<-- create: %p('%s')", Diags_LIFECYCLE, 0);

Log_L_destruct = createEvent("--> destruct: (%p)", Diags_LIFECYCLE, 0);

Log_L_delete = createEvent("--> delete: (%p)", Diags_LIFECYCLE, 0);

Log_L_warning = createEvent("WARNING: %$F%$S", Diags_STATUS, Diags_WARNING);

Log_L_info = createEvent("%$F%$S", Diags_INFO, 0).

Log_put requires an event as its first argument. SO, custom events can also be created using the createEvent API and passed.

3.10 How to use Slog on Windows?

Build the Slog library using gmake command instead of make.

Change the rm keyword in bld.mak file in the <Slog installation directory>/src folder to del.

4 APIs

4.1 Log_print[X]

Void Log_printX(Diags_Mask mask, String fmt, …);

COMMENTS

Generate a Log "print event" with X arguments

ARGUMENTS

mask — enable bits and optional detail level for this event

fmt — a printf style format string

a1 — value for first format conversion character

a2 — value for second format conversion character

a3 — value for third format conversion character

a4 — value for fourth format conversion character

a5 — value for fifth format conversion character

a6 — value for sixth format conversion character

DETAILS

As a convenience to C (as well as assembly language) programmers, the Log module provides a variation of the ever-popular printf function. The print[0-6] functions generate a Log "print event" and route it to the current module's logger.

The number of values (a1, a2 etc.) passed to the Log_print depends on the value of X in Log_printX. E.g Log_print2(mask, fmt, a1, a2);

The arguments passed to print[0-6] may be characters, integers, strings, or pointers. However, because the declared type of the arguments is IArg, all pointer arguments must be cast to an IArg type. IArg is an integral type large enough to hold any pointer or an int. So, casting a pointer to an IArg does not cause any loss of information and C's normal integer conversions make the cast unnecessary for integral arguments.

The format string can use the following conversion characters. However, it is important to recall that all arguments referenced by these conversion characters have been converted to an IArg prior to conversion; so, the use of "length modifiers" should be avoided.

Conversion Character	Description
%c	Character
%d	Signed integer
%u	Unsigned integer
%x	Unsigned hexadecimal integer
%o	Unsigned octal integer
%s	Character string
%p	Pointer
%f	Single precision floating point (float)

Format strings, while very convenient, are a well known source of portability problems: each format specification must precisely match the types of the arguments passed. Underlying "printf" functions use the format string to determine how far to advance through their argument list. For targets where pointer types and integers are the same size there are no problems. However, suppose a target's pointer type is larger than its integer type. In this case, because integer arguments are widened to be of type IArg, a

format specification of "%d" causes an underlying printf() implementation to read the extended part of the integer argument as part of the next argument(!).

To get around this problem and still allow the use of "natural" format specifications (e.g., %d and %x with optional width specifications)

See Ssystem_printf for complete details.

The %f format specifier is used to print a single precision float value. Note that %f assumes that sizeof(Float) <= sizeof(IArg). Most clients that interpret float values except that they are represented in IEEE 754 floating point format. Therefore, it is recommended that the float values are converted into that format prior to supplying the values to Log functions in cases where targets do not generate the float values in IEEE 754 floating point format by default.

The first argument to a Log_print call is the diags category to be associated with the event.

It is also possible to associate an event level with the event to enable filtering of events based on event level. Conceptually, it is best to regard the event level as completely separate from the event's diags category; however, the priority value actually occupies a part of the diags mask. For this reason, it is possible to specify an event level by ORing the level with the diags mask. For example, to print an INFO event of LEVEL2, you'd simply write: (Diags.INFO | Diags.LEVEL2)

Specifying an event level is optional. Log_print calls which do not specify a level will receive the highest priority by default.

4.2 Log_errorX (String fmt, …);

COMMENTS

Generate a Log "error event" with X arguments

ARGUMENTS

fmt — a reference to a constant error string / fmt string

a1 — value for an additional parameter

a2 — value for an additional parameter

a3 — value for an additional parameter

a4 — value for an additional parameter

a5 — value for an additional parameter

DETAILS

The number of values (a1, a2 etc.) passed to the Log_error depends on the value of X in Log_errorX (X = 0-5). E.g Log_error2(mask, fmt, a1, a2);

The Log_error APIs are intended to allow users to easily log error events in their code. Similar to the Log_print APIs, Log_error does not require that you define an event. You simply pass an informative error string which can optionally be formatted with additional arguments. The error is logged with the predefined event L_error.

Log_error prepends a string to the message which identifies it as an ERROR and specifies the filename and line number of the Log_error call site.

Users may provide additional information in the error event, such as a predefined error code or details of the error. These additional values will be used to format the string passed to Log_error.

Log_error does not use a variable length argument list-you must call the appropriate Log_errorX API based on the number of arguments.

SEE

For information about format strings, See Log_printX

EXAMPLES

The following example demonstrates a typical usage.

```
Int myArg;
Log_error1("Invalid argument: %d", myArg);
```

The above event is formatted as, for example:

```
ERROR: "MyCode.c", line 35: Invalid argument: -1
```

4.3 Void Log_infoX(String fmt, …);

COMMENTS

Generate a Log "info event" with X arguments

ARGUMENTS

fmt — reference to a constant event string / fmt string

a1 — value for an additional parameter (e.g. an event code)

a2 — value for an additional parameter

a3 — value for an additional parameter

a4 — value for an additional parameter

a5 — value for an additional parameter

DETAILS

The number of values (a1, a2 etc.) passed to the Log_error depends on the value of X in Log_infoX (X = 0-5). E.g Log_info2(mask, fmt, a1, a2);

The Log_info APIs are provided for easily logging generic "informational" events with call site information. They are similar to the Log_print APIs in that they do not require you to define an event-- you simply pass an informative printf-style string which can optionally be formatted with additional arguments. The info record is logged with the predefined event 'L_info'.

The Log_info APIs log the L_info event which uses the 'INFO' diags category. They do not allow you to specify an event priority.

Log_info prepends the filename and line number of the call site to the message.

SEE

For information about format strings, See Log_printX

EXAMPLES

The following example demonstrates a typical usage.

```
Int load;
Log_info1("Current load: %d", load);
```

The above event is formatted as, for example:

"MyCode.c", line 15: Current load: 25

4.4 Void Log_warningX(String fmt, …);

COMMENTS

Generate a Log "warning event" with X arguments

ARGUMENTS

fmt — reference to a constant warning string / fmt string

a1 — value for an additional parameter (e.g. a warning code)

a2 — value for an additional parameter

a3 — value for an additional parameter

a4 — value for an additional parameter

a5 — value for an additional parameter

DETAILS

The number of values (a1, a2 etc.) passed to the Log_warning depends on the value of X in Log_warningX (X = 0-5). E.g Log_warning2(mask, fmt, a1, a2).

The Log_warning APIs provide the same features as the Log_error APIs, but are used to specifically log "warning" events.

The Log_warning APIs are equivalent to the Log_error APIs except that they use the predefined L_warning event. Log_warning prepends a string to the message which identifies it as a WARNING and specifies the filename and line number of the Log_warning call site.

SEE

For information about format strings, See Log_printX

EXAMPLES

The following example demonstrates a typical usage.

```
Int myArg;
Log_warning1("Value may be too high: %d", myArg);
```

The above event is formatted as:

```
WARNING: "MyCode.c", line 50: Value may be too high: 4096
```

4.5 Void Log_putX(Log_Event evt, …);

COMMENTS

Unconditionally put the specified Log event

ARGUMENTS

evt — the Log event to put into the log

mid — module ID of the module putting the event

a1 — value for first format conversion character

a2 — value for second format conversion character

a3 — value for third format conversion character

a4 — value for fourth format conversion character

a5 — value for fifth format conversion character

a6 — value for sixth format conversion character

a7 — value for seventh format conversion character

a8 — value for eighth format conversion character

DETAILS

The number of values (a1, a2 etc.) passed to the Log_put depends on the value of X in Log_putX (X = 0, 1, 2, 4, 8). E.g Log_put2(mask, fmt, a1, a2);

This method unconditionally puts the specified Log_Event evt into the log. The Types_ModuleId mid should be the module ID of the module which is putting the event.

SEE

For information about format strings, See Log_printX

4.6 Void Log_writeX(Log_Event evt, …);

COMMENTS

Generate a Log event with X arguments

ARGUMENTS

evt — the Log event to write

a1 — value for first format conversion character

a2 — value for second format conversion character

a3 — value for third format conversion character

a4 — value for fourth format conversion character

a5 — value for fifth format conversion character

a6 — value for sixth format conversion character

a7 — value for seventh format conversion character

a8 — value for eighth format conversion character

DETAILS

The number of values (a1, a2 etc.) passed to the Log_write depends on the value of X in Log_writeX (X = 0-8). E.g Log_write2(mask, fmt, a1, a2);

If the mask in the specified Log event has any bit set which is also set in the current module's diagnostics mask, then this call to write will "raise" the given Log event.

4.7 Log_EventId Log_getEventId(Log_Event evt);

COMMENTS

Get event ID of the specified (encoded) event

ARGUMENTS

evt — the Log event encoding a mask and event ID

DETAILS

This method is used to compare "known" Log events with "raised" Types_Event.

RETURNS

event ID of the specified event

SEE

Types_getEventId

4.8 Diags_Mask Log_getMask(Log_Event evt);

COMMENTS

Get the Diags mask for the specified (encoded) event

ARGUMENTS

evt — the Log event encoding a mask and event ID

RETURNS

Diags mask for the specified event

4.9 Void Diags_setMask(String control);

COMMENTS

Set a module's diagnostics mask at runtime

ARGUMENTS

control — diagnostic mask control string

This control string defines one or more actions where each action consists of a module name, an operator character, and a list of bit specifiers. Use the % character as a wildcard to turn the module name into a prefix matching pattern for a set of modules. Multiple actions are separated with the ; character.

The control string has the following format:

<module[%]><op><bits>[;<module[%]><op><bits>]

Specify individual module names explicitly (e.g. Main), or match multiple modules using a prefix matching pattern specified with the % character (e.g. Mai%).

The operator is specified with a single character from the following table.

Operator	Description
+	Set only the specified bits (other bits preserved)
-	Clear only the specified bits (other bits preserved)
=	Assign the entire mask to the given value where the specified bits are set and all other bits are cleared

The bits are specified with a list of characters from the following table. Refer to the Mask Summary for a list of each bit of the diagnostics mask.

Control Character	Diagnostics Constant	Description
E	ENTRY	Function entry
X	EXIT	Function exit
L	LIFECYCLE	Object life-cycle
I	INTERNAL	Internal diagnostics
A	ASSERT	Assert checking
Z	ANALYSIS	Analysis event
F	INFO	Informational event
S	STATUS	Status (error, warning) event
1	USER1	User defined diagnostics
2	USER2	User defined diagnostics
3	USER3	User defined diagnostics
4	USER4	User defined diagnostics
5	USER5	User defined diagnostics
6	USER6	User defined diagnostics
7	USER7	User defined diagnostics
8	USER8	User defined diagnostics

DETAILS

Use the given control string to set or clear bits in a module's diagnostics mask. The control string defines one or more actions where each action modifies the diagnostics mask in one or more modules. Each action can either set, clear, or assign a module's diagnostics mask. To both set and clear bits in the same diagnostics mask requires two actions, or you can assign the entire

mask explicitly in one action. Each action can specify a given module or a set of modules using name prefix matching.

WARNING

Each bit of a module's diagnostics mask that is to be modified at runtime, must be configured to be runtime modifiable in the program's configuration script. Use either RUNTIME_OFF or RUNTIME_ON as the configuration value for the desired bit in the diagnostics mask. Finally, the following configuration parameter must have the values indicated (which are their default values):

- Text_isLoaded = true;

Note: any error that occurs during the parsing of the control string causes Diags_setmask() to return without processing the remainder of the control string.

SEE

Appendix for flag details

4.10 Bits32 createEvent(String msg, Bits16 mask, Bits16 level)

COMMENTS

Generate Events

ARGUMENTS

msg – The msg defines a printf style format string that defines how to render the arguments passed along the event in a Log_write call. For a description of the allowable format strings.

mask - The mask defines which bits in the module's diagnostics mask enable this Log event.

level - The 'level' defines the event level of the event.

DETAILS

As explained above, Events has 3 attributes. Mask, Level and a msg. This API creates events when called with the appropriate parameters. This function actually puts the string in the memory chunk with starting address as chartab. The function returns in Bits32 format with the upper 16 bits representing the index with respect to the chartab where the msg string is stored and the lower 16 display the mask ored with level.

In order to use APIs like Log_error, Log_warning, Log_info, some events need to be predefined. These and certain more events are needed to be defined before the actual application code starts. These events are the following:

Log_Event Log_L_construct: Lifecycle event posted when an instance is constructed
Log_Event Log_L_create: Lifecycle event posted when an instance is created
Log_Event Log_L_delete: Lifecycle event posted when an instance is deleted

Log_Event Log_L_destruct: Lifecycle event posted when an instance is destructed
Log_Event Log_L_error: Error event posted by Log_errorX API
This event is marked as a STATUS event and given the priority level of ERROR.
This event prints the Log call site (%$F) and a format string (%$S) which is recursively formatted with any addition arguments.
Log_Event Log_L_info: Info event posted by Log_infoX API
This event is marked as an INFO event. The event priority is not specified in the event definition. Rather, it is specified as an argument to the Log_infoX APIs.
This event prints the Log call site (%$F) and a format string (%$S) which is recursively formatted with any addition arguments.
Log_Event Log_L_warning: Warning event posted by Log_warningX API
This event is marked as a STATUS event and given the priority level of WARNING.
This event prints the Log call site (%$F) and a format string (%$S) which is recursively formatted with any addition arguments.

RETURNS

The function returns in Bits32 format with the upper 16 bits representing the index with respect to the chartab where the msg string is stored and the lower 16 display the mask ored with level.

SEE

charTab

4.11 Registry_Result Registry_addModule(Registry_Desc *desc, String modName);

COMMENTS

Add a runtime module to the registry with the specified name

ARGUMENTS

desc — non-*NULL* pointer to a {*Registry_Desc*} structure.

modName — non-*NULL* string name of the module being registered.

DETAILS

The *desc* parameter and the *modName* string provided must both be permanent since the *Registry* will maintain references to both of these.

RETURNS

Registry_addModule returns one of the following *Result* status values indicating success or the cause of failure:

- *SUCCESS*
- *ALREADY_ADDED*
- *ALL_IDS_USED* There are a total of 16,384 - 1 module ids available for use by *Registry*.

5 Version History

Revision Number	Date	Description
0.1	24-Feb-11	Initial draft
0.3	07-Mar-11	
1.0	28-Jul-11	Few more review comments addressed

«««« § »»»»