Extensible Logger And Tracer
(ELATE)

version 0.3.7

User's guide.

Author: Boris Vigman

revision: 1.4

Changes from revision 1.3

Added:

1. New macros for group/condition masking/unmasking added

Changed:

2. Extra info logging parameter redesigned

Changes from revision 1.2

Removed:

1. Appendix B: “Report Forwarding Unit Format”

Added:

3. New macro TRACELOG_COMPLETE

Changed:

1. Macro FLUSH_REPORTS is now obsolete and its additional use is discouraged. However it stays for compatibility with previous versions.

Table of contents

Introduction

ELATE originally were designed to provide one basic functionality – generation and forwarding of predefined structured reports from inside the program toward the outer world. Varying of structure of these reports allows to user to use this package for different purposes, like runtime logging of information, tracing of arbitrary chosen internal events and conditions and other similar tasks.
The next chapters will provide information that will let the package make clear all its capabilities.

Installation

The installation procedure consists of the next simple steps:

1. Download the package from Sourceforge.net site;
2. Unzip it; this will make the directory named 'elate' under which all relevant files are located;
3. If you use Microsoft Visual Studio – there are project and workspace files named 'elate_lib.dsp' and 'elate_lib.dsw' accordingly. If you are using another development environment – you should make your own makefiles or whatever needed; to assist in doing that there is a file named 'Filelist' that contains list of all source and header files needed for successful compilation.
4. After the package is ready for compilation with your environment – you may proceed with embedding of it into your program.

Starting with ELATE

Initialization

The first thing you are expected to do is to include main header file of ELATE into your source file. This is done by adding the line:

#include "elate.h"

The next step is ELATE initialization. To initiate ELATE you should write the next string:

TRACELOG_INIT;

Make sure, that this line is added **before** first call to ELATE's API. However, **additional** calls of this command may take place later also.

Target's linkage

ELATE forwards generated reports to the entities called **'output targets**'. There are no restrictions regarding the nature of output target – this may be UNIX' pipe, Windows' message queue, file, printer, serial connection, IPC object, socket, display etc. Currently there are 2 predefined targets: File target and Console target.
Next important term is **'output channel**'. Report generated by ELATE is not written directly to a target but into entity called **'proxy target**'. Proxy target receives the report and sends it to a bunch of object called **'output channels**'. Each channel has its number in range of 0..7[1]. To each channel we may arbitrary link one or more output targets. Report written to channel will be automatically written to all of targets linked to this channel (currently each channel may be linked with no more than 2

[1] This number can be different for a specific configuration of ELATE.

targets[2]). By default all channels are linked with so called Null target – writing to this target will cause absolutely no result.

Linkage API consists of next macros:

LINK_TARGET_FILE(chan, tgtNum) – linkage of File target
LINK_TARGET_CONSOLE(chan, tgtNum) – linkage of Console target
LINK_TARGET_NULL(chan) – linkage of NULL target

Where:
chan – means number of channel that target should be linked to;
tgtNum – means number of target (0 or 1) in the pool of targets of this kind[3].

RESET_channel(chan) – resets the given channel and replaces all targets in it with Null target.

There are two additional actions that should be applied to targets: target initialization and termination. This is done through the use of next macros:

OPEN_TARGET_FILE(fnamePtr, tgtNum) – opening of file target

Where:

fnamePtr – name of file where reports should be written
tgtNum – number of this target in the pool of available file targets (0 or 1)

CLOSE_TARGET_FILE(tgtNum) – closing of file target

Where:

tgtNum – number of this target in the pool of available file targets (0 or 1)

OPEN_TARGET_CONSOLE(tgtNum) – opening of console target with number *tgatNum* (0..1)

CLOSE_TARGET_CONSOLE(tgtNum) – closing of console target with number *tgtNum* (0..1)

NOTE! Closing/opening and linking/unlinking of targets are independent actions and may by performed in any order.

Additional useful macro is *SUBST_CHANNEL(oldchan, newchan).* This macro redirects the forwarded reports from *oldchan* to *newchan*. This macro does not affects work of the whole mechanism so it may be used anytime.

Management

ELATE forwards generated reports to the Proxy target using intermediate queue.

[2] This number can be different for a specific configuration of ELATE.
[3] Currently there are 2 independent File targets available and 2 Console targets available; however those quantities may different for a specific configuration of ELATE

To force the pulling of the reports from the queue use the FLUSH_REPORTS macro. Actually it may be put into separate task thus causing regular clearing of the queue.

Disabling/Enabling

ELATE by default is considered enabled. That means embedding of ELATE's commands into sources will give to the hosting program tracing/logging ability. There are 2 ways to avoid this:

1. There are two macros declared in the file 'include/globDefs.h': __LOG_ENABLED and __TRACE_ENABLED. Comment any of these macros or both to eliminate unwanted functionality of the compiled program.

2. The same effect may be achieved using macros __NO_ELOG, __NO_ETRACE, __NO_ELATE in compiler's command line; the first macro cancels logging functionality, the second – cancels tracing functionality, the third one cancels both. Worth to mention that cancelled commands are not included into sources and thus are not compiled.

3. The developer may arbitrary enable/disable selected targets, using macros: TGT_targetname_ENABLED.

For full ist of commands commands see Appendix A.

Tracing and Logging

The main actions that ELATE is supposed to do is generation and forwarding of structured reports. For this purpose there are lots of macros that may be generalized in the next way:

cc_tl_aa_rr(set of parameters)

Where:

cc (optional) is a prefix that designates forwarding to specific channel. It may take value ‘C’ or have no value at all – in that case report will be sent to all existing channels.

tl is a prefix that designates type of report in terms of tracing and logging and should have one of two values – TRACE or LOG.

aa – name of report.

rr (optional) is a suffix defining that this report will be related to a specific predefined module.

Thus macro in its minimal form should be written in the next way (example[4]):

LOG_XXX(ui16proc, ui8level, teId, mnamePtr, rptLocatorPtr, xDataPtr)

Where:

ui16proc – number of processor that originates this report. If system has only one processor, it is recommended to have this number set to 0, however this issue is not crucial;

ui8level – level of report; this number may help to user to classify generated reports according to some arbitrary criteria; anyway this parameter does not affect work of the whole mechanism.

teId – Traceable Entity id; the object describing the resource that report relates to. In this case that parameter designates the pointer to allocated memory block.

mnamePtr – pointer to a string describing the name of current module.

rptLocatorPtr – pointer to a string, describing location of this macro call in the source file. This may be line number, or routine name or anything else.

xDataPtr – see the Extra data parameter paragraph.

Modifications of the macro are:

***TRACE**_XXX(ui16proc, ui8level, teId, mnamePtr, rptLocatorPtr, xDataPtr)*

***C**_LOG_XXX(ui8chan, ui16proc, ui8level, teId, mnamePtr, rptLocatorPtr, xDataPtr)*

Where *ui8chan* is channel number

***C_TRACE**_XXX(ui8chan, ui16proc, ui8level, teId, mnamePtr, rptLocatorPtr, xDataPtr)*

Where *ui8chan* is channel number

NOTE! Macros prefixed with C_ may put macro TGT_BROADCAST as the first parameter (*ui8chan*). This will cause forwarding of this report to **all** defined channels. Thus C_-prefixed macro will do exactly the same job as non-prefixed one.

[4]full list of macros of this type may be found in files ‘private/logIfs.h’ and ‘private/traceIfs.h’

Additional modification of this macro has the next forms:

*TRACE_XXX_**RM**(ui16proc, ui8level, teId, rptLocatorPtr, xDataPtr)*

*C_TRACE_XXX_**RM**(ui8chan, ui16proc, ui8level, teId, rptLocatorPtr, xDataPtr)*

*LOG_XXX_**RM**(ui16proc, ui8level, teId, rptLocatorPtr, xDataPtr)*

*C_LOG_XXX_**RM**(ui8chan, ui16proc, ui8level, teId, rptLocatorPtr, xDataPtr)*

This type of macros allows to programmer not to specify the module name, thus letting define one module name for several files. This process will be detailed in chapter *Module Management*.

There is another set of macros having similar form:

LOG_VAR(ui16proc, ui8level, varPtr, ui16len)

LOG_ASSERT(ui16proc, ui8level, expr)

LOG_FREETEXT(ui16proc, ui8level, ui8teCond, textPtr)

C_LOG_VAR(ui8chan, ui16proc, ui8level, varPtr, ui16len)

C_LOG_ASSERT(ui8chan,ui16proc, ui8level, expr)

C_LOG_FREETEXT(ui8chan,ui16proc, ui8level, ui8teCond, textPtr)

TRACE _VAR(ui16proc, ui8level, varPtr, ui16len)

TRACE _ASSERT(ui16proc, ui8level, expr)

TRACE _FREETEXT(ui16proc, ui8level, ui8teCond, textPtr)

C_ TRACE _VAR(ui8chan, ui16proc, ui8level, varPtr, ui16len)

C_ TRACE _ASSERT(ui8chan, ui16proc, ui8level, expr)

C_TRACE_FREETEXT(ui8chan, ui16proc, ui8level, teCond, textPtr)

cc_tl_VAR macro in all its variations allows dumping of the specified variable.

Parameters are:

ui16proc – number of processor

ui8level – user-defined level of this report

varPtr – pointer to variable to be dumped

ui16len – length (in bytes) of this variable

NOTE! cc_tl_VAR macro dumps given variable as a memory region (byte-by-byte) thus byte order issue is irrelevant.

cc_tl_ASSERT macro in all its variations allows assertion of given expression; if assertion fails then the report is originated.

Parameters are:

ui16proc – number of processor

ui8level – user-defined level of this report

expr – evaluated expression

cc_tl_FREETEXT macro in all of its variations allows generation of text message.

Parameters are:

ui16proc – number of processor

ui8level – user-defined level of this report

teCond – constant defining the kind of this report; may have values of TE_FREETEXT_NEUTRAL, TE_FREETEXT_WARNING, TE_FREETEXT_PANIC

textPtr – pointer to a string to be forwarded.

Extra data parameter

The tracing/logging command may also include some additional information, like number, memory pointer, string etc. In order simplify and generalize this procedure set of macros is provided:

XTPAR_BYTE(x) – order to interprete parameter *x* as a byte/octet (8 bit)

XTPAR_SHORT(x) – order to interprete parameter *x* as a short integer (16 bit)

XTPAR_INT(x) – order to interprete parameter *x* as an integer number (platform-dependent)

XTPAR_LONG(x) – order to interprete parameter *x* as a long number (platform-dependent)

XTPAR_FLOAT(x) – order to interprete parameter *x* as a float number (platform-dependent)

XTPAR_DOUBLE(x) – order to interprete parameter *x* as a double number (platform-dependent)

XTPAR_ADDR(x) – order to interprete parameter *x* as a memory address (platform-dependent)

XTPAR_STRING(x) – order to interprete parameter *x* as a string

Examples:

```
   TRACE_MEMALLOC(0, 0, ds, "Module",routineName1,XTPAR_STRING
("memory alloc"));

   TRACE_MEMALLOC(0, 1, ds, "Module",routineName2,XTPAR_INT(sizeof
(struct DemoStruct)));

   TRACE_MEMALLOC(0, 2, ds, "Module",routineName3,XTPAR_FLOAT(sizeof
(struct DemoStruct)));
```

Addition of output targets

Addition of a new target is not pretty simple, so there is a tool for automatized creation of the targets. The Perl script named *'tgtmake.pl'* being run properly creates number of C source files with target-specific finction and data structure templates. The developer's obligation is to implement created buctions' bodies.
The process of target creation consists of 4 steps.
Step 1: run the script with list of required targets
example: '> tgtmake.pl tgt1 tgt2 tgt3'
This procedure will create 8 files:
nullTarget.h
tgt1.h
tgt2.h
tgt3.h
nullTarget.c
tgt1.c
tgt2.c
tgt3.c
Step 2: implement the empty functions, update the targets' data structures if needed.
Step 3: place these files to *'private/targets/'* directory and/or add them into makefile.
Step 4: in the *target.h* file add *#include* line for each target.

Addition of custom traceable entities

In addition to the built-in report types described in Tracing and Logging chapter the developer may add its own tracing/logging reports. This is achieved by using of '*temake.pl*' script located in '*tools'* directory. The script takes as argument the name of some text file containing description of desired reports. The script reads that file, processes each line and produces 5 new files organized into tree structure under new directory '*private*':

./private/te/teDefs.h; Traceable Entities Definitions file

./private/te/trEntities.h; Traceable Entities header file

./private/te/trEntities.c; Traceable Entities implementation file

./private/logIfs.h; Logging Interfaces header file

./private/traceIfs.h; Tracing Interfaces header file

All files must be copied to the corresponding directories where they may be compiled. Last 2 files contains new interfaces that may now be used by developer for generation of custom reports.

Creation of description file.

The description file contains sequence of tokens representing the traceable entities groups. Each token is a word constructed of symbols 'a' to 'z' (case is not relevant), 0 to 9 and '_'. The symbole '#' is used for comments, all symbols follwing it in the current line will be ignored. Each token must be followed by ';' otherwise it will be ignored. Tabs and spaces are ignored too.

Groups are declared as next:

```
[Group-Name]; #brakets declares the new group beginning

item1;

item2;

item3;
```

Items, not preceded by a group name will be ignored.

Example: say the developer wants to add 2 new groups of reports: Window report and Timer report. Say he decided that most interesting aspects of window's existence are: Closing, Opening, Refreshing, Selection. In the same way he defined for the Timer 3 aspects: Creation, Expiration, Deletion.

The descriptive file will look like this:

[Window];

Close; #trace closing

Open; #trace opening

Refresh; #trace refreshing

Select; #trace selection

[Timer];

Create;

Expire;

Delete;

Files produced by script from the given descriptive file will contain tracing and logging interfaces corresponding to given groups.

Module management

Module for ELATE is nothing more than a simple string that contains the module name.

There are 2 ways to add module name to report: making it explicitly in the regular report macro or using special form of macro (having the suffix _RM).

ELATE gives to a developer an opportunity to create its own module names. The procedure is pretty simple:

1. Put the module names you want to use inside your program into text file (say, named 'modnames'; the actual filename is arbitrary chosen by the developer); use very simple notation consisting of 2 rules:

- write down desired module names as usual words using only symbols '_', 'a' to 'z' (both cases are legal) and 0 to 9;

- separate the words by semicolon ';' - words that are followed by semicolon will be silently ignored; spaces and tabs are also ignored; symbol '#' designates start of inline comment – this symbol and all subsequent ones in the same line will be ignored;

2. save file and run on it the script 'modmake.pl', i.e. Run the command: *modmake.pl modnames;*

3. after the script succesfully finished – two files will be produced under directory '*modules*': *modules.c* and *modules.h* . Copy this files into directory '*private*';

That's all. Now you may use the module names in your reports. To allow authomatical inclusion of module name into reports the developer MUST include the next string at the beginning of its code:

#define ROOT_MODULE modname

where the *modname* is one of words mentioned in the source text file. Once you have added string like this into beginning of your C/C++ source file the call for any RM-suffixed macro in this file will cause automatic addition of module name into generated reports. Macros without RM suffix will remain unaffected.

> Example: say there are three modlue names to be defined: *mod1, mod2, mod3;*
>
> in the text file they will be written as:
>
> mod1;
>
> mod2; #comment may be placed here
>
> mod3;

After running of '*modmake.pl*' script and copying of relevant files module names should be employed in each C file in the next way:

#define ROOT_MODULE mod1

or

#define ROOT_MODULE mod2

or

#define ROOT_MODULE mod3

Masking of reports

New feature available in ELATE since version 0.3 is **reports masking**.

Let us imagine the next situation: there are lots of reports generated by some application that uses ELATE package. Some of reports are critically important, some are less important, some are unneeded and/or irrelevant. Obviously there is a need to filter the ussued meesages somehow i.e suppress unneded reports. This may be done by redirection: messages from some output channel may be forwarded to other channel. However, such a filtering is based on one pretty wide criteria: channel number. Another way is to filter each message by more fine criteria applicable to a specific report. This kind of filtering is called 'masking' and currently is based on 4 independent properties: module name, group type, report type and report level.

Module-name based filtering.

The next set of commands covers all possible cases of use:

MASK_MODULE(ui16index) – macro; masks module designated by the given 16-bit number; after call of this macro any report related to this module name will be supressed i.e. will not be forwarded to any output target.

UNMASK_MODULE(ui16index) – macro; unmasks module designated by the given 16-bit number; supression is cancelled.

Next 2 functions provides interfaces for getting info regarding the database:

unsigned short moduleGetDBSize() - returns the number of items inside the modules names database;

*char * moduleGetByIndex(unsigned short index)* – returns the pointer to item designated by the given index; if no pointer can be found – 0 is returned.

NOTE! All module names are inside the dedicated database. They are inserted automatically at the time of script running at the order of coming; the first item is inserted with index 0 (zero), the second one takes index 1 and so on; for precise correlation between indices and actual module names look at generated *'modules.c'* file.

Level based filtering.

The other filtering criteria is report **level.** Level is expressed by arbitrary 8-bit number in range of 0..255. 0 is the highest level. The next commands are available for level-based masking:

MASK_LEVEL(ui8level) – macro; masks all reports with level lower or equal to *ui8level*. Example: MASK_LEVEL(0) will suppress all reports issued after call of this macro. Another example: MASK_LEVEL(3) will suppress all reports issued after call of this macro with level lower or equal to 3. Reports with levels 0,1,2 will not be suppressed.

UNMASK_ALL_LEVELS – macro; unmasks all reports previously level-masked reports; has the same effect as MASK_LEVEL(255) .

Group based filtering.

Whole **condition group** filtering . The next commands are available for group-based masking:

MASK_GROUP(ui8grp) – macro. Mask**s a**ll reports belonging to agiven group.The reports will be supressed with no respect to the masking level if any was set before. Example: MASK_GROUP(128) will suppress all reports belonging to a group with index 128.

UNMASK_GROUP – macro; unmasks all reports belonging to a given group.

Condition based filtering.

Separate condition filtering . The next commands are available for condition-based masking:

MASK_CONDITION(ui8grp, ui8condition) – macro. Masks the reports with specified confition inside the specified group. The reports will be supressed with no respect to the masking level if any was set before. Example: MASK_CONDITION (128, 1) will suppress report with condition number 1 inside the group number 128.

UNMASK_CONDITION(ui8grp, ui8condition) – macro; unmasks the report with given condition belonging to a given group.

NOTE! All traceable entities are masked by default. You MUST to unmask each entity or group of entities that you want to be issued. Remember, that each entity including the pre-defined ones belongs to some group, i.e. there is no entity that belongs to no group.

User commands

Masking process may be controlled statically (by including masking commands into compiled code) and dynamically as well. For dynamic masking user may issue special commands which will be sent to the agent from the outer world. The agent will receive them, parse somehow and execute.

To provide such functionality there are existing several tools:

1. macro TRACELOG_COMPLETE – runs all waiting activities, among them the arrived and ready-for-execution user commands;

2. function *hookUsrCommandRun(),* located in the *'private/usrCommand.c'* file; the mplementation of this function is provided by user; the function is supposed to run the arrived commands, calling for defined masking interface.

3. Functions hook*UsrCommandInit()* and *hookUsrCommandKill()* performs initialization/termination functionalities respectively. Their implementation also must be provided by user.

NOTE! There is no synchronization mechanism provided. However, since *hookUsrCommandRun()* is expected to alter some internal global variables the implementation MUST take the synchronization issue into account and provide one if needed.

ELATE vs specific system

ELATE intended to be OS independent SW. However, sometimes it is impossible to ignore capabilities of specific environment. For example, one of most prominent guidelines in ELATE's design process was presentation of maximally detailed information regarding current report. As a result, all reports generated by the program carry OS specific info like timestamp and identification of the context in which the report was generated. To adopt this data to definitions of specific OS there are several steps that should be taken:

1. In the file 'private/os/elateOS.c' function *hookGetTimestamp()* should be implemented to return credible timestamp value;

2. In the same file function *hookGetContextId()* should be implemented to return actual context identification.

Appendix A: List of Commands

Targets Management

LINK_TARGET_FILE(chan, tgtNum) – linkage of File target

LINK_TARGET_CONSOLE(chan, tgtNum) – linkage of Console target

LINK_TARGET_NULL(chan) – linkage of NULL target

Parameters:

chan – means number of channel that target should be linked to;

tgtNum – means number of target (0 or 1) in the pool of targets of this kind.

OPEN_TARGET_FILE(fnamePtr, tgtNum) – opening of file target

Parameters:

fnamePtr – name of file where reports should be written

tgtNum – number of this target in the pool of available file targets (0 or 1)

CLOSE_TARGET_FILE(tgtNum) – closing of file target

OPEN_TARGET_CONSOLE(tgtNum) – opening of console target with number *tgatNum* (0..1)

CLOSE_TARGET_CONSOLE(tgtNum) – closing of console target with number *tgtNum* (0..1)

Parameters:

tgtNum – number of this target in the pool of available file targets (0 or 1)

Channels Management

RESET_channel(chan) – resets the given channel and replaces all targets in it with Null target.

SUBST_CHANNEL(ui8chanOld, ui8chanNew) – substitutes a channel by a new one.

Masking management

UNMASK_ALL_LEVELS

MASK_MODULE(ui16index)

UNMASK_MODULE(ui16index)

MASK_GROUP(ui8grp)

UNMASK_GROUP(ui8grp)

MASK_CONDITION(ui8grp, ui8cond)

UNMASK_CONDITION(ui8grp, ui8cond)

Tracing enabling/disabling

Compilationa flags:

__NO_ELOG – disable logging

__NO_ETRACE – disable tracing

__NO_ELATE – disable ELATE

__LOG_ENABLED – enabling the logging

__TRACE_ENABLED – enabling the tracing

__TGT_X_ENABLED – enable the X target

__TGT_SERIAL_ENABLED – enable the Serial target

__TGT_FILE_ENABLED – enable the File target

__TGT_CONSOLE_ENABLED – enable the Console target

__TGT_IPC_ENABLED – enable the IPC target

Built-in tracing/logging macros

LOG_VAR(ui16proc, ui8level, varPtr, ui16len)

C_LOG_VAR(ui8chan, ui16proc, ui8level, varPtr, ui16len)

TRACE _VAR(ui16proc, ui8level, varPtr, ui16len)

C_ TRACE _VAR(ui8chan, ui16proc, ui8level, varPtr, ui16len)

Parameters:

ui8chan – channel where reorts should be forwarded to; constant TGT_BROADCAST will allow forwarding to all existing targets.

ui16proc – number of processor

ui8level – user-defined level of this report

varPtr – pointer to variable to be dumped

ui16len – length (in bytes) of this variable

LOG_ASSERT(ui16proc, ui8level, expr)

C_LOG_ASSERT(ui8chan,ui16proc, ui8level, expr)

TRACE _ASSERT(ui16proc, ui8level, expr)

C_ TRACE _ASSERT(ui8chan, ui16proc, ui8level, expr)

Parameters:

ui8chan – channel where reorts should be forwarded to; constant TGT_BROADCAST will allow forwarding to all existing targets.

ui16proc – number of processor

ui8level – user-defined level of this report

expr – evaluated expression

LOG_FREETEXT(ui16proc, ui8level, ui8teCond, textPtr)

C_LOG_FREETEXT(ui8chan,ui16proc, ui8level, ui8teCond, textPtr)

TRACE _FREETEXT(ui16proc, ui8level, ui8teCond, textPtr)

C_TRACE_FREETEXT(ui8chan, ui16proc, ui8level, teCond, textPtr)

Parameters:

ui8chan – channel where reorts should be forwarded to; constant TGT_BROADCAST will allow forwarding to all existing targets.

ui16proc – number of processor

ui8level – user-defined level of this report

teCond – constant defining the kind of this report; may have values of TE_FREETEXT_NEUTRAL, TE_FREETEXT_WARNING, TE_FREETEXT_PANIC

textPtr – pointer to a string to be forwarded.

General management

TRACELOG_COMPLETE -– performs all waiting activities, like running of arrived user command, flushing buffers and so on

TRACELOG_INIT – initiates the logging mechanism

TRACELOG_KILL - terminates the logging mechanism

FLUSH_REPORTS – OBSOLETE; additional use is discouraged; stays for compatibility ONLY