# A Log-Tool Suite for Embedded Systems

Harald Schuster, Martin Horauer, Michael Kramer
University of Applied Sciences Technikum Wien
Vienna, Austria
e-mail: {schuster, horauer, kramer}@technikum-wien.at

Heinz Liebhart, Josef Büger
Kapsch TrafficCom AG
Vienna, Austria
e-mail: {heinz.liebhart, josef.bueger}@kapsch.net

***Abstract*—Logging is a common method to monitor the operation of a system and to identify failures of services and system components. When developing software for an embedded system there are stricter restrictions that result from the limited hardware resources. The performance of the hardware as well as the available memory is limited. Furthermore, embedded systems are often not accessible for a long period of time. This paper presents tools to improve existing logging methods without compromising existing development flows. Essentially, these tools on one hand transparently pre-process the source code in order to optimize logging instructions and add on the other hand a post-processing step when analyzing log-files. The presented tools were evaluated by integrating them to an industrial project where they show a significant improvement of the logging performance.**

***Keywords*—*Logging; Embedded Systems; Memory Limitation; Log Analysis.***

## I. INTRODUCTION

Electronics paired with software is a key enabler for many modern products and systems. For example, it allows to add new features and functionalities, improve reliability, safety, environmental efficiency, or comfort. At the same time the complexity of these systems is steadily increasing mandating more efforts and new approaches for verification. In practice, various processes and approaches are in use — very common are static analyses, code reviews paired with testing and sometimes even formal methods. For embedded systems this problem is aggravated since usually different environmental conditions, limited controlability, and observability need to be taken into consideration. Thus, rigorous verification is a challenging task that is often limited by economical aspects.

In the field of embedded systems, Embedded Linux gained momentum in recent years due to plummeting costs of available hardware resources, the maturity and the feature rich functionality of the open-source code-base. Here, as with desktop operating systems, logging is a common approach to identify problems of an operational system. Services and applications can write information, warnings and encountered errors to log-files in a chronological fashion. Analyzing these logs can help to identify problems and hint to their sources. Especially when dealing with long-running server applications logging is a common practice. For example, [1] observed that on average common server programs provide one logging instruction every 30 code lines and that 18% of all committed revisions are due to modifications of the respective logging messages. In contrast, to desktop and server systems, memory is still a limiting and costly factor for embedded systems. Hence, for such systems a trade-off between the time a log-file is accumulated, the number of log messages, their frequency and verbosity is important. On one hand, lowering these values is necessary when the amount of available memory is limited – this will typically reduce the expressiveness and usefulness of the logs. On the other hand, frequent and verbose log-messages will also impair on the available processing power — the latter being another limited factor.

The main contribution of this paper is to present some tools and a flow that improves the efficiency of existing logging practices. In particular, it optimizes the length, verbosity and performance of the logs without impairing the existing log facilities and requires only a marginal adaption of the development flow. We evaluate our approach using an industrial strength Embedded Linux device that is executing a communication stack used for road-pricing systems.

The remainder of this paper is structured as follows: Section II provides an overview of existing tools and approaches to improve the performance of logging on embedded systems. Then we present our modified tool flow before we provide some implementation details of our LogEnhancer and the LogAnalyzer tools. Afterwards, we describe how we evaluated our tools before we conclude the paper giving an outlook on future directions.

## II. RELATED WORK

Logging is a common approach to identify problems in a computing solution [2][3][4]. When looking at Linux, for example, multiple different logging methods are available. In the majority of these cases, however, these methods are designed for standard desktop or server systems. Widespread in use are `syslog` [5] and their forks `rsyslog` and `syslog-ng`. Another common logging method is `Journal` which is included in the system and service manager `Systemd`. All these systems, however, are not optimized for embedded targets.

For these systems dedicated logging methods exist that improve several aspects when compared to the “standard” methods, cf. [6]. For example, Amontamavut et al. [7] describes a logging mechanism that is optimized for embedded systems with small memory. Their approach is to split the logging mechanism into two parts. One part is executed on the embedded device and sends reduced log messages via Ethernet to a server. The second part is the server and consists of different monitoring and debugging tools. One of these tools is a log analyzer to generate readable log messages. Overall, this approach reduces the log-file size and improves the log performance, however, it requires an active network connection to a remote host when the system is operational; a requirement that rules this approach out for our intended field of application.

Jeong et al. [8] describe a logging mechanism to improve the message throughput and to speed up the latency. They achieved their goals by avoiding the message transfer between the user and the kernel space. The message is stored in a shared memory and forwarded to flash memory or a remote host by a message collector. The developer can use the normal logging API. This changes increases the message throughput but does not affect the log-file size.

## III. Approach

The motivation of our work is to improve the logging functionality of an Embedded Linux device. In particular our approach intends to optimize memory usage and performance without impairing on the existing development process, in particular, how log messages are coded. To that end we first describe existing approaches and development flows used by our industrial partner before we detail our new concept and implementation.

### A. Status Quo

We implemented and evaluated our approach on code provided by our industrial partner used for embedded devices in the field of road pricing around the globe. In fact, there is already a very large base of devices running in the field; for instance, most commercial road vehicles in the European Union are equipped with these devices. Hence, similar to automotive electronics even small savings have a significant impact on the costs thus simply expanding available resources (computing power, memory size) is critical. Here, next to elaborate testing, logging and off-line analysis of the log-files is a common approach to tackle runtime mis-behavior.

The target employs OpenWRT a Linux distribution for embedded devices as operating system. The application mostly uses C code along with CMake as build system. The employed logging method employs macros to emit log-messages and define statements to set the logging level. The logging levels are ERROR, WARNING, NORMAL, INFORMATION, DEBUG and TRACE. Furthermore, using logging masks it is possible to filter the kind of messages that are logged (e.g., communication related log-messages, layer 2 messages, layer 3 messages).

The source-files consist of roughly 150k lines (counted using the tool cloc) of source-code targeting 3 different targets; Table I lists the log messages found therein separated by their logging level. In total, the source-files hold 2259 log messages; thats on average about one log message every 67 lines of code. Compared to several open-source projects listed in [1] this code contains roughly 2-3 times less log messages. One reason therefore, might be that the code at hand targets an embedded platform, whereas the listed references are aimed at server applications.

TABLE I. LOG MESSAGE STATISTICS

| LOG_ERR | LOG_WARN | LOG_INFO | LOG_DBG | LOG | LOG_HEX |
|---|---|---|---|---|---|
| 517 | 262 | 340 | 53 | 1085 | 2 |

Logging is implemented as follows: The developer uses a logging macro to insert log messages into the source code. Furthermore, he has to define the logging level (e.g., LOG_ERROR) and the logging mask (e.g., LM_COM). The logging level is set by a macro – there exists one for every level. The logging mask is the first parameter of the macro followed by the message and the arguments that should be

```
/* logging macro */
ret = 12;
LOG_ERR(LM_ALL, "layer2_init returned %d %s\n", ret,
     time_buffer);

/* logging macro resolved */
log_(LL_ERR, LM_ALL, "layer2_init returned %d %s\n",
     ret, time_buffer);

/* log message */
<E>LM_ALL: layer2_init returned 12 2014-02-16_13
    .52.14.012
```

Figure 1. The ‘present’ logging approach.

logged. The text string of the message is often put together using several ANSI/ISO C functions and finally output to *stdout*. In order to get a coherent logging style, the use of these macros is enforced by a strict development process using various approaches like code-reviews or static code analysis. Hence, our approach must comply with the same rules.

Figure 1 depicts an example of a typical logging macro and how it is resolved. The last line of this listing illustrates a log message entry produced by this code.

### B. Requirements

The requirements for our approach are based on the above analysis of the source code, the physical limitations of the embedded target as well the existing development flow. The detected requirements are as follows:

(i) File Size: A significant reduction of the file-size of the log-files shall have top most priority. This will make room for more and more verbose logging instructions and longer logging intervals before the logs get rotated.

(ii) Performance: Improve the performance or at least keep the penalty on the performance as low as possible. Note that this requirement rules out computing intense (run-time) compression approaches.

(iii) Tool Flow: Keep modifications to the existing toolflow and especially the way logs are generated as low as possible. The developers should not be affected by our approach requiring that the tools must be integrated in the standard build process in use.

(iv) Log Information: Enhance the log messages in a way so that the time and origin of the message is better traceable to locations in the source-code of the respective revision.

(v) Filtering: Allow for a better, more efficient filtering of relevant messages that led to erroneous log-entries. The aim here is to help in the investigation of causes that were leading to this message.

(vi) Revision Management: Typically many embedded devices are operational in the field using different revisions of the respective application. Thus proper bug tracking and revision management are essential in order to correlate the log-files with the correct revision of the source-code.

### C. Design Concept

Our design approach first runs as a pre-step to the build process and is as follows:

```
/* replaced logging macro */

log_ts_(LL_ERR, LM_ALL, "1 %d\n", ret);

/* log message */
1 12 <ts>
```

Figure 2. The ‘new’ logging approach.

```
<E>LM_ALL: layer2_init returned 12 [l2_state_machine
     @ layer2.c(368)]
```

Figure 3. Reconstructed log message.

(1) Enumerate found logging-commands in the source-code and store the messages in a separate decoder-file.
(2) Replace the logging commands with the enumerator from step (1) along with the values of variables found in the logging command separated by a white-space.
(3) Build the program and deploy the binary to the target system and set the latter into operation.

When the log-files are analyzed the following steps are performed:

(4) Retrieve the log-files from the host; therefore the target must be connected to the respective workstation.
(5) Reconstruct the original log-messages using the logged enumerator and variables along with the decoder-file.

By logging only a simple number instead of an entire text string the logging becomes more efficient and the log-files get smaller. Furthermore, the code on the embedded target becomes slightly more efficient as well, since we offload the bulk of the work (putting the log-commands together) to a host computer.

Figure 2 illustrates our new logging approaches produced by our LogEnhancer tool; the first code line shows the log command as replaced by our tools. The last line again shows the log message written to the log-file; note that for readability we replaced the binary time-stamp with `<ts>` in this example.

Filename, and source-line of the log-command that produced a log-entry is stored along with the enumerator in the decoder-file. Figure 3 shows the log-message that was reconstructed by our LogAnalyzer tool. Note that this log message additionally contains the name of the function, the file-name and the source-code line where the log-entry originated.

## IV. Tool Implementation

As described in Section III, the tool is split into two parts the LogEnhancer used prior to the build process and the LogAnalyzer used when log-messages get inspected and analyzed. Both tools were implemented using Python.

### A. LogEnhancer

The detailed work flow of the LogEnhancer is illustrated in Figure 4. This process is transparent and hidden from the developer and works as follows:

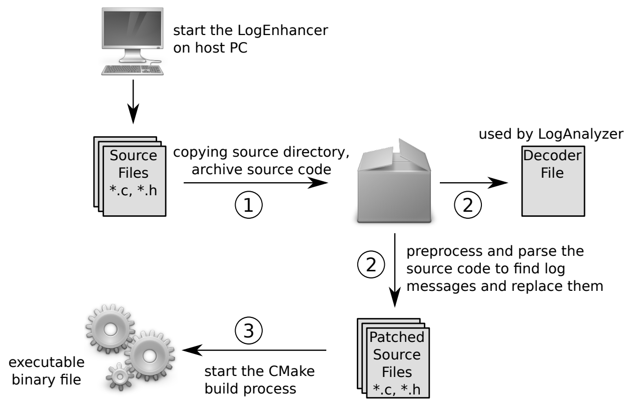

start the LogEnhancer on host PC
Source Files *.c, *.h
copying source directory, archive source code
1
used by LogAnalyzer
Decoder File
2
2
preprocess and parse the source code to find log messages and replace them
3
executable binary file
start the CMake build process
Patched Source Files *.c, *.h

Figure 4. Workflow of the LogEnhancer.

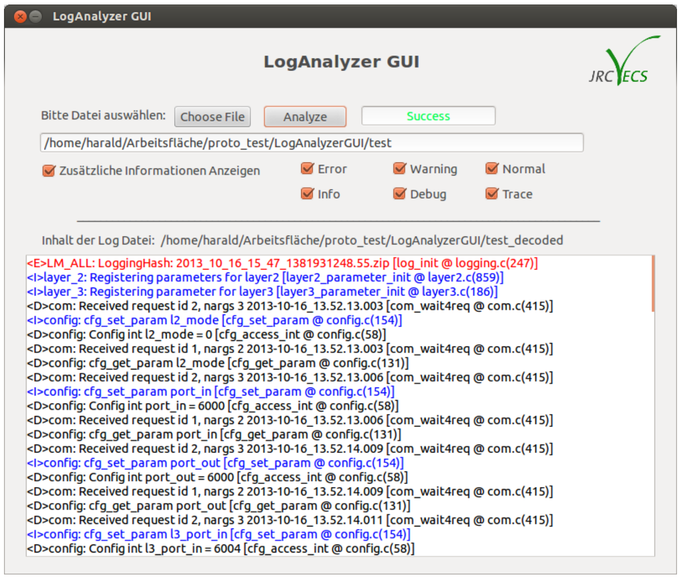
1. The tool creates a copy of the source code in a temporary sub-directory and additionally creates an archive file for backup purposes. For proper identification purposes both the directory and file-name use date, time-stamp, and version information in their name.
2. Next, the code is pre-processed using the compiler infrastructure. Based on the pre-processed files, the C-code is parsed using the Python module PYCParser that supports almost the entire C99 language standard. The parser generates a structured tree representation of the source-code that allows convenient identification of all logging commands. The module provides a function to locate specific function calls. Furthermore, the function calls are represented as a node in the tree and also includes information about the exact position of the logging-command within the source-code. This location information and the logging message is written to a decoder-file. The file-name of the decoder-file follows the same naming convention as in step 1 in order to provide a convenient mapping to the respective source-files. Furthermore, the file-name is added as an ERROR log-command to the source-code that gets executed once when the system is put into operation. A small additional tweak of the script that is responsible for the rotation of the log-files is necessary in order to retain this information. Next, the logging-command is modified in a way so that a single enumerator is output along with the values of the variables (if there are any logged at the particular location), cf. Figure 2.
3. The modified source-files are built by invoking the standard build process — our industrial partner uses CMake therefore. As a result, we obtain the executable binary that can be deployed to the target system.

All these steps are transparent to the developer.

### B. LogAnalyzer

The LogAnalyzer is run whenever a recorded log-file is analyzed in order to obtain readable log-files, cf. Figure 3. The involved steps are:

a. Scan the log-file for the file-name in order to map the correct decoder-file and source-tree to the log-file, cf.



Bitte Datei auswählen: /home/harald/Arbeitsfläche/proto_test/LogAnalyzerGUI/test

[✓] Zusätzliche Informationen Anzeigen [✓] Error [✓] Warning [✓] Normal [✓] Info [✓] Debug [✓] Trace

Inhalt der Log Datei: /home/harald/Arbeitsfläche/proto_test/LogAnalyzerGUI/test_decoded

```
<E>LM_ALL: LoggingHash: 2013_10_16_15_47_1381931248.55.zip [log_init @ logging.c(247)]
<I>layer_2: Registering parameters for layer2 [layer2_parameter_init @ layer2.c(859)]
<I>layer_3: Registering parameter for layer3 [layer3_parameter_init @ layer3.c(186)]
<D>com: Received request id 2, nargs 3 2013-10-16_13.52.13.003 [com_wait4req @ com.c(415)]
<I>config: cfg_set_param l2_mode [cfg_set_param @ config.c(154)]
<D>config: Config int l2_mode = 0 [cfg_access_int @ config.c(58)]
<D>com: Received request id 1, nargs 2 2013-10-16_13.52.13.003 [com_wait4req @ com.c(415)]
<D>config: cfg_get_param l2_mode [cfg_get_param @ config.c(131)]
<D>com: Received request id 2, nargs 3 2013-10-16_13.52.13.006 [com_wait4req @ com.c(415)]
<I>config: cfg_set_param port_in [cfg_set_param @ config.c(154)]
<D>config: Config int port_in = 6000 [cfg_access_int @ config.c(58)]
<D>com: Received request id 1, nargs 2 2013-10-16_13.52.13.006 [com_wait4req @ com.c(415)]
<D>config: cfg_get_param port_in [cfg_get_param @ config.c(131)]
<D>com: Received request id 2, nargs 3 2013-10-16_13.52.14.009 [com_wait4req @ com.c(415)]
<I>config: cfg_set_param port_out [cfg_set_param @ config.c(154)]
<D>config: Config int port_out = 6000 [cfg_access_int @ config.c(58)]
<D>com: Received request id 1, nargs 2 2013-10-16_13.52.14.009 [com_wait4req @ com.c(415)]
<D>config: cfg_get_param port_out [cfg_get_param @ config.c(131)]
<D>com: Received request id 2, nargs 3 2013-10-16_13.52.14.011 [com_wait4req @ com.c(415)]
<I>config: cfg_set_param l3_port_in [cfg_set_param @ config.c(154)]
<D>config: Config int l3_port_in = 6004 [cfg_access_int @ config.c(58)]
```

Figure 5. Graphical User Interface of the LogAnalyzer.

Section IV-A. In case no suitable files are available, a respective notification is output to the developer.

b. Next, the LogAnalyzer parses the log-file line by line and decodes the messages with the help of the decoder-file.

Using various options this tool allows to apply convenient filters to the log-file analysis. The LogAnalyzer tool itself is available both as a command-line tool and a program with a graphical user-interface (GUI), see Figure 5 for a screen-shot. The GUI variant, additionally supports a colored highlighting mode for a more convenient overview. The reconstructed output can be saved as a simple text-file.

## V. Evaluation

In order to evaluate our approach, we applied the tool suite on a industrial embedded systems project in the field of road pricing systems. One part of the project is a communication stack that handles the communication between an on-board vehicle unit and an observer unit located at defined way-points along traffic routes.

### A. Setup

In order to evaluate our logging approach, we chose the setup in a way so that we are able to control the frequency the communication is triggered in order to test the system in a reasonable time frame. To that end, the communication stack runs in a loop and produces frequent log-entries.

The primary focus of our evaluation was on the log-file size, however, we additionally monitored also the impact on the required computational resources. To that end we automated the tests using scripts and executed them multiple times for different time spans in order to avoid impacts due to the initialization process of the stack. The following presentation illustrates the results using test-runs spanning 10 and 30 minutes, respectively. As expected (and also evaluated by additional experiments) longer test-runs yield the same results.

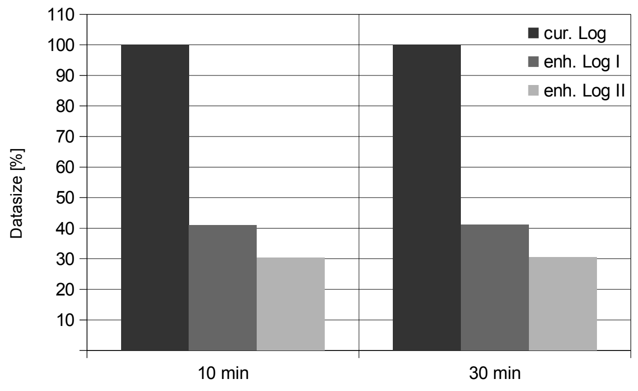


Figure 6. Log File Size with different Log Methods.

In addition we conducted the test-runs using different development stages of our implementation:

- **cur. Log:** These tests represent the reference runs where we executed the unaltered source-code using the standard logging method.
- **enh. Log I:** The first development stage of the tool suite replaces the entire text string of the log messages by a unique ID, thus only this ID along with a time-stamp is written to the log-file.
- **enh. Log II:** The second development stage includes an alteration of the time-stamp. Here the time-stamp is written in binary format without loss of information, whereas it is written in a readable format in both stages before.

### B. Results

Figure 6 presents the results of two test-runs spanning 10 and 30 minutes, respectively, using the most verbose logging level. The run with the current log method (cur. Log) is the reference run and represents 100% log-file size. Using our tool suite at the first development stage (enh. Log I) we are able to reduce the log-file size by 59%. Performance wise we could monitor a very small almost negligible reduction of the required processing power. The second development stage (enh. Log II) reduces the file-size even further down to an average of 69%. In particular, the time-stamp is reduced from 23 bytes in readable format to 7 bytes in binary format. This reduction of the needed bytes has no significant effect on the somputing performance. A field test shows that the current logging method generates a log-file with a size of 48 MB within one day. By using the LogEnhancer in the second development stage the size of the log-file is reduced to 19 MB without any loss of information.

Summarizing, the results show a high potential to save memory space by replacing the text string with a unique number and by modifying the time-stamp. The solution has only a marginal impact regarding the improvement of the performance.

## VI. Conclusion and Future Work

This paper presents an approach and some tools to improve the efficiency of logging in embedded systems with minimal alterations on an existing design flow, i.e., how a designer adds logging commands to the source code and how the log-files get analyzed. The solution reduces the log-file sizes, slightly optimizes required computing resources, and even adds more verbosity to the log-messages like file, function and line-number where a logging message emanated without further ado. As a benefit logging times can be increased before they get rotated and more verbosity can be added, thus improving the chances to catch rare or unlikely problems encountered in the field.

The solution was designed for an industrial road pricing solution in mind, is however, applicable ‘as is’ for any other kind of Embedded Linux application. It can be improved even further in various different ways. For example, in practice many log-messages recur multiple times, thus instead of logging the message one could simply log the message once and how often they recur. Furthermore, a tool that guides the designer where to insert logging commands and what messages shall be logged could help to further improve the chances to catch problems. Additionally, tighter integration with test- and version-management systems would be advantageous.

## ACKNOWLEDGMENT

This work has been supported by Kapsch TrafficCom AG and by the public funded R&D project Josef Ressel Center for Verification of Embedded Computing Systems (VECS) managed by the Christian Doppler Gesellschaft (CDG).

## REFERENCES

[1] D. Yuan, S. Park, and Y. Zhou, ”Characterizing Logging Practices in Open-Source Software”, 1em 34th International Conference on Software Engineering (ICSE), June 2012, pp. 102–112.

[2] J. H. Andrews, ”Testing using log file analysis: tools, methods, and issues, Proceedings of the $13^{th}$ IEEE International Conference on Automated Software Engineering, Oct. 1998, pp. 157–166.

[3] J. H. Andrews and Y. Zhang, ”General test result checking with log file analysis”, IEEE Transactions on Software Engineering, vol. 29, no. 7, July 2003, pp. 634–648.

[4] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage, ”Improving software diagnosability via log enhancement”, ACM Transactions on Computer Systems (TOCS), April 2012, pp. 3–14.

[5] R. Gerhards, The Syslog Protocol, http://tools.ietf.org/html/rfc5424, 2009, [retrieved: Sept., 2014].

[6] Y. Shi, L. Renfa, L. Rui, and X. Yong, ”Log analysis for embedded real-time operating system based on state machine”, International Conference on Mechatronic Science, Electric Engineering and Computer (MEC), Aug. 2011, pp. 1306–1309.

[7] P. Amontamavut, Y. Nakagawa, and E. Hayakawa, ”Separated Linux Process Logging Mechanism for Embedded Systems”, $18^{th}$ IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Aug. 2012, pp. 411–414.

[8] J. Jeong, ”High Performance Logging System for Embedded UNIX and GNU/Linux Application”, $19^{th}$ IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Aug. 2013, pp. 310–319.