Retaining efficiency in an embedded system while introducing Lua as a means to improve maintainability: an actor model approach

Bachelor of Science Thesis in Software Engineering and Management

R. JOHANSSON

J. TEBRING

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
Göteborg, Sweden, May 2011

Retaining efficiency in an embedded system while introducing Lua as a means to improve maintainability: an actor model approach

R. JOHANSSON

J. TEBRING

Examiner: HELENA HOLMSTRÖM OLSSON

University of Gothenburg
Chalmers University of Technology
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone: + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden, May 2011

Retaining efficiency in an embedded system while introducing Lua as a means to improve maintainability: an actor model approach

Johansson, R. and Tebring, J.

May 31, 2011

Abstract

Embedded touch display systems with limited resources still require responsive user interfaces, which puts high demands on the efficiency of the software. Embedded systems are expensive to maintain due to the low-level programming languages used for efficiency. By integrating the Lua scripting language as a complement to C, and using the message passing semantics of the actor model, maintainability and modularity can be increased while retaining efficiency. With this approach it is possible to develop efficient embedded systems with high maintainability responsive enough to be used in touch screen systems.

Keywords: embedded system, touch display, software design, component-based design (CBD), concurrency, actor model, message passing, Lua, efficiency, maintainability

1 Introduction

Today touch screens are used in a wide variety of consumer electronics. From a technological perspective there is a big difference between for example a general purpose device, such as a smartphone, and a special purpose device, such as a GPS[1]. These special purpose embedded devices would become too expensive if fitted with the same hardware as smartphones, but they still need to operate within the same responsiveness constraints. Therefore their software needs to be more efficient compared to that of smartphones.

It is common that new requirements, bug fixes or optimizations need to be implemented. A maintainable system decreases the cost and time required to perform these modifications. Software maintenance has always been a large part of the total system cost, and usually accounts for up to 70% of the total cost (Brooks, 1975; Kim and Weston, 1988; Pearse and Oman, 1995; Aggarwal et al., 2005). In a real-time embedded system the cost of maintenance can be up to four times that of development (Sommerville, 2006).

The levels of abstraction, like object orientation, that reduce source code complexity increase maintainability but have a negative effect on efficiency. Therefore these two quality attributes are opposed.

Subsequently, the research objective was to evaluate three hypotheses by developing an embedded touch display system, in which main-

[1]For instance, the HTC Desire (a smartphone) has almost four times the computing power and nine times the memory of the TomTom XL IQ Routes Edition 2 (a GPS device).

tainability and efficiency are prioritized. The hypotheses we have been focused on are the following; Lua as a complement to C increases maintainability while retaining efficiency; Message passing increases maintainability at the cost of efficiency; Lua is efficient enough to manage a touch display in an embedded device.

C is one of the dominating programming languages in embedded systems, it is very portable and can be used to write resource efficient software. Because of this, it is used as the main programming language in the system.

The main strategy for improving maintainability is integrating a high-level language as complement to C and a system design that simplifies modularity and concurrency. Previous research has suggested that high-level languages, which are more productive and produce less lines of code (Ousterhout, 1998; Prechelt, 2002), improve maintainability (National Bureau of Standards, 1984; Sanner, 1999; Ward, 2003).

Lua was selected as the high-level scripting language. It can easily be integrated with C. Two of the major benefits of Lua, compared to C, is that it has garbage collection and dynamic data structures. Lua is used to define the behavior of the system by connecting other software modules. This makes the source code defining behavior shorter and allows it to be maintained by programmers who are not familiar with low-level development or memory management.

The system has been designed so that each functional module runs in its own process. Message passing is used for inter-process communication. These concepts derive from the actor model (Hewitt et al., 1973; Agha, 1986) which is the concurrency model used by for example Erlang. From the programmer's point of view the message passing interface is very simple and transparent (Lieberman, 1981).

The study relies on design research to iteratively determine if Lua can be introduced as a complement to C, in a modular design using the actor model, in order to increase maintainability while retaining efficiency. Our contributions will be to argue a solution and demonstrate its feasibility through the evaluation of three hypotheses, and by providing responsiveness measurements of the design artifact. To support our design decisions which increase maintainability, we have used previously established guidelines and methods found in literature. Benchmarks from the system are provided that show system responsiveness and resource utilization for both Lua and the message passing mechanism.

This paper is structured as follows: Section 2 describes the underlying theory that the artifact's design was based on. Section 3 presents the research approach and the method used. In section 4 the implementation, discoveries and results are presented. Section 5 discusses the method used, results, reviews the initial hypotheses and suggests further work. Finally, section 6 gives conclusions of the research.

2 Theoretical foundation of the design

In this section the theory from which the research is derived can be found. This includes the background of why certain decisions have been made and the theoretical foundation of the system design.

2.1 Quality attributes

There exists different classifications of quality in software (Boehm et al., 1976; McCall, 1977; IEEE/ANSI, 1993; ISO/IEC, 1999, 2001).

In this paper we rely on ISO/IEC (2001) definitions of the different quality attributes. ISO/IEC (2001) groups all quality attributes by six characteristics: efficiency, maintainability, reliability, portability, usability and functionality. Two specific quality attributes have been target of interest and prioritized in the system developed during this study, namely efficiency and maintainability.

The definition of efficiency incorporates time behavior, resource utilization and efficiency compliance. Efficiency was prioritized due to the hardware constraints and because the touch screen interaction requires high responsiveness. Like in most systems, as much as possible is wanted out of the given hardware.

Responsiveness is a part of time behavior and an important factor when interacting with a system. It is elusive; in order to assess it in the system, the definition and formal metrics provided by Seow (2008) were used. Responsiveness is subjective, nonexclusive and relative to the interaction. Subjective because some users will be more sympathetic to delays than others. Nonexclusive since any form of indication can be interpreted as a response. It is relative because different forms of interactions will have different windows of acceptable response times. For example, if a request is simple, the user will expect a shorter response time compared to a more complex request. According to Seow (2008), graphical components that mimic physical objects, e.g buttons, should show instantaneous responsiveness (less than 0.1 seconds). Other more complex interactions, e.g. displaying a menu, should show immediate responsiveness (less than 1 second).

The definition of maintainability incorporates analyzability, changeability, stability, testability and maintainability compliance. This means that maintainability is the ease with which software can be changed to satisfy user requirements or can be corrected when deficiencies are detected (National Bureau of Standards, 1984).

Because the system was built from scratch, maintainability was important because of the initial time investment required. The system will be durable over a longer period of time and it will also be easier to extend and reuse software modules in a modular design. Maintainability is a big portion of the total system cost (Brooks, 1975; Kim and Weston, 1988; Pearse and Oman, 1995; Aggarwal et al., 2005) and by increasing maintainability, money will be saved over time. In accordance with the guidelines established by National Bureau of Standards (1984), the focus on maintainability existed since the beginning of the system design, and feedback from iterative development, has been used to improve it:

> *If the software is designed and developed initially with maintenance in mind, it can be more readily changed without impacting or degrading the effectiveness of the system. This can be accomplished if the software design is flexible, adaptable, and structured in a modular, hierarchical manner.*
>
> National Bureau of Standards, 1984

Efficiency and maintainability are in some cases working against each other. One reason for this are the levels of abstractions, like object orientation, used by the developers to increase maintainability consumes more hardware resources which in turn decreases efficiency. A typical example is the use of low-level programming languages (like assembly) compared to a high-order language (like Lua). It is possible to implement a very efficient system in a low-level programming language, but it will decrease the maintainability in the sense of code readability and understandability (levels of abstractions are added to make it simpler for humans to understand).

The other quality attributes defined by ISO/IEC (2001) have also been given consideration, but have not been prioritized as highly as efficiency and maintainability during this research.

One of these attributes is reliability which is the system's capability of performing as intended over a period of time. The ISO/IEC (2001) definition of reliability includes maturity, fault tolerance, recoverability and reliability compliance. Reliability will be indirectly improved if the modularity, testability and analyzability aspects introduced through maintainability are used in the quality assurance of the system.

Portability consists of adaptability, installability, co-existence, replaceability and portability compliance. The portability of the system is improved because of the use of ANSI C (Lua is written in ANSI C as well). Programs written in C can be compiled for a wide number of hardware architectures. The modular design of the system makes it easy to change hardware components simply by writing a new driver.

Usability contains understandability, learnability, operability, attractiveness and usability compliance. This attribute will be more important later on when designing the graphical user interface (GUI) and other higher level applications. It is favored by how well the software can mimic physical objects (Seow, 2008). By prioritizing efficiency and the responsiveness of the touch display it is also possible to improve usability.

The final quality attribute functionality, which includes suitability, accuracy, interoperability, security and functionality compliance, is an important quality attribute for any system. By allowing the behavior of the system to be defined in a highly maintainable way, suitability and accuracy will be easier to increase as a result. The message passing mechanism in the system could be expanded, to allow messages to be received from remote systems and passed on to the local components transparently, greatly increasing interoperability.

2.2 C and Lua

Based on the two quality attributes prioritized, efficiency and maintainability, the programming language C and the scripting language Lua were selected for implementation.

C was selected because it gives full control of resources, like memory usage or other hardware, and can thus be used to create very efficient software. It is commonly used in embedded systems and there exists compilers for a variety of hardware architectures making it very portable. Other candidates were C++ and Java, which are both object oriented but not comparable to C when it comes to efficiency (Prechelt, 2000), furthermore object orientation was only considered necessary in the application layer and not in the whole system.

To increase the system's maintainability a scripting language was selected as a complement to C. In opposition to C, a scripting language produces in most cases less lines of code and is more productive (Ousterhout, 1998; Prechelt, 2002), which increases maintainability (National Bureau of Standards, 1984; Sanner, 1999; Ward, 2003). The lines of code is the most important factor affecting maintainability according to Ward (2003). Memory management related bugs are common when writing programs in C. Code written in a scripting language that automatically manages memory can be maintained by programmers that are not familiar with memory management.

Scripting languages come at a cost: they are less efficient (Ousterhout, 1998) requiring more resources (Prechelt, 2002). The combination of C and a scripting language allows a compromise between efficiency and maintainability: C is used where efficient resource utilization is required (e.g. drivers). And Lua is used where flexibility and productivity is important (e.g. application writing).

There exists many scripting languages, but few which are fit for embedded systems. For example, Lisp and Scheme were excluded because they are not adapted for embedded devices, quite uncommon these days and their syntax was unfamiliar. Perl and Python were excluded due to the big footprint. A Python related project called python-on-a-chip could have worked on the hardware but it was excluded due to its GPLv2 copyleft license, which is not compatible with the proprietary license of the system used in this research. Tcl was excluded because of its primitive syntax, slow performance, lack of data types and it does not offer good support for data description (Ierusalimschy et al., 1995, 2007). The final choice stood between Rexx and Lua. Lua has been used successfully in embedded systems before (Clark, 2009). Furthermore, it is widely used in the gaming industry, for instance used in the games World of Warcraft and The Sims (Ierusalimschy et al., 2007). Clark (2009) describes how Lua has been embedded in to smart instruments allowing users to access all the underlying instrument commands through a Lua interpreter. Lua is a good complement to C according to Roberto Ierusalimschy, one of Lua's authors, who describes it this way (Ierusalimschy, 2006):

> *Lua is a tiny and simple language, partly because it does not try to do what C is already good for, such as sheer performance, low-level operations, and interface with third-party software. Lua relies on C for these tasks. What Lua does offer is what C is not good for: a good distance from the hardware, dynamic structures, no redundancies, ease of testing and debugging. For this, Lua has a safe environment, automatic memory management, and good facilities for handling strings and other kinds of data with dynamic size.*
>
> Roberto Ierusalimschy, 2006

Since version 5.0, Lua is released under the MIT license, which allows Lua to be integrated with proprietary software[2] and altered in any way. Additionally, it is still under development (5.2 alpha released Nov 23, 2010) and compiles on all platforms that have an ANSI/ISO C compiler, which makes it very portable. Lua can easily be configured to use a custom memory allocation function. This can be an advantage in an embedded system to assure Lua is not consuming more memory than allowed, which can not be controlled or monitored using the standard function in C.

The API between Lua and C is bidirectional, this makes Lua both an extension and extensible language (Ierusalimschy et al., 2007). Embedding Lua in a system gives it a mature, powerful scripting facility and it can create and control as many Lua virtual machines as required (Hirschi, 2007). Moving behavior to Lua scripts gives many advantages: logic can be loaded on demand, tests can be set up faster, behavior becomes more accessible making it more convenient to review and the overall product evolves easier (Hirschi, 2007).

Lua includes a full suite of standard libraries, that are internally linked to the standard ANSI-C libraries (Clark, 2009). Some of the libraries are: I/O, string, math, OS, debug. It is possible to select which libraries should be included in order to reduce the footprint by removing unused libraries.

The scripts executed through Lua can be easily sandboxed, so embedding Lua in a system does not come at the expense of security. Sandboxing can be done by not loading libraries or by overriding individual functions using closures (Ierusalimschy, 2006).

Object oriented programming can be done in Lua using tables and metamechanisms (de Figueiredo et al., 1996). Inheritance is also

[2]On the condition that the license is distributed with the software.

possible through fallbacks that allow the value of an absent field to be looked up in another table (its parent) (de Figueiredo et al., 1996).

2.3 System design

The system used, for exploring how maintainability can be improved while retaining efficiency, is an embedded touch screen device (specification in Section 3.1).

The system uses a layered architecture. It is one of the most common architectures in embedded systems and it has proved to work well (Bass et al., 2003; Noergaard, 2005).

The layered architecture helps to create levels of abstractions in the system which increases maintainability. For example, hardware drivers are implemented separately in one layer and controlled by components in the layer above. The system has an underlying real-time operating system (RTOS), which makes it possible to run modules as separate processes concurrently with full preemption.

In order to deal with the added complexity of using two programming languages and still improve maintainability, a modular design was used to integrate all software components (National Bureau of Standards, 1984; Sommerville, 2006). There is a strict separation between what is written in C and what is written in Lua. The software modules are written in C and then applications, that define how these modules should behave and interact, are written in Lua.

National Bureau of Standards (1984) states three basic design principles software modules should be constructed with to increase maintainability; Modules should perform only one principal function; Interaction between modules should be minimal; Modules should have only one entry and one exit point.

These three design principles have been taken into account while designing the system. Modules are decomposed by functionality (Sommerville, 2006), for instance each hardware driver is a separate module (the touch display is represented by two: one for input and one for output). The interaction between the modules are represented by delta changes and events, to decrease the amount of messages. As entry and exit points message passing (send and receive) is used.

2.4 Actor model

The touch display system greatly benefits from using concurrency, for instance, the display can be re-rendered while reading input from the touch screen. Another example is that the GUI can work independently and does not freeze while receiving serial data.

The actor model (Hewitt et al., 1973; Agha, 1986) was used in the design to handle concurrency and to support a modular design. It was used because it provides a low complexity concurrency model and allows an intuitive and process-safe way of sharing data between processes. In the actor model each concurrent process is called an actor and the actors communicate with each other through message passing. The actor model also supports dynamic creation and destruction of processes at runtime, but this has not been implemented nor used during this study since it was not required in order to accomplish the research objective. The operating system is capable of prioritizing processes, in our case actors, which can be used in order to assure that important modules are executed as intended.

Lieberman (1981) argues that the actor model allows parallelism and synchronization to be implemented transparently, so that parallel and synchronized resources can be used as easily as their serial counterparts. As a result, each module's code, interacting with other modules, becomes less complex and more readable. The concept of message passing can be compared to e-mail. To send an e-mail all that is needed is the address of the recipient and incoming messages will be put in an inbox. Similarly, the messaging API consists of 2 functions, one for sending messages and one for retrieving messages from the inbox. Just like e-mail, all messages are sent asynchronously. This makes the sending actor independent of the receiving actor (given it has the receiver's identifier). However, it is still possible to simulate synchronous calls: the receiver replies to the sender who in turn waits for a response after sending the message. A problem with asynchronous message passing is that the system becomes non-deterministic (Berry, 1989). It is impossible to predict the state of the system at any given time when actors are interacting. A reason for this is that an actor is not required to handle the incoming messages at all nor within a certain time. The number of pending messages can not be determined in advance. Furthermore, each message must be handled before the next in the queue (in the normal case, but it is up to the actor's implementation), which may also differ in time depending on message type.

Different kinds of actors can easily be implemented in the system. An example would be the future actor (Lieberman, 1981), which can be initialized with a computation to be calculated. It will calculate the result and return it to the sender when done. In this way larger calculations can be executed in parallel without locking the acquiring actor. Shared memory, like the blackboard pattern, can also easily be implemented as an actor with the advantage of being process-safe (Lieberman, 1981). This means that shared memory can be replaced, increasing maintainability (Sommerville, 2006), with an actor instead and the risks for deadlocks decreases. Busy wait can easily be used simply by waiting for incoming messages with infinite timeout. This causes the process to sleep until a message is placed in its message inbox.

The actor model gives a strict, but yet simple, interface between for instance a C module and a Lua module. This separation makes it easy for Lua developers to only focus on Lua, but still be able to pass messages to the rest of the system. It is also possible to integrate modules written in other programming languages in the design, simply by writing a wrapper for the actor model interface.

3 Research approach

This section introduces the research method which was used during the study. It also describes the conditions under which the research was performed, the contributions made and the limitations of the research.

3.1 Research setting

The research was conducted during two months in the beginning of year 2011 after which this report was written. The study was performed at a company that engineers security solutions. The problem the company faced was to have a highly responsive embedded touch display system while keeping it maintainable, adaptable and extendable for future changes. Additionally, third party application development was a requirement, which puts high demands on ease of integration and sandboxing. The system setup and requirements were established by the company before this study started.

The following system setup was selected by the company for the new system:

- ARM7 72MHz (512KB internal flash)
- 8MB SDRAM
- 4.3" TFT Touch Screen (272x480 pixels)
- USB interface
- Micro SD card reader
- Real-Time Operating System (RTOS)
- *Note: no GPU nor FPU*

The research is positioned in the design segment of the software engineering area. Specifically the research focus is about balancing quality attributes in a newly developed concurrent system using a component-based design (CBD).

3.2 Research design

Based on the reliance on iterative exploration as the means for theoretical and practical reflection, this study relies on design research (Hevner et al., 2004). Design as a process is a sequence of activities that produces an innovative product (i.e. the design artifact, which is design as a product).

The iterative nature of continuously improving the design artifact, by shifting perspective between design processes and the designed artifact, supports a problem-solving paradigm that can be used to solve complex problems.

Through *build* and *evaluate* we can use design research while designing the system to continuously monitor to which extent the maintainability improvements affects efficiency. The benefit of being able to continuously monitor the impact of design decisions is that the artifact can be directed as new knowledge about the problem is gathered.

The build and evaluate was divided into iterations and were carried out based on the model proposed by Vaishnavi and Kuechler (2008). The iterations were started by identifying a problem with the current design. Once a problem was identified a solution was proposed, which was then applied to the design artifact. The resulting artifact was evaluated based on system requirements and the new knowledge gathered fed back into the design process. The concluding phase of the iterations was used to determine if the artifact can be used to evaluate the hypotheses. In which case, the iterative phase of the design research ended; otherwise a new iteration was carried out.

This study was design research and not routine design or system building because the research addresses an existing problem in an innovative way. Compared to routine design, which is the application of existing knowledge to organizational problems, we have contributed with new knowledge about how combining a complementary high-order language and the actor model in an embedded interactive system can increase maintainability while retaining efficiency.

Two iterations were executed before it was possible to evaluate the hypotheses and accomplish the research objective. At which point the iterative phase of the design research ended.

The problem targeted in the first iteration was to establish an initial system design that would result in higher maintainability. The solution was to integrate Lua as a complement to C in a modular design using the actor model's message passing semantics. This was implemented on the embedded device and evaluated in terms of maintainability and hardware resource utilization. The design artifact's utility was evaluated using informed argument with regards to maintainability based on the existing knowledge base. At this point it was only possible to measure the responsiveness of the touch display and the efficiency of the message passing qualitatively, and it seemed to be fast enough for its purpose. The reason for not being able to measure the responsiveness, an emergent system property, was because the UI process which has a central role in the system had not been implemented. A way to quantitatively measure the responsiveness of the touch display was required, and a new iteration was started.

The second iteration's problem was the inability to measure crucial aspects of the system's efficiency. In order to accomplish this an initial implementation for a UI process was required. The solution was to design and implement a basic UI process to handle touch display events and render graphical components. Benchmarks were also set up to measure time behavior in different aspects of the system. The responsiveness of the GUI was measured and compared to established guidelines of human perceived responsiveness. Static and dynamic analysis were used to evaluate the design. Static analysis was used to examine that the structure of the artifact followed the theoretical foundation. Dynamic analysis was used to measure time behavior of the message passing implementation and responsiveness of both the touch display and the UI process. After the second iteration, results supporting the research objective became evident, thus halting the iterative work in order to focus on reflecting on the findings.

3.3 Research contributions

The main contribution of the research is the evaluation of an instantiation, the design artifact. The design artifact combines existing knowledge in a modern embedded system design. It demonstrates the feasibility of using Lua as a complement to C, combined with the actor model in a modular design. The feasibility of the solution is shown through the evaluation of three hypotheses, and by providing responsiveness measurements and support our design decisions by using previously established methods found in literature. Additional benchmarks contribute to enable concrete assessment of the design artifact. These contributions show that the solution is suitable to its intended purpose: as a maintainable and responsive embedded touch display system. The solution is not specific to embedded touch display systems and could be applied to any embedded system. However, we believe that the approach is particularly useful in interactive embedded systems.

3.4 Research limitations

The impact of maintainability as a measurement over time will not be performed during this study. Instead, previously proved solutions to increase maintainability will be used in the software design (see Section 2.1).

There was no need to implement a fully functional actor model during this study. The concurrent message passing was the subset of it implemented (see Section 4.7). Properties like starting and stopping actors dynamically at runtime can easily be implemented in the future if the system requires it.

Real time constraints of the system is out of scope for this study but may have been plausible (see Section 5.7).

4 Software construction

This section shows the discoveries and results found during the study. It starts by explaining the Lua integration on the embedded device, continues with a system overview before describing the actor model implementation. Finally, the Lua and GUI processes are described and the touch display responsiveness benchmark is presented.

4.1 Software setup

All software was written in ANSI C (ISO, 1999) and Lua 5.2 alpha (which also is implemented in ANSI C). The system was compiled, with arm-eabi-toolchain[3] using newlib[4] as C library, to one binary file (monolith) which fits in the CPU flash where it is executed. Compiler flags to strip unused data from the data segment and functions were used to decrease the application footprint. Lua was compiled with the optimiza-

[3] `https://github.com/jsnyder/arm-eabi-toolchain`

[4] `http://sourceware.org/newlib/`

tion level two flag (-02), which is the default setting.

4.2 Getting Lua to run

There were some minor problems integrating Lua and the operating system. The first issue was that the default process stack size was too small in the operating system and by increasing it Lua worked fine. The second issue was that the Lua environment required a process-safe memory allocation function, this was solved by using a different function when instantiating it. This will block the scheduler and prevent all other processes from being executed during the allocation, which in turn slows down the entire system. But it is required and we did not consider it slow enough to implement our own process-safe memory allocation method. When these issues were solved Lua's garbage collector executed as intended despite of the operating system's preemption. Noteworthy is that Lua uses double as standard data type for all its calculations but can be recompiled for other numeric types, which means an optimization can be made in the target embedded system (which may lack a FPU) by using fixed point numbers instead.

4.3 Lua footprint

Lua is said to have a small footprint of around 100kB, but we were not able to match that during this study. The footprint of the whole system with and without the Lua process was measured. This made it possible to see the difference in size when Lua was not used to make an assumption of the footprint impact. Lua's default configuration with all its libraries included was used while measuring. The results are shown in Figure 1.

The table shows that Lua increases the footprint of the system with around 240kB (Lua uses -O2 by default) in our implementation, which is acceptable considering the additional advantages of the scripting facilities. Lua uses no static data, and the size decrement of the data and bss segments is stripped away software related to the Lua process itself.

	.text	.data	.bss
Without Lua	104592	2384	20776
With Lua	401752	2640	21300
With Lua (-O2)	342840	2640	21300
With Lua (-Os)	319608	2640	21300

Figure 1: Lua footprint impact (in bytes)

4.4 Load Lua environment

The Lua environment is accessed through a single variable and can easily be instantiated in different concurrent processes. This makes it possible to execute several Lua environments (e.g. scripts) in parallel. It is also possible to pass a Lua script as an actor message to a Lua process which interprets and executes the script in an existing environment.

A benchmark of loading the Lua environment was performed to determine the actual startup time of Lua. To remove the variations in the scheduler, these tests were executed without an operating system as a single application. Lua scripts of different sizes were loaded and executed in a new environment. At the bottom of each script the same function was placed, and it was called directly after the script was loaded.

The time measured is the time it took to create a new Lua environment, load the script and call the function. The results of the benchmark are shown in Figure 2.

Scripts of different sizes were used because we wanted to see how the size affected the loading and execution time in the Lua environment. None of the scripts were loaded from a disk but instead compiled statically into the program. This was done to avoid deviations in I/O ac-

	Script size	Load time
State only	0 bytes	15 ms
Tiny	32 bytes	16 ms
Small	535 bytes	23 ms
Medium	1478 bytes	32 ms
Large	5224 bytes	64 ms
Huge	18210 bytes	212 ms

Figure 2: Lua environment loading time

cesses, but may not reflect the actual time it takes to load the script compared to if it was loaded from for instance an SD card.

4.5 Lua memory consumption

Lua has a garbage collector (GC) which manages the memory used in the Lua environment. This could be dangerous in an embedded device if the GC is not properly executed and results in allocated but unused memory.

The dynamic memory usage of a Lua environment has been observed to see how much memory is allocated before it is freed. In our implementation, it seems that Lua starts at a level of around 40kB allocated which increases up to around 150kB at a maximum. When reaching the upper level the GC is executed and the memory usage drops to around 100kB and around 50kB is freed. A cycle from where the memory is freed until it is freed again, takes about ten seconds. It is also possible to force the GC to be executed which releases even more memory than if it was executed on a normal basis.

4.6 System overview

The system design has been kept simple just because it increases maintainability (National Bureau of Standards, 1984). It is easy to understand and grasp the concept if one is new to the system. There are low-level parts which are designed to be very efficient, and not that maintainer friendly, but this is where the compromise between efficiency and maintainability is revealed. Based on the assumption that the parts which are efficient, like the hardware drivers, are rarely altered. On the other hand, the application logic which has been developed to be very flexible through Lua can easily be modified and maintained.

As mentioned in Section 2, a layered architecture was used in the design. The hardware drivers are located at the bottom as static methods and data. These are then used by processes, one for each driver or hardware component. This also makes it very easy to simulate hardware: create a process sending and receiving the same messages as the real hardware driver. It is also easy to change the underlying hardware: create a new hardware driver and replace the old one behind the process (which hopefully can use the same message passing interface). Because of the abstract message passing interface it is easy to integrate other programming languages in the system, simply by writing a wrapper for it.

A part of the modular system design can be seen in Figure 3. The named circles represent concurrent processes which communicates through asynchronous message passing, here shown by solid arrows. The dashed arrows represent synchronous data flows from and to the lower layers in the design. The input (touch function) and output (rendering) were separated in the touch display driver to make message passing even simpler.

4.7 Message passing

The operating system used has a data queue that passes data safely between processes, however we found that the interface was not convenient enough, we wanted to introduce the actor model's message passing semantics. In the implementation, a queue is created to act as a message inbox for each process when the pro-

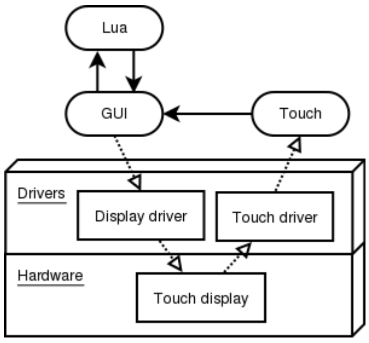

Figure 3: System overview

cess is created. A centralized queue would have had a negative impact on efficiency, by becoming a bottleneck during heavy load. Furthermore, it would have made the message passing implementation more complex, reducing maintainability. A simple send/receive interface hiding the queues was created to make it easy to manage messages. If a process wants to pass a message to another process, the message is enqueued to that process' inbox queue by passing the target process' identifier to the interface along with the message. Each process is responsible for continuously dequeuing the messages from its inbox queue through the interface. How the messages are passed through the inbox queues can be seen in Figure 4. This message passing interface has been made available to both C and Lua.

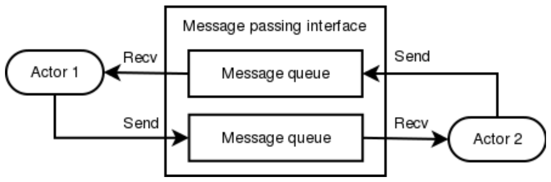

Figure 4: Message passing queues

To avoid flooding the amount of messages is minimized by only sending delta changes or events. The messages passed between processes are kept small because they are passed by copy for inter-process safety. A message consists of following: 1 byte for sender id, 4 bytes for control bits and 4 bytes of data. This makes the total size of any message 9 bytes.

Currently the sender byte is set by the sender itself, but this will later on be automatically set behind the scenes to make it more secure.

The control field is used for describing what kind of message it is. The first byte is generic for all messages and the other three are message specific. The three message specific bytes can also be used for passing data if required.

Sending larger amounts of data is possible by passing a pointer stored in the data variable. A bit in the generic control byte tells that a data pointer is passed instead of the data itself. Also, setting the corresponding bit in the control field tells the receiver to free the memory when not needed any more. If the same bit is not set the contents of the passed data pointer is read-only.

To get insight in how long it takes to pass messages, a benchmark environment was setup. A message was passed from one module to another which passed it back again through the send and receive interface. This was performed multiple times to get an average of the time it took to do one pretended synchronous message passing. The results are shown in Figure 5.

	Delay
C to C	0.181ms
C to Lua	0.359ms
Lua to C	0.286ms
Lua to Lua	0.428ms

Figure 5: Message passing delay time

An example implementation of an actor can be found in Figure 6. The actor will read its inbox and reply the sender with the same message, but with the sender identifier exchanged to its own.

```
void actor(void* parameters)
{
   pid_t pid = register();
   int timeout = 0;
   message_t msg;

   while (1)
   {
      sleep(10);

      if (recv(pid, &msg, timeout) == 0)
      {
         pid_t sender = msg.sender;
         msg.sender = pid;
         send(sender, &msg, timeout);
      }
   }
}
```

Figure 6: Example actor written in C

4.8 The Lua process and custom Lua libraries

The Lua process is general in the sense that it can run any Lua script that defines an entry point, in this system a function called init. Several instances of this process can be run concurrently.

Three types of applications are supported. The first type is a callback application, defined by returning nil in the init function. When a Lua process is running a callback application it will wait for incoming messages and if the Lua script has provided callbacks for these types of events they will be executed. Callback type applications are particularly suitable for GUI applications, since these often only update when the user has interacted with the system. The second type is a tail call application, defined by returning a function in the init function. Each subsequent function returns the next function to be called. Tail call applications can, for example, be used to model finite state machines. The third type of application is a mix of both, that is, an application that receives callbacks and executes iteratively. Figure 7 is an example of the latter. It continuously increases the counter value, until the label is pressed (the reset_counter() function will be executed).

A Lua process can receive scripts through messages that it will load and execute in its current Lua environment. Thus an application to be modified in any way at runtime (variables and functions can be changed or new ones added). This feature combined with USB communication allows interactive consoles, uploading scripts and debugging (using Lua's debug library) at runtime.

```
function init()
    counter = 0
    g = gui.new()
    label = g:new_label("Hello world!")
    label.position = {50,50}
    label.press = reset_counter
    return main
end

function main()
    counter = counter + 1
    label.text = "Hellos: " .. counter
    return main
end

function reset_counter()
    counter = 0
end
```

Figure 7: Example Lua application

When designing the custom Lua libraries used in the system, the focus was on developing user friendly interfaces to the rest of the system. An example application using the library would be written and when it was as simple and intuitive as possible then the actual library would be implemented. This approach is a recommended way of designing interfaces to make them more intuitive and maintainable (Martin, 2009). The simplest of the libraries, those that wrap up messages to other processes, were written directly in Lua and the more complicated ones written in C or a combination of both. Examples of a wrapper and its usage written in Lua can be found in Figures 8 and 9.

```
function init()
    -- off, during initialization
    indicator:set_mode(0)

    -- Do some initialization

    indicator:set_color(255, 0, 0, 255)
    indicator:set_mode(3) -- sinus
    return main
end

function main()

    -- Main loop

    return main
end
```

Figure 8: LED indicator usage

```
StatusLED = {}
StatusLED.__index = StatusLED

-- Constructor
function StatusLED.new(pid)
   local led = {}
   setmetatable(led, StatusLED)
   led.pid = pid
   led.timeout = 0
   led.mode = 0 -- off
   led.r = 255
   led.g = 255
   led.b = 255
   led.a = 255
   return led
end

function StatusLED:set_mode(mode)
    self.mode = bit32.lshift(mode, 8)
    send(
        self.pid,
        self.mode,
        self.r+self.g+self.b+self.a,
        self.timeout
    )
end

function StatusLED:set_color(r, g, b, a)
    self.r = r -- no shift
    self.g = bit32.lshift(g, 8)
    self.b = bit32.lshift(b, 16)
    self.a = bit32.lshift(a, 24)
    send(
        self.pid,
        self.mode,
        self.r+self.g+self.b+self.a,
        self.timeout
    )
end

indication_actor_pid = 2
indicator = StatusLED.new(
    indication_actor_pid
)
```

Figure 9: LED indicator wrapper in Lua

4.9 The user interface process

The user interface (UI) process is a central part of the system, responsible for updating GUI components on screen and passing touch screen events to a theme library (described below) and Lua applications. It synchronizes the GUI components with the touch screen, for instance when GUI components are updated in a way that affects the touch screen, a message containing the updated information will be sent to the touch screen process. The information sent to the touch screen process is a list of rectangular regions containing flags for the events a given region is listening for. If an event is generated within any of these regions, and the flag for that event has been set, the touch screen process will report to the UI process. The reasoning behind passing the touch events through the UI process, and not directly to the application, is that a theme library can quickly give user feedback (for instance redrawing a button in an active state or playing a sound) independently of any delay caused by the application written in Lua.

The theme library is configured through Lua; it sets the appearance and behavior of standard components such as buttons, labels and sounds. Sounds will be used to give complementary user feedback but have not yet been implemented in the system.

In addition to relying on a theme library the UI process also relies on a GUI library to render graphical components to the screen. The GUI library is a very simple library that draws directly on the framebuffer and it only knows how to draw bitmaps and texts. The components appearing in the GUI are accessed through an array that indexes them by their id, and updated through messages from the controlling application. This makes it efficient to update a given component and iterate over all components to redraw them. The controlling application does not need to know anything about how or when components are redrawn, it all happens internally in the GUI library, thus a GUI application can be as simple as the one shown in Figure 7. More complex components are made up of a combination of bitmaps and texts, for instance a button contains a bitmap as background and a text as label. Those two basic components are implemented in C and the rest are implemented in Lua.

4.10 GUI responsiveness

Using the formal metrics suggested by Seow (2008), the responsiveness of the GUI can be evaluated. Simple user input events, such as pressing down on a button, should in order to avoid having a negative impact on usability provide user feedback instantaneously (less than 0.1 seconds). The tolerance for clicking (release after pressing down) a button is higher, since this is a slightly more complex operation the user feedback should be immediate (less than 1 second).

The benchmarks shown in Figure 10 are from a Lua application that has two buttons (182x51 pixels) and a long text. The benchmarks started from when the touch process sent the event and ended when the UI process had updated the display. The text spans 12 lines and covers about two thirds of the screen. Pressing down either of the buttons will cause them to be redrawn in an active state, i.e. the background image of the button is changed. When clicked they cause the text to be changed.

	Response time	Limit
Button down	9 ms	100 ms
Button clicked	79 ms	1000 ms

Figure 10: GUI response times

5 Discussion

This section discusses the results of the research found in the previous section. It will revisit and review the hypotheses stated earlier in the paper. Furthermore, it will bring up issues that are not covered in this research, but which we have found interesting.

5.1 Research approach

The iterative structure of design research helped us to focus on single issues, developing and improving the design artifact one step at a time while trying to retain efficiency.

During both iterations a lot of time was spent on project setup, configuring the operating system, developing hardware drivers and software resource management (images and fonts). This made the iterations consume more time than expected but not to a problematic extent.

There was no way of quantitatively measuring the efficiency of the system during the first iteration, simply because everything was developed from scratch and it is difficult to measure things which do not work properly. Instead qualitative measurements were used in the first iteration to assure the system continued to be efficient and responsive. In the second iteration enough software was developed and quantitative measurements were setup. After this iteration and by the help of design research, we managed to get the results we were looking for to evaluate the initial hypotheses and to make a final conclusion.

5.2 System design

The design was kept simple because it increases maintainability. Also the interfaces and APIs were designed to be simple. As an example, the interfaces were designed the way we wanted the Lua application to interact with the rest of the system before they were implemented. This approach is normally not the easiest when it comes to implementation, but when done, it makes the API look like and work as one would expect it to (think of a black box interface).

Because it was not a proof of concept (throwaway) prototype, we were forced to dig deeper and learn more in-depth, so the learning outcome was greater. It would have been easier to make a proof of concept only by creating a prototype for a special purpose. An example would be just running Lua on an embedded device. But this may not have proved if it actually is possible to control a touch display through Lua with high responsiveness. There are too many factors which impact the efficiency of the system to make a proper evaluation of prototype with only Lua in it. The resulting design artifact developed during this study is the core of a complete embedded touch display system system which is going to be used in industry.

5.3 Actor model

In the actor model implementation the focus was on the message passing mechanism. A full implementation of the actor model was not required during this study. But it should not be a problem to implement the missing parts of the actor model like dynamic creation and destruction of processes at run time. The message passing of the actor model was successfully implemented in the system. It works as intended and programmers can easily pass messages transparently between different concurrent processes. The concurrent software components are decomposed by functionality and uses message passing to communicate with each other. This makes it easy to perform component based testing by passing test messages to a component and observing how it behaves. Component simulation is another possible thing which is easily made by creating a simulator interacting with message passing like the original component. The actor model implementation does not care about the programming language used. Which makes it possible to implement a component in any language and make it communicate simply by writing a wrapper for the actor model interface. This is how Lua was integrated with the rest of the system.

All these aspects improve maintainability of the system as a whole if properly used.

We have not been able to do more in depth testing of how the implementation behaves during heavy load. But we have tried to foresee where the load will be and optimized these parts. Also decisions like only passing deltas (no spamming or polling) and the event mechanism decreases the overall load significantly. Collection of message passing statistics (for instance messages sent, received and lost) and process scheduler statistics (for example CPU time) were implemented and can easily be used during debugging to locate for instance bottlenecks in the system.

5.4 Reflecting on the use of Lua

After some minor calibrations Lua executed fine on the embedded touch display device. Lua works as intended and so far we have not discovered any problems in using it together with C in the embedded system. Actually, Lua is a very good complement to C because it does not do what C is good for but the parts where C lacks ease of use, for example, garbage collection and dynamic data types. The footprint of Lua, including all libraries and dependencies, in the system is around 240kB. If required, the size could be decreased by removing unused libraries. The Lua environment is rather quick to load even with larger scripts which is an advantage of Lua. Lua's GC executes as intended and works well with dynamic memory allocations. The fact that Lua uses double as standard data type for all numbers, and that the embedded device does not have an FPU, decreases the efficiency but has not been a problem in our system.

5.5 Lua as an actor

It was very easy to integrate Lua with the actor model (in this case the message passing), only by creating wrappers for the interface. This allows Lua developers to safely communicate with the rest of the system through a simple interface. Sandboxing for third party applications is easy because sandboxing is an inherent feature of Lua. The delays in the message passing mechanism are negligible even for messages passed between two Lua processes.

So far, we have not seen any significant efficiency sufferings from using Lua in some of the system modules. The responsiveness tests performed show that it is fast enough for the touch display purpose.

A software module was also implemented which can receive data from a computer over USB and pass to another process as a message. Messages can also be received and passed back to the computer over USB. This was used to pass Lua scripts to the Lua process and the print method of Lua was mapped to pass back the printout. It made it more convenient to do execution testing while developing without the need of recompiling and flashing the embedded device. Furthermore, Lua scripts can be loaded from an SD-card dynamically, which simplifies updates and software changes compared to reflashing the embedded device.

5.6 Reviewing the hypotheses

— **Lua as a complement to C increases maintainability while retaining efficiency**: Lua as a scripting language increases the productivity of the developers (Ousterhout, 1998; Prechelt, 2002). It also provides a dynamic way of simply alter the application without recompilation. Sandboxing is another strength of Lua which makes it possible to sandbox applications to make them safer to execute. The dynamic data structures and GC makes it easy to manage data, for instance strings, which could be cumbersome in C and comes with a risk of introducing memory leaks. The ability to update the Lua applications at runtime over USB greatly decreases the required time to perform updates during development. No maintainability prediction techniques or models were used. Of the models proposed in literature few are supported with accuracy measures, use any form of cross-validation or have evidence for external validity (Riaz et al., 2009). Due to this, we chose to use guidelines from literature instead of prediction models. Efficiency can be retained where required by implementing parts using C instead of Lua. The responsiveness benchmarks proves that we successfully use Lua to define the system behavior without sacrificing responsiveness.

— **Message passing increases maintainability at the cost of efficiency**: The message passing creates a concurrency transparent and simple interface for developers. It also decreases the coupling of the modules with asynchronous calls and forces modularity which improves maintainability. The actor model can be interfaced from any programming language able to wrap up the C message passing interface. Modularity makes component based testing and module simulation easier which also increases maintainability. Efficiency suffers because it is a level of abstraction and it would have used less hardware resources by calling functions directly. But that would have raised other problems with concurrency, like shared memory, instead.

— **Lua is efficient enough to manage a touch display in an embedded device**: After integrating Lua using actor model the efficiency is still at an acceptable level, supported by our measurements. In the modules of the system where real efficiency is required, it is still possible to use C (or any other programming language fit for the purpose) in the design.

5.7 Further work

During this study we have found an other interesting part of the area which can be researched upon. We have used an RTOS as a scheduler but without any real-time constraints. We also know for sure that our design (with asynchronous message passing and nondeterminism) using Lua (with its garbage collector and dynamic data types) may be risky to use where real-time constraints exists. It may be possible to use this approach and still live up to real-time constraints. An example of where asynchronous message passing is used with real time constraints is Enea OSE[5] (the signals used). Berry (1989) brings up real-time constraints in combination with the actor model, but he does not use a high-order (scripting) language in his work.

6 Concluding remarks

The goal of this research was to increase maintainability in an embedded touch display system while retaining efficiency, in order to meet responsiveness constraints. Through the use of Lua as a complement to C and the message passing semantics of the actor model in the design of a embedded touch display system, we have found that this approach is applicable in order to improve maintainability, without sacrificing responsiveness of the touch display. These findings contribute to the knowledge area of software design and development of embedded systems used for human computer interaction. Our focus has been on quality attributes, specifically efficiency and maintainability, which is likely to be of importance to producers of embedded touch display systems. As further work we suggest to evaluate the use of this approach in embedded systems with real-time constraints.

[5] `http://www.enea.com/`

References

Aggarwal, K. K., Singh, Y., Chandra, P., and Puri, M. (2005). Measurement of software maintainability using a fuzzy model. *Journal of Computer Science*, 1:538–542.

Agha, G. (1986). *Actors: a model of concurrent computation in distributed systems.* MIT Press, Cambridge, MA, USA.

Bass, L., Kazman, R., and Clements, P. (2003). *Software Architecture in Practice.* Addison-Wesley Professional, 2nd edition edition.

Berry, G. (1989). Real time programming: Special purpose or general purpose languages. In *World Computer Congress*, pages 11–17.

Boehm, B. W., Brown, J. R., and Lipow, M. (1976). Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 592–605, Los Alamitos, CA, USA. IEEE Computer Society Press.

Brooks, Jr., F. P. (1975). The mythical man-month. *SIGPLAN Not.*, 10:193–.

Clark, D. L. (2009). Powering intelligent instruments with lua scripting. Technical report, Round Rock, TX 78681 USA.

de Figueiredo, L. H., Ierusalimschy, R., and Celes, W. (1996). Lua: an extensible embedded language. *Dr. Dobb's Journal*, 21(12):26–33. http://www.lua.org/ddj.html.

Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design science in information systems research. *MIS Quarterly*, 28(1):75–105.

Hewitt, C., Bishop, P., and Steiger, R. (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Hirschi, A. (2007). Traveling light, the lua way. *IEEE Software*, 24:31–38.

IEEE/ANSI (1993). *Recommended practice for software requirements specifications. International Standard 830-1993.*

Ierusalimschy, R. (2006). *Programming in Lua.* Lua.org, 2nd edition edition.

Ierusalimschy, R., de Figueiredo, L. H., and Celes, W. (2007). The evolution of lua. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 2–1–2–26, New York, NY, USA. ACM.

Ierusalimschy, R., de Figueiredo, L. H., Henrique, L., Waldemar, F., and Filho, W. C. (1995). Lua - an extensible extension language. *Software: Practice & Experience*, 26:635–652.

ISO (1999). *ISO/IEC 9899:1999: Programming Languages — C.*

ISO/IEC (1999). *ISO/IEC 14598. Software engineering – Product evaluation.* ISO/IEC.

ISO/IEC (2001). *ISO/IEC 9126. Software engineering – Product quality.* ISO/IEC.

Kim, C. and Weston, S. (1988). Software maintainability: perceptions of edp professionals. *MIS Q.*, 12:167–185.

Lieberman, H. (1981). Thinking about lots of things at once without getting confused: Parallellism in act 1. Technical report, Massachusetts Institute of Technology.

Martin, R. C. (2009). *Clean Code: A handbook of agile software craftsmanship.* Prentice Hall.

McCall, J. (1977). *Factors in Software Quality: Preliminary Handbook on Software Quality for an Acquisiton Manager*, volume 1-3. General Electric.

National Bureau of Standards (1984). Guideline on software maintenance. Technical report, FIPS Pub 106, U.S. Dept. of Commerce.

Noergaard, T. (2005). *Embedded Systems Architecture: A Comprehensive Guide for Engineers and Programmers*. Elsevier, Oxford.

Ousterhout, J. K. (1998). Scripting: Higher-level programming for the 21st century. *Computer*, 31:23–30.

Pearse, T. and Oman, P. (1995). Maintainability measurements on industrial source code maintenance activities. In *Proceedings of the International Conference on Software Maintenance*, ICSM '95, pages 295–, Washington, DC, USA. IEEE Computer Society.

Prechelt, L. (2000). An empirical comparison of c, c++, java, perl, python, rexx, and tcl for a search/string-processing program. Technical report, Universitat Karlsruhe, Fakultat fur Informatik, D-76128 Karlsruhe, Germany.

Prechelt, L. (2002). Are scripting languages any good? a validation of perl, python, rexx, and tcl against c, c++, and java. *Advances in Computers*, 58.

Riaz, M., Mendes, E., and Tempero, E. (2009). A systematic review of software maintainability prediction and metrics. *Empirical Software Engineering and Measurement, International Symposium on*, 0:367–377.

Sanner, M. F. (1999). Python: A programming language for software integration and development. *J. Mol. Graphics Mod*, 17:57–61.

Seow, S. C. (2008). *Designing and Engineering Time: The Psychology of Time Perception in Software*. Addison-Wesley Professional, 1 edition.

Sommerville, I. (2006). *Software Engineering*. Addison-Wesley, 8th edition edition.

Vaishnavi, V. and Kuechler, W. (2008). Design research in information systems. *Order A Journal On The Theory Of Ordered Sets And Its Applications*, 48(2):1–393.

Ward, M. P. (2003). Language oriented programming. *Language*, 15(4):1–21.