Lightweight Framework for Runtime Updating of C-Based Software in Embedded Systems

[a]*Simon Holmbacka,* [b]*Wictor Lund,* [b]*Sébastien Lafond,* [b]*Johan Lilius*
[a]*Turku Centre for Computer Science - TUCS*
[b]*Department of Information Technologies, Åbo Akademi University*
Joukahaisenkatu 3-5 FIN-20520 Turku
Email: firstname.lastname@abo.fi

Abstract

Software updates in embedded systems are typically performed by bringing the system to stop, replacing the software and restarting the system. This process can in certain cases be very time consuming and costly, which leads to less frequent software updates. In order to establish both long uptime and up-to-date software, the software must be updated during runtime. This paper presents a runtime updating framework for embedded systems capable of replacing parts of software without stopping the system. The framework is based on FreeRTOS and mechanisms have been added to dynamically link and re-link FreeRTOS tasks to the system during runtime. Our framework enables the programmer to easily create updatable software with simple annotations to the program. Experiments demonstrate the benefits of updating software during runtime with an acceptable overhead when transferring the application state.

1 Introduction

Long uptime and up-to-date software are highly valued properties in many industrial plants, automation devices, complex machineries etc. The production rate of such devices is dependent on the uptime, which means that a long and complex reboot process highly unwanted [18] or in some cases simply not allowed [8]. As software is updatable only during maintenance stops, the running systems will suffer from obsolete software as the maintenance cycle increase. To solve this problem, the system requires a mechanism to replace software during runtime. Runtime updating of software is a technique for replacing a running part of software with another part. The state of the old part is moved to the new part, which continues to execute from the point at which the old part was interrupted. Runtime updating also enables the possibility of transmitting new software versions on-the-fly over a communications media such as the Internet. It reduces costs and eventual risk of having personnel physically updating software in inaccessible or remote locations.

Runtime updating of software has existed for decades and has been previously presented in many forms [15]. Its behavior and requirements for updating varies based on functionality, OS, underlying hardware platform etc. In this paper we present the implementation of a lightweight runtime updating framework for embedded systems. The advantages of our framework are: *a)* small footprint for data and program memory, which makes it suitable for even small scale embedded systems *b)* no steady state execution overhead *c)* easy annotations of application state without including explicit meta data from the programmer *d)* the framework is compatible with all 32bit FreeRTOS compatible hardware and uses the standard GCC without any modifications.

By using the framework, the programmer can create updatable applications as plug-in components, and on-the-fly insert them into the system. The framework is an extension to the operating system FreeRTOS [5] and is implemented using 1600 lines of C-code. Early evaluations demonstrate the benefits of updating software during runtime with regard to performance and functionality, and a low overhead for state transfer in typical applications. The runtime updating mechanism is freely available from [9].

2 Related Work

Runtime updating techniques exist in several forms and on different levels of abstraction. Highlevel mechanisms such as [4] and [20] update Java systems online by using component based solutions such as the OSGi. On levels closer to hardware, a component framework THINK [14] has been used to create updatable components with well defined interfaces. THINK supports re-linking of components during runtime and explicit mechanisms for starting and stopping a component. In contrast to the extensive THINK framework, we address the importance

of a lightweight framework for embedded devices. Our framework is created for small to medium range embedded devices with a restricted amount of memory and limited amount of processing power. The framework can be compiled with the standard GCC and requires no new language learning.

Wahler et al. present in [19] a component framework for enabling runtime updating in real-time embedded systems. Similar to our approach, the state context in the applications is stored in a shared memory area, but instead of tagging the memory area with meta data, we define a new memory section (`.rtu`) to store data related to runtime updating.

An important question for runtime updating is the level of transparency to the programmer, and what implications the update will have on the application. In [12] the updating mechanism uses a global function pointer to dynamically point on the current function version, and changes this pointer value as the software is updated. This framework requires therefore an added version tag to each version of the functions. The framework in [12] also requires a special compiler and certain modifications to the program in order to achieve updatable compilable applications. We solve several of these issues by introducing dynamic linking of ELF binaries and by using simple annotations to describe the state context. In our system, The updated functions are not referred to by their name, but their address in the linked binary.

State transfer [7, 17, 13] enables the updated application to continue executing with the same state as the older version, and is one of the key issues in runtime updating. The approach for implementing state transfer is dependent on the abstraction level the mechanism is operating on. Java-based systems [13] can swap complete classes while C-based systems cannot. Hayden et al. [7] describe the development of a serialization mechanism used to trace the content of each data structure in C-based systems to determine the content of the application state. To trace the state content in a structured way, we created a tracing mechanism which uses debug information derived from the DWARF library [21]. The benefit of using DWARF is that the programmer is not required to specify the content explicitly (such as array size) in data structures being part of the state context.

Lastly we allow the programmer to mark the state context with a special annotation. This approach reduces the transparency slightly, but in contrast to [6] requires no static analysis of the application. It neither puts constraints on the program structure or introduces any steady state overhead because no implicit monitoring is needed during runtime.

3 Implementation

This section describes the most important parts of the implementation of the runtime updating mechanism and is further specified in [10]. We have implemented the mechanism as an extension to FreeRTOS [5] because of the lightweightness, fast response and portability. The runtime updating mechanism is created using roughly 1600 SLOC, and a minimum configuration with OS and runtime updating mechanism requires in total only 700 kB of program memory. To further optimize this size, a hand tailored pointer tracer would need to be implemented. Nevertheless chose the DWARF library due to its rich functionality and matureness. Sections 3.1 and 3.2 focus on the two most important aspects of the implementation: *Dynamic binary linking* and *state transfer*.

3.1 Dynamic binary linking

The first main feature of the runtime updating mechanism is to componentize FreeRTOS tasks into separate Executable and Linkable Format (ELF) binaries. This componentization enables the possibility of re-linking software during runtime and we have implemented the mechanism explicitly for this reason. The feature is not available in the default FreeRTOS package, and has been added to dynamically add and remove software binaries to the system during runtime.

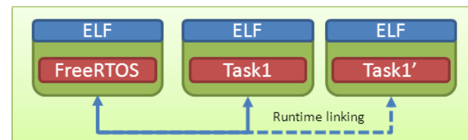

Figure 1: Structure of the ELF binary-based composition

Figure 1 illustrates FreeRTOS and the runtime updating environment using ELF binaries for two parts: The FreeRTOS base system and the tasks schedulable by FreeRTOS. At startup and during runtime, the system binary is linked with all task binaries in the following manner:

1. An ELF binary is inserted into the memory from a given source
2. The system recognizes all tasks and their entry point pointer inside the ELF binary (the pointer is given by the linker)
3. The system registers all tasks to a task manager by passing the entry point pointers to a `task_register()` function

4. System is able to create new FreeRTOS tasks with standard FreeRTOS system calls using the given task pointers

5. The newly created task can be inserted into the FreeRTOS scheduling list and the old task can be removed with standard system calls

After this, the tasks can use kernel resources and the kernel is able to schedule FreeRTOS tasks from the linked ELF binaries. With dynamic linking, tasks can be externally compiled and inserted into the running system from external sources such as ftp, flash cards, etc. and linked into the kernel. To create linkable code, the tasks must be compiled *position independent* and *relocatable* with the GCC compiler in order to be loaded into any address and support run-time linking. More detailed information about dynamic linking in FreeRTOS is found in [10].

Moreover, we have adopted a set of steps to follow in order to update software components. To update running software, the executing task must stop and the new version is re-linked afterwards the component is started. We have followed the algorithm is described in [19] (Section 3.2) with the exception of steps 4 and 5.

3.2 State transfer

A task's state is a task's obtained properties when executing to a certain point in the code. The state is usually represented by a collection of declared variables, pointers or data structures with certain values. The value of the context defining the state must be temporarily stored in a predefined place in order for the new software version to continue with the same state as the old version. Since the framework is aimed for small size embedded system with limited hardware and simple runtime, the state transfer mechanism operates on physical memory addresses. This approach has been chosen because FreeRTOS (and similar lightweight operating systems) does not support virtual memory and is able to run on platforms without MMU.

In our framework, the programmer annotates the context with `_RTU_DATA_` e.g `unsigned int _RTU_DATA_ a;` This annotation allows the variable to be saved in a special data segment called `.rtu`. It gives the program a structured way of storing selected information in an application, and gives the updated version of the program a predictable way of retrieving the information. As the context notation has been placed, the programmer declares *checkpoints* `TASK_IN_SAFE_STATE()` in the program to determine point at which the program is *safe* for update. A safe state is a state in which no external events such as inter task communication, open file descriptors, open sockets, etc., can disturb the state of the task. At this point the updating mechanism is invoked and all annotated variables are stored in the `.rtu` segment. The current state of the lightweight runtime updating library requires the programmer to manually insert checkpoints, and varies naturally with the implemented task. Methods for determining the optimal checkpoint placement is, however, not part of this paper.

As the new version of the program is re-linked with the FreeRTOS system, the state context can be transferred from the temporary `.rtu` segment to the context used in the program. The runtime update is initiated as soon as any actor sets a `rtu_requested` flag high in the application. This actor can be a timer, external event or another task in the system; practically the flag is set in a defined callback function in the task itself. Figure 2 shows two versions of a simple program. The first version of

```
/*Version 1 of the program*/
int _RTU_DATA_ state1 = 0;
int main()
{
rtu_requested = 0;
 while (1) {
  if (rtu_requested) {
   /* Go into safe state */
   TASK_IN_SAFE_STATE();
  }
  state1 += 1;
 }
}
```

```
/*Version 2 of the program*/
int _RTU_DATA_ state1 = 0;
int main()
{
 rtu_requested = 0;
 while (1) {
  if (rtu_requested) {
   /* Go into safe state */
   TASK_IN_SAFE_STATE();
  }
  state1 += 2;
 }
}
```

Figure 2: Two versions of an updatable program

the program contains a counter which increments by one unit each loop iteration. After the `rtu_requested` flag is set, the runtime update mechanism transfers the counter value to the new version. The new version of the program continues with the stored counter value, but increments this value with two units. In case the type of the context changes (e.g. from `int` to `float`) the application must perform an explicit state transform on the context [19].

3.2.1 Pointer tracer

Our framework supports the usage of pointers as part of data structures in the state context. In order to determine what data to include in the context transfer, we introduced debug information to the compiler and created a pointer tracer to collect information of all variables. For this purpose we use the DWARF debug library [21] for GCC to mark the content of memory references according to data type. DWARF injects debug information regarding memory sections into the code. The sections are then labeled according to what type the content is representing e.g. integer, float or pointer. The debug information is stored in `.debug` memory segments at which our pointer tracer is able to retrieve the information.

The pointer tracer is able to follow pointer references down to the leaf node and extract information along its path using the stored debug information. It also marks

an already searched reference in order to avoid infinite search loops. With this function, our framework is able to determine and transfer data types such as atomic variables, pointers, arrays, arrays of pointers, structs etc. without explicit information from the programmer.

4 Evaluation

To demonstrate early results, we performed a set of experiments with the runtime updating mechanism on top of FreeRTOS. All experiments used a timer to trigger the runtime updating mechanism at a given moment.

4.1 Platforms

The evaluation was run on two embedded platforms with different architectures.

The first platform was a Versatile Express board [3] equipped with an ARM Cortex-A9 based CA9 NEC CoreTile 9x4 quad-core chip running at 400 MHz with 1 GB of DDR2 main memory. This ARM device uses the ARMv7 architecture and was compiled with the ARM instruction set. Moreover, we created a port [1] of FreeRTOS for the Versatile Express, which was used as the underlying software platform. The second platform used in the experiments was an ARM926EJ-S Processor based on the Atmel ARM9 [2] CPU, which was using the ARMv6 architecture. The CPU was running on 200 MHz and had 64MB of external DDR2 main memory. We mapped the runtime updating mechanism on an existing FreeRTOS port designed for the ARM926EJ-S device.

4.2 Software update

We evaluated two different outcomes of using the runtime updating mechanism: *Performance* and *functionality*. Later, we also performed overhead measurements on both platforms to determine at what cost a runtime update can be achieved. All tests were benchmarked against a baseline application (v1), which later was updated (v2).

ARM Cortex-A9 In the first case, a simple controller use-case was created for demonstrating the effects of runtime updating. Our system shown in Figure 3 consists of a common control loop in which a PID controller regulates the frame rate of a video used for entertainment purposes. The plant is a video player which outputs the frame rate value to the controller, which in turn strives to keep it on 40 fps.

The system is also influenced by load disturbances which may alter the frame rate of the video randomly. The task of the controller is therefore to regulate the sleep time between the frames depending on what disturbances

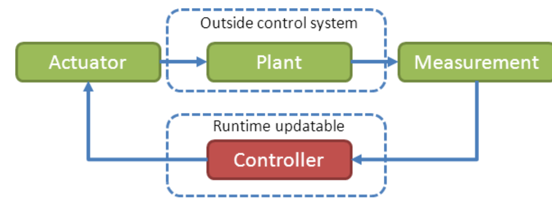

Figure 3: Structure of the video player and its controller

are present at the moment. Initially the system used a normal PID controller (v1). This controller proved to be unable to keep the frame rate stable enough to satisfy the users. Our runtime updating mechanism was implemented to on-the-fly update the PID controller with software modifications (v2). The modifications added features such as minimum variance (MV) to the PID controller, which made it more suitable for the environment in which it is used. The state of the control system containing the I-term and D-term was, as described, stored before the update and restored after the update.

Figure 4 shows the frame rate output of both controllers, and the runtime updating is taking place at 115 samples. From Figure 4, it is clear that the new controller (v2) is more suitable to regulate the frame rate. Hence, superior functionality can be achieved by adding the notion of runtime software updating.

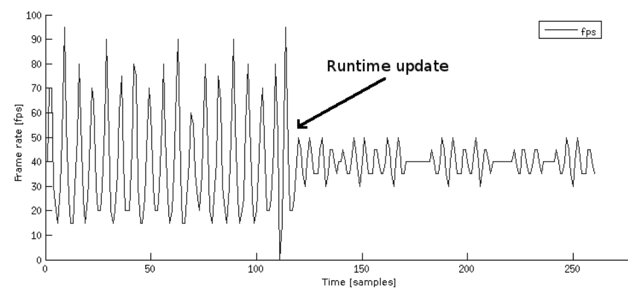

Figure 4: Improved stability after updating the controller

ARM9 In the second case, we used the ARM9 platform to demonstrate increased functionality in the application by updating the software. The application in question is a display used to output measurement values from the chip.

The first software version (v1) included two measurement values: On-chip temperature and Internal chip voltage. The software also included two derived values: Max chip temperature and average chip temperature. These values are based on previous sensor values and are therefore part of the state context stored as integers.

Figure 5 shows the result of the runtime update: v2 includes more sensor values such as external temperature, and auxiliary voltage levels. It also displays all temperatures in both Centigrade and Fahrenheit degrees – a fea-

ture added by the new software. Furthermore, v2 of the software uses floats instead of integers for representing certain temperatures. This property is achieved by transferring the state to the newer version and performing explicit state conversion.

```
#-------------Display v1.0-------------#
| Onchip temperature is          47   C |
| Max Chip temperature:          47   C |
| Average onchip temperature:    46   C |
| Internal Chip voltage is       928 mV |
#---------------------------------------#
```

```
#-------------Display v2.0------------#
| External temperature:        64    C |
|                              147   F |
| Chip temperature:            47    C |
|                              117   F |
| Avg. external temperature:   47.2  C |
|                              117.2 F |
| Max Chip temperature:        47    C |
|                              117   F |
| Chip voltage:                928  mV |
| Aux voltage:                 2346 mV |
#--------------------------------------#
```

Figure 5: Runtime update of display software. Version 1 to the left and Version 2 to the right

4.3 Overhead

Updating software requires a certain time for pointer tracing and state transfer and for the linking of ELF binaries. Since the user experience is degraded if this overhead is too large [11], we set up experiments to determine the overhead based on different program configurations. Furthermore, we run the experiments on both the Cortex-A9 platform and the ARM9 platform in order to compare the results from different performance domains.

The overhead was measured as timing values. Timing measurements in FreeRTOS is derived by measuring the OS ticks (time stamp) at two points in the program and afterwards calculate the difference. We defined the overhead as the time difference between: *entering the safe state in the old version* and *start of task execution in the new version*. In order to compare time differences in two different task versions, we stored the time stamp from the old version as part of the context and transferred it to the new task version. The time stamp in FreeRTOS is defined as an `unsigned long int`. The base line for measuring the overhead was a zero size transfer including only the time stamp from the old version. After this reference measurement, we conducted experiments with two settings: 1) *one array with different sizes* and 2) *Different amount of single variables*. All data types in the arrays and variables in the experiments were double precession floats.

The first experiment with one array of different sizes is presented in Table 1. As seen in the table, the overhead from increasing the array size does not influence much since the pointer tracer does not trace every single element, but only the array itself. The second experiment used different numbers of single variables as context and is presented in Table 2. By introducing small but many variables, the overhead increases more and reaches 5 seconds for the ARM9 in the case of 10000 variables. While this overhead is significant, the usual defined context in a task seldom includes this huge amount of variables. Table 2 also shows that a task including up to 100 variables for defining the context introduces an overhead of only 104 *ms* for the ARM9.

ARM9

Array size	3	100	1k	10k	20k	50k	100k
Time [ms]	56	57	57	61	65	79	99

ARM Cortex-A9

Array size	3	100	1k	10k	20k	50k	100k
Time [ms]	11	11	11	12	13	15	18

Table 1: One array overhead

ARM9

Nr. Var.	10	100	1k	2500	5k	10k
Time [ms]	61	104	537	1311	2579	5162

ARM Cortex-A9

Nr. Var.	10	100	1k	2500	5k	10k
Time [ms]	12	15	51	112	215	426

Table 2: Single variable overhead

5 Conclusions and Discussion

We have presented a lightweight runtime updating framework for FreeRTOS based systems. In contrast to related frameworks, we have focused on small size, easy annotation and no steady state overhead – this makes our framework suitable for systems with limited resources and predictability as key feature. The mechanism is able to transfer annotated context between application versions, which makes the update bump-less to the user.

The mechanism can be integrated into a system with real-time constraints as long as timing violations in real-time tasks are not occurring. Timing overheads from updating software must ensure no timing violations in cases a) the updated task uses real-time constraints b) other tasks using real-time constraints are affected by the updating mechanism. A non-preemptive system must guarantee the necessary amount of slack to perform the update according to the state size overhead. On the other hand, high priority jobs in systems with preemption will not be affected by updating lower priority tasks since the updating mechanism does not use critical sections or resources and hence no priority inversion can occur.

We have evaluated the mechanism on two different platforms and according to two of the major aspects: performance and functionality. The overhead for updating

a given context has also been measured, and results indicate an acceptable overhead for the typical use case. Lastly, the increase in overhead showed similar trends on both platforms even if the Cortex-A9 run with the Data cache enabled and the ARM9 did not.

6 Future Work

We intend to further improve the componentization of software by integrating an already created component framework [16] into our system. With a proper component framework [14, 19], the safe state can be more clearly defined and e.g. for explicit communication between tasks. A more rigorous component framework can also more formally define safe states since the control of communication between components are more precisely handled. With a raised level of abstraction, a proper component framework can more clearly define component functionality and is especially useful in a multi-programmer environment.

Acknowledgment

This work has been supported by the Artemis JU project RECOMP: Reduced Certification Costs Using Trusted Multi-core Platforms (Grant Agreement number 100202).

References

[1] ÅGREN, D. Freertos cortex-a9 mpcore port. `https://github.com/ESLab/FreeRTOS---ARM-Cortex-A9-VersatileExpress-Quad-Core-port`.

[2] ARM. Arm926 processor. `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.arm9/index.html`, 2008.

[3] ARM. Coretile express a9x4 technical reference manual. `http://infocenter.arm.com/help/topic/com.arm.doc.dui0448e/DUI0448E_coretile_express_a9x4_trm.pdf`, 2011.

[4] BANNO, F., MARLETTA, D., PAPPALARDO, G., AND TRAMONTANA, E. Tackling consistency issues for runtime updating distributed systems. In *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on* (2010), pp. 1–8.

[5] BARRY, R. *FreeRTOS Reference Manual: API functions and Configuration Options.* Real Time Engineers Ltd, 2009.

[6] CHEN, H., YU, J., CHEN, R., ZANG, B., AND CHUNG YEW, P. Polus: A powerful live updating system. In *in Proc. of the 29th Intl Conf. on Software Engineering* (2007), pp. 271–281.

[7] CHRISTOPHER HAYDEN, EDWARD SMITH, M. H. J. F. State transfer for clear and efficient runtime updates. In *3:rd Workshop on Hot Topics in Software Upgrades (HotSWUp11)* (2011).

[8] KLEINER, P. Satellite provider fixes business-critical error in orbiting satellite. *Boards and Solutions, ECE* (September 2012), 20–23.

[9] LUND, W. Runtime updating library source code repository. `https://github.com/ESLab/rtdl`.

[10] LUND, W. A unified run-time updating and task migration mechanism. Master's thesis, Åbo Akademi University, Turku, Finland, 2012. `https://research.it.abo.fi/projects/recomp/msctheses/WictorLundThesis.pdf`.

[11] MAKRIS, K., AND BAZZI, R. A. Immediate multi-threaded dynamic software updates using stack reconstruction. Tech. rep., 2008.

[12] NEAMTIU, I., HICKS, M., STOYLE, G., AND ORIOL, M. Practical dynamic software updating for c. 72–83.

[13] NOUBISSI, A., IGUCHI-CARTIGNY, J., AND LANET, J. Hot updates for java based smart cards. In *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference* (2011), pp. 168–173.

[14] POLAKOVIC, J., MAZARE, S., STEFANI, J.-B., AND DAVID, P.-C. Experience with safe dynamic reconfigurations in component-based embedded systems. In *Proceedings of the 10th international conference on Component-based software engineering* (Berlin, 2007), Springer-Verlag, pp. 242–257.

[15] SEGAL, M., AND FRIEDER, O. On-the-fly program modification: systems for dynamic updating. *Software, IEEE 10*, 2 (march 1993), 53 –65.

[16] SLOTTE, R. A lightweight rich-component framework for real-time embedded systems. Master's thesis, Åbo Akademi University, Turku, Finland. `https://research.it.abo.fi/projects/recomp/msctheses/Rober_Slotte_Thesis.pdf`.

[17] SSU, K.-F., AND JIAU, H. C. Online non-stop software update using replicated execution blocks. In *24th International Computer Software and Applications Conference* (Washington, DC, USA, 2000), COMPSAC '00, IEEE Computer Society.

[18] TOIVONEN, H. T., AND TAMMINEN, J. Minimax robust lq control of a thermomechanical pulping plant. *Automatica 26*, 2 (Apr. 1990), 347–351.

[19] WAHLER, M., RICHTER, S., AND ORIOL, M. Dynamic software updates for real-time systems. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades* (New York, NY, USA, 2009), HotSWUp '09, ACM, pp. 2:1–2:6.

[20] WANG, T., ZHOU, X., WEI, J., AND ZHANG, W. Towards runtime plug-and-play software. *Quality Software, International Conference on 0* (2010), 365–368.

[21] WORKGROUP, D. D. I. F. Dwarf debugging information format version 3. `http://dwarfstd.org/doc/Dwarf3.pdf`, 2005.