Summary of existing software update strategies for deeply embedded systems.

Prof. Dr. Martin Orehek
University of Applied Sciences Munich
martin.orehek@hm.edu

Abstract: Many different strategies exist for the software update function of deeply embedded systems. The paper starts by collecting important properties to consider when analysing existing strategies and introduces three classes of runtime environments relevant for the update function. Than an overview of existing strategies is given with their pros and cons. At the end a summary table is provided.

Introduction

The Internet-of-Things (IoT) is becoming a reality, almost every new high-tech product today contains at least one programmable microcontroller deeply embedded in the overall system and with new communication standards like IEEE 802.15.3, ZigBee, BT Low Energy IP and PLC just to name a few, they are connected to the world. Experience shows that connectivity not only brings more functionality and flexibility to the system but also inherently creates vulnerabilities as described in the essay of Bruce Schneider [10].

One important part to manage the lifecycle of connected devices is the ability to safely and securely push important software updates to the devices in the field. Software updates are needed to patch vulnerabilities but also to fix bugs in the application itself or improve and extend the functionality.

A software update function as part of an overall device management infrastructure is not specific to a special IoT application and should be designed as an independent reusable standard service. In this paper existing software update strategies for deeply embedded systems are summarized.

Properties to consider for a software update function in deeply embedded systems

Robustness and resilience: Most of the challenges come from the fact that deeply embedded systems are by nature resource constraint devices running in the field. They are often distributed over vast areas difficult to reach. In case of a problem during the update process there function can break, in most cases this leads to a bricked device. Recovery of such devices can be extremely cost intensive, in such cases the update function should be robust and resilient and avoid bricking devices.

Security: The software update function is intended to provide vulnerability patches in the field, so as a consequence this function itself needs to be secure.

Seamless operation: Often deeply embedded systems are expected to operate 24 hours a day, 7 days a week and 365 a year. Updating software in such a scenario requires the capability to update parts on-the-fly keeping a seamless operation of the system.

Energy efficiency, memory efficiency, computation efficiency: The resource limitation within deeply embedded systems have different roots. Even though Moore's law is also applicable to the embedded field, its impact is much slower. The HW cost pressure on high production volumes is much higher than in the PC environment and even cent differences have a considerable impact. In addition deeply

embedded devices are often battery powered or even run on collected energy from the environment (energy-harvesting). Energy efficiency of the overall design is therefore a paramount design goal leading to simple HW designs aiming for single chip solutions with only a few external components. Microcontrollers in use integrate cores ranging from 8 to 32 bit CPUs [4] with core clock frequency ranges between 8MHz and 70 MHz, limiting the available computational power (MIPS). As a single chip solution they integrated 512B-64kB RAM and 10k-512kB flash memory. Typical communication bandwidths for such systems are also very limited (e.g. 10-250kbit/s) compared to traditional internet connected devices like PCs or servers. Memory efficient solutions adopt concepts to reduce the amount of data to be updated. This not only improves the memory use but also has a positive impact on the data to be transmitted and therefore reduces the transmission time and the resulting energy consumption. The limited computational power requires the avoidence of computational intensive parts in the design, for the security concept this means that existing security HW engines available in the design should be leveraged as much as possible to avoid wasting CPU cycles.

Classes of runtime environments relevant for the SW update function

Runtime environments (RE) found in deeply embedded system designs range from bear metal to highly abstract virtual machine solutions. Because SW update schemes can have a runtime environment dependency here three distinct classes are presented which are used during the analysis later on.

Monolithic runtime environment: If the RE is without an integrated update feature it is part of this class of systems. Examples of this class are all bare metal implementations and operating systems like e.g. TinyOS [9], FreeRTOS[1]. Due to the lack of an update function in the runtime system, it requires an additional component like a bootloader or micro programmer to handle the actual updating of the SW.

Runtime environment with dynamic linking and loading function: If a RE provides an integrated update function it is part of this class. Example designs are operating systems like e.g. Contiki [3], SOS [6]. Such environments provide full update capabilities enabling the replacement of application parts as well as services of the operating system itself.

Virtualization or middle-ware layer: If a RE provides a virtualization layer over which the actual application runs it is part of this class. An examples of this class are e.g. Maté [8], VM* [7]. They include update mechanisms but normally for the abstract application and not for the services provided by the virtualization layer.

[1] http://www.freertos.org/

Overview and summary of existing SW update schemes

In this part the different SW update schemes are presented with their pros and cons.

Micro Programmer in RAM [5]

The micro programmer is a very simple piece of code placed in RAM and executed from there. It receives the SW update over a communication channel and writes the received data to flash memory.

Pros:

- Because it runs from RAM the whole flash can be used for the update. The HW device does not need complex memory layouts (e.g. ROM, flash1 and flash2) to avoid problems when programming the flash.
- It is a very simple approach without any dependency from the runtime environment, complete or partial image updates are possible.

Cons:

- The risk to brick the device is very high. If for instance a power loss happens during the update procedure the persistently stored SW most likely will be broken and the device becomes unrecoverable.
- During the update procedure the normal system function is disabled, the operation is interrupted.
- The implementation of security features is difficult or even impossible due to the limited program resources running from RAM.

Bootloader fixed in persistent memory (ROM, Flash) receives update [11, 13, 14]

A bootloader is permanently stored on the device and receives the SW update to be written to flash. The update function is similar to that of the micro programmer but can be extended due to the bigger memory portion available.

Pros:

- Very robust, bootloader is always runnable to recover the system
- No dependency from the runtime environment, complete or partial image updates are possible
- Security features can be used

Cons:

- Memory resources are permanently consumed by the bootloader reducing available memory for actual application (increased costs)
- Normal device operation is interrupted during update procedure.
- Bootloader functions cannot be updated

Application receives and stores update and triggers bootloader [12, 15]

As proposed by the ZigBee alliance in there ZigBee over-the-air (OTA) update cluster [16], the application receives the update data and stores it during normal operation onto flash memory (internal or external) and triggers the update activation process when possible. The update activation process is executed by running the persistently stored bootloader.

Pros:

- Very robust, if enough memory is available to keep multiple SW update versions stored. In case of an error an old version can be activated again.
- Safety is considered in the design, the image can be extended with a 128 bit integrity check (8-bit CRC aligned to 128-bit or 128-bit CBC MAC)
- Security is considered in the design
    - Image can be encrypted AES-128

    - Image and integrity code can be encrypted AES-128
- Existing HW support for security features are resused
- Normal operation during retrieval of SW update
- No dependency from the runtime environment, complete or partial image updates are possible

Cons:

- Normal device operation is interrupted (only) during update activation procedure.
- Bootloader functions cannot be updated

Application receives and stores update and then starts dynamic linker and loader [1, 2]

The firmware consists of a runtime environment with an OS part and an application part. The modular concept allows the update of dedicated pieces even during normal operation. Content of the updates are partial executable binaries which are dynamically linked and loaded into flash on the target. Even services of the OS can be loaded and unloaded. To provide memory location independent loading of the modules, a dynamic runtime linking is required or Place Independent Code needs to be adopted.

Pros:

- Very flexible, every piece of the SW can be replaced
- Normal operation during complete SW update process, no interruption.
- Partial SW updates are possible

Cons:

- Flexibility most likely reduces robustness.
- Safety features need to be adopted in order to handle unexpected situations like missing symbols during linkage or power outage during linking.
- Linking and loading requires resources during the update step
- Requires dedicated runtime engine
- Security features can be implemented on top

Application receives and stores update and runs the application in a virtual environment or middle-ware layer

A virtual machine or middle-ware layer abstracts the hardware and provides well defined high level functions. The application is byte code mapped and executed in the virtualized runtime environment. The memory size required by the application is minimized, therefore an application update is optimized.

Pros:

- Robust, application cannot brick the device
- Application updates are reduced in size

Cons:

- Framework updates (VM, middle-ware) are not considered

Summary table of considered update schemes

	Robustness	service interruption	memory efficiency	runtime dependency	what can be updated?
Micro Programmer in RAM	low	yes, during complete update cycle	complete or partial update	none	everything
Bootloader fixed in persistent memory (ROM, Flash) receives update	high	yes, during complete update cycle	complete or partial update	none	everything, except bootloader
Application receives and stores update and triggers bootloader	high	yes, only during update boot process	complete or partial update	none	everything, except bootloader
Application receives and stores update and then starts dynamic linker and loader	middle	none	only partial executable binaries	OS with dynamic linker and loader	everything
Application receives and stores update and runs the application in a virtual environment or middle-ware layer	high	none	only application byte code	VM is needed	only application

Literature

[1] Dong, W., Chen, C., Liu, X., Bu, J. and Liu, Y. 2009. Dynamic linking and loading in networked embedded systems. *Mobile Adhoc and Sensor Systems, 2009. MASS'09. IEEE 6th International Conference on* (2009), 554–562.

[2] Dunkels, A., Finne, N., Eriksson, J. and Voigt, T. 2006. Run-time dynamic linking for reprogramming wireless sensor networks. *Proceedings of the 4th international conference on Embedded networked sensor systems* (2006), 15–28.

[3] Dunkels, A., Grönvall, B. and Voigt, T. 2004. Contiki-a lightweight and flexible operating system for tiny networked sensors. *Local Computer Networks, 2004. 29th Annual IEEE International Conference on* (2004), 455–462.

[4] Fredriksen, I. and Kastnes, P. 2014. *8 Bit or 32 Bit? Choosing Your Next Design's MCU*. electronic design.

[5] Gatliff, W. 2004. Implementing downloadable firmware with flash memory. *The Firmware Handbook*. Elsevier, Burlington, Massachusetts.

[6] Han, C.-C., Kumar, R., Shea, R., Kohler, E. and Srivastava, M. 2005. A dynamic operating system for sensor nodes. *Proceedings of the 3rd international conference on Mobile systems, applications, and services* (2005), 163–176.

[7] Koshy, J. and Pandey, R. 2005. VMSTAR: synthesizing scalable runtime environments for sensor networks. *Proceedings of the 3rd international conference on Embedded networked sensor systems* (2005), 243–254.

[8] Levis, P. and Culler, D. 2002. Maté: A tiny virtual machine for sensor networks. *ACM Sigplan Notices*. 37, 10 (2002), 85–95.

[9] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E. and others 2005. TinyOS: An operating system for sensor networks. *Ambient intelligence*. Springer. 115–148.

[10] Schneier, B. 2014. The Internet of Things Is Wildly Insecure And Often Unpatchable. (Jan. 2014).

[11] 2011. *AN3262, Application note: Using the over-the-air bootloader with STM32W108 devices*. ST Microelectronics.

[12] 2014. *AN728, Application Note: Over-The-Air Bootloader Server and Client Setup using EM35x Development Kits*. Silicon Labs.

[13] 2013. *AT02333, Application note: Safe and Secure Bootloader Implementation for SAM3/4*. Atmel.

[14] 2014. *AT07175, Application note: SAM-BA Bootloader for SAM D21*. Atmel.

[15] 2015. *AVR2058, Application Note: BitCloud OTAU User Guide*. Atmel.

[16] 2014. *ZigBee Doc. 095264r23, ZigBee Over-the-Air Upgrading Cluster, Revision 23, Version 1.1*. ZigBee Alliance.