Towards Automatic SW Integration in Dependable Embedded Systems

Leandro Batista Ribeiro
Graz University of Technology
Institute of Technical Informatics
lbatistaribeiro@tugraz.at

Fabian Schlager
Graz University of Technology
Institute of Technical Informatics
fabian.schlager@student.tugraz.at

Marcel Baunach
Graz University of Technology
Institute of Technical Informatics
baunach@tugraz.at

Abstract

Embedded systems must support regular software updates, due to changes in legal regulations, improved algorithms, bug/security fixes, etc. Currently, most embedded systems only support updates through full image replacement. This image is a monolithic software statically built, i.e., all pieces of software are integrated at build time. This approach has several disadvantages, such as the need to stop the functionality and reboot the device upon updates, which is unacceptable on many (safety-)critical applications, e.g., in power plants. In this paper, we present the modular architecture used in MCSmartOS, and how it supports dynamic updates at module level. This is a first step to enable automatic integration on embedded devices. However, many embedded devices are classified as "dependable systems", which must satisfy a set of functional and non-functional properties (FPs/NFPs) w.r.t. real-time, safety, security, and maintainability. Thus, before integrating a new module into the software stack, it is necessary to ensure that the target device still satisfies all required properties after the update. Therefore, the automatic integration must include a pre-validation, which we named Compatibility Check (CC). CC performs various operations: from simple ones, such as checking if a device has enough memory to store the new software, to complex NFP checks, such as schedulability analysis and deadlock detection. Due to the wide range of concepts involved in CC, only some aspects are covered in this paper; others will be discussed in future work. Since some of CC's operations require so much performance or memory, many embedded devices cannot afford to execute them. Hence, we also present a client-server update protocol that distributes update operations between the embedded devices (clients) and high-performance servers that provide the updates; the more resource-constrained a device is, the more operations it outsources to the server.

1 Introduction

Embedded systems must support regular software updates, due to changes in legal regulations, improved algorithms, bug/security fixes, etc. Currently, most embedded systems only support updates through full image replacement. This image is a monolithic software statically built, i.e., all pieces of software are integrated at build time.

Full image replacement is a very simple update mechanism, but it presents several drawbacks. For example, (i) even when just one line of code is modified, the full software must still be built; (ii) devices must always interrupt their normal operation and reboot upon updates, which can be unacceptable on many (safety-)critical applications, e.g., in power plants; (iii) a high amount of data (full image) must be transmitted to devices, which can drain battery-powered devices and increase the network load.

Furthermore, this approach requires a central trusted party, which gathers and integrates all necessary pieces of software. While this is the state of the art, e.g. in the automotive domain, future embedded systems will offer more freedom for customization, where system administrators or users can install software from, e.g., repositories of different SW providers. In this scenario, two factors complicate the centralized approach: (i) proprietary software is not made available to other SW providers. In such cases, it is impossible to test a module against other providers' software; (ii) more customization leads to more variants. With billions of devices out there, it will be unfeasible to build a monolithic image for every variant.

Therefore, we see the need to develop methods to enable devices to dependably perform *automatic integration* of software, i.e., the devices must be able to receive a piece of software and incorporate it into the system without disrupting its normal operation or violating any of the dependability properties. Two requirements must be satisfied to achieve this goal: the first one is an embedded OS that supports dynamic updates, so that a device's software can be partially modified instead of requiring full image replacement; the second one is a mechanism to evaluate if an update would lead to violation of any desired NFP.

To offer support for dynamic updates, we modify MCSmartOS [6] to add the ability to add/update/remove modules without interrupting the normal operation of the devices (see Section 2). There exist general approaches with finer granularity (and consequently higher management overhead), e.g.,

International Conference on Embedded Wireless Systems and Networks (EWSN) 2020
17–19 February, Lyon, France © 2020 Copyright is held by the authors.
Permission is granted for indexing in the ACM Digital Library
ISBN: 978-0-9949886-4-5

updating single functions or variables. However, we need low management overhead, so that our solution is also suitable for resource-constrained devices.

MCSmartOS provides basic OS features, including a timeline, events, resource management and protection, interrupt handling, and a preemptive and priority-based scheduler for concurrent tasks with different real-time requirements. While these features provide flexibility for SW development, they are the reason why automatic integration is so challenging. For example, an update could disrupt a real-time system in case a newly added task uses a shared but exclusive resource for too long, causing other tasks to miss their deadlines. Similarly, an update could introduce a task with high energy consumption, which would cause a battery-powered device to die before the planned time.

To evaluate NFPs upon updates, we envision a mechanism called *Compatibility Check (CC)*, as depicted in Figure 1. CC is divided in two major operations: *pluggability check* and *interoperability check* (see definitions in Section 1.1). Pluggability check within modular MCSmartOS is discussed in Section 3.4; the interoperability check will be addressed in future work, due to its complexity.

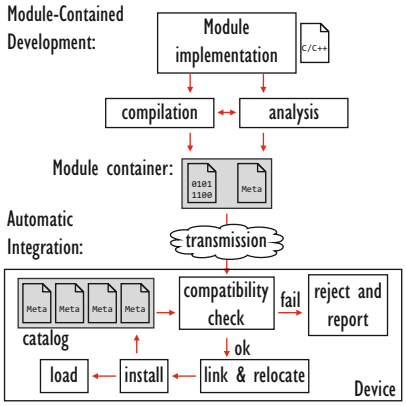


Figure 1: Ideal concept of independent module development and automatic integration.

Figure 1 shows an ideal concept, in which modules are independently developed, and automatically integrated by the embedded devices. At build time, an analysis step runs in parallel with the compilation, in order to generate the module meta-information. CC requires this meta-information, plus the meta-information describing the device's current characteristics: the *catalog*, which is a collection of meta-information of installed modules and HW/OS properties (CPU frequency, scheduling algorithm, etc). If a module is compatible, the update is accepted and its meta-information is added to the catalog. Otherwise, the update is rejected and the reason of failure is reported. CC is based on meta-information, because oftentimes software providers do not share source code with clients/partners, in order to protect their intellectual property. The meta-information generated at build-time shall provide enough knowledge for the CC, without exposing the SW provider's intellectual property.

It is impossible to implement the described concept for every embedded device, because many of them are resource-constrained (and often battery-powered), so (i) they do not have enough memory to store all the required information to perform CC, or (ii) they would need too much time or too much energy to perform CC. Nonetheless, it is still desirable to develop an unified concept, which covers all embedded devices. To achieve this goal, we propose a client-server update protocol that distributes the update operations between clients (devices) and servers (see Section 3.3), which are high performance computers that act as software repositories; they can be, e.g., part of the IoT infrastructure, located in the cloud.

In addition to module transmission, five operations must be performed during updates: CC, linking, relocation, installation, and loading (see definitions in Section 1.1). Installation and loading are the only two operations that must run on the embedded devices. All the others can be performed either by servers or by devices, as long as the required information is available, as detailed in Section 3.

1.1 Terms Definitions

In this work, we use some terms that are not unanimously defined in the literature (e.g., module), and others that are associated with our envisioned concept (e.g., pluggability). For the sake of clarity, the relevant terms for this work are defined below, in alphabetical order.

Application: software that performs a set of tasks to satisfy the requirements of the end user/customer. It is composed of one or more modules. The source code of application modules is HW-independent, but it can make use of HW-dependent modules, such as drivers.

Compatibility Check (CC): pluggability + interoperability checks.

Dependencies: OS and other modules (drivers, libraries, etc.) required by a module, including their corresponding versions.

Hard Deletion: erasing the content of a portion of memory.

Installation: write an already linked and relocated module into ROM.

Interoperability Check: it is satisfied if the update does not violate any NFP. This is key to support dynamic updates on dependable systems, since dependability is a set of NFPs.

Linking: resolution of a module's external symbols.

Loading: initialization of a modules' RAM and OS data structures for execution.

Module: Piece of software independently developed against the target system's interface, but with no information about its implementation (black box model). A module contains zero or more tasks. For example, within MCSmartOS, drivers and libraries have no tasks; services and applications have one or more tasks. Modules are isolated from each other, but tasks within a module are not.

OS: in modular MCSmartOS, the OS is the minimal software to allow a system to run, and to support partial updates. It is composed by the kernel, startup code, interrupt vectors and essential modules (e.g., sysclock and network stack).

Pluggability Check: upon module addition/update, it is satisfied if the device offers enough memory to install the new module, and if all dependencies (with suitable versions) are present. Upon module removal, it is satisfied if no remaining

module depends on the one being removed.
Relocation: resolution of a module's internal symbols.
ROM: shorter notation for flash/program memory.
Soft Deletion: mark a portion of memory as deleted, without erasing its content.
Task: MCSmartOS' basic schedulable unit, analogous to processes in UNIX systems. Every task has a name, a base priority, a dedicated stack, and an entry point. MCSmartOS builds a Task Control Block (TCB) for each task in the system.

1.2 Paper Scope and Organization

The scope of this paper is to present a framework (Figure 1) that prepares an embedded system to support dynamic updates and perform automatic SW integration for provable dependable systems. We focus on:

- how MCSmartOS supports dynamic modular updates (Section 2);
- which operations are performed in our proposed update mechanism, and how our update protocol copes with any connected embedded device (Section 3).

A holistic solution also involves many other topics, such as generation of meta-information, data transmission, authentication, fault-tolerance techniques, etc. We are aware of the importance of such topics, but we do not discuss them in this work, since they are orthogonal to the concepts we present, i.e., they do not influence our design/implementation decisions.

CC is partially covered in this work. We discuss pluggability check in the context of MCSmartOS, and leave interoperability check for future work, due to its complexity and wide range of involved topics.

The remainder of this paper is organized as follows: in Section 2 we describe the related OS design decisions and give an overview of mechanisms used by MCSmartOS to support partial updates. In Section 3 we show how our update protocol copes with the high variety of embedded devices. In Section 4 we analyze the memory and processing overhead in devices, during linking, relocation, or pluggability check. In Section 5 we present related work and highlight how our work is different. Finally, in Section 6 we summarize this paper and present open questions and future work.

2 The Architecture

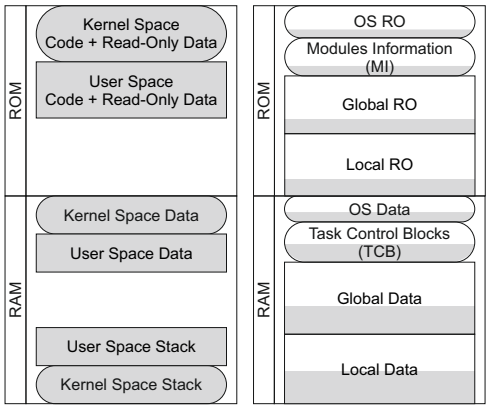
Our proposed architecture supports dynamic updates, i.e., it offers mechanisms to update, add or remove modules without interrupting the operation of the remaining system. In contrast to existing approaches for embedded systems, where such a re-integration is done offline and mostly manually, our goal is to enable automatic self-integration by each target device. However, in order to maintain the dependability of the resulting system stack, additional management and validation effort is required in the device. This introduces significant overhead in comparison with monolithic software, which is maintained offline.

In this section, we present our architecture memory layout, management mechanisms and data structures. Furthermore, we compare our modular approach with the monolithic one.

2.1 Memory Layout

We partition the memory in logical regions, which are initially not fully used; the free memory within the regions enables the addition or update of modules at runtime.

Figure 2 shows how MCSmartOS distributes the software in memory on the monolithic and modular approaches. Gray areas represent used memory, and white areas represent free memory. The rectangles represent regions that belong to user space (e.g., applications and libraries), and the capsules represent regions that belong to kernel space (e.g., OS scheduler and data structures). The isolation of kernel and user spaces is a logical concept of MCSmartOS; effective isolation requires HW support.



(a) Monolithic layout: memory regions are full, and cover part of the memory space.

(b) Modular layout: memory regions are not full, and cover the complete memory space.

Figure 2: Comparison of memory layouts in MCSmartOS.

The purpose of each region is described below, and the way they are populated upon updates is shown in Section 3.

- **OS RO**: This region stores the read-only part of the OS (i.e., code and read-only variables), which must be accessed/executed only in kernel mode. The scheduler, Modules Manager (MoM) and Memory Manager (MeM) are examples of code stored in OS RO.
- **Modules Information (MI)**: Region where MCSmartOS stores the data structure used by MoM to keep track of the modules in the system.
  Details are described in Section 2.2, where the operation of MoM is explained.
- **Global RO**: Code and read-only variables that are visible globally, i.e., that can be accessed by any module in the system. Libraries and drivers compose this region.
- **Local RO**: Code and read-only variables accessible only by the module they belong to. Application modules compose this region.
- **OS Data**: Read-write data that belongs to the OS (data, bss, and stack). This region is only accessible in kernel mode.

- **TCBs**: State of every task in the system, with one TCB per task. MCSmartOS keeps also other control blocks, e.g., for resources and events, but they are not depicted in Figure 2, since they are not crucial to the presented concept.
- **Global Data**: Read-write data globally accessible, in user or kernel mode. Data and bss sections of libraries and drivers compose this region.
- **Local Data**: Read-write data accessible only by the module they belong to. Besides data and bss sections of applications, this region stores the contexts and stacks of all tasks in the system.

2.2 Modules Manager

The ability to add or remove software modules at runtime can potentially generate a huge number of software variants, especially since each device can contain a different combination of modules, due to the possibility of high customization.

In order to retain the dependability of a system, updates are only applied if they are compatible with the target system. CC must test properties that depend on the current combination of modules. Therefore, it is necessary to keep track of the installed modules, and that is the role of the Modules Manager (MoM).

MoM is part of the operating system, and relies on the information stored in the MI region, which belongs to the kernel space. MoM therefore runs in kernel mode.

Figure 3 shows the data structure of the memory region MI. It starts with a header, consisting of an integer N, stating the maximal number of modules supported by the system, and 4N status bits (4 bits/module), storing state information of the N modules and of their respective Load Information (LI). Finally, the N LIs complete the MI region. Table 1 lists and describes the fields of a LI.

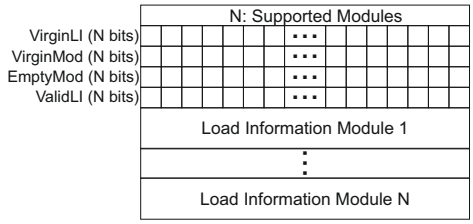


Figure 3: Data structure of MI region.

Table 1: Load Information (LI) fields

Field	Description
Module_ID	Unique identifier for the module, including its version.
TaskCtr	Number of tasks contained in the module.
Base_ROM	Base address where the module is stored in ROM.
Base_RAM	Base address where module's data and bss are loaded.
Size_TEXT	Number of bytes of text segment.
Size_DATA	Number of bytes of data segment.
Size_BSS	Number of bytes of bss segment.
Size_Stack	Number of bytes required for module stack: sum of stacks required by every task in the module.

Three of the status bits are modified during installation, which starts after a module is completely received by the device, and places it in Base_ROM (see Figure 10). Figure 4 shows when VirginLI, VirginMod, and EmptyMod status bits are modified during installation. These three bits are required to persistently track the installation process, i.e, keep the installation status even upon power outage or reboot. The installation is concluded when EmptyMod is zeroed, indicating that the module is already in ROM.

With these 3 bits, it possible to identify if an installation failed on step 1 or 2. This is important for MeM: if there is a failure on step 1, MeM can immediately mark the the memory region starting at Base_ROM as free again; with a failure on step 2, the region needs to be erased before it is marked as free.

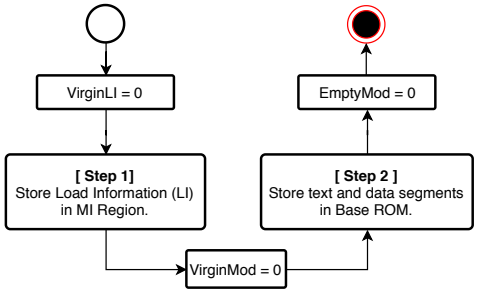


Figure 4: Module installation.

The remaining status bit, ValidLI, is used to indicate that a LI must not be used, either because the corresponding module was uninstalled, or because the data is corrupted. When a module is uninstalled, its content is not erased from ROM. Instead, the corresponding ValidLI is zeroed. When MoM detects installation failures, it zeroes ValidLI, indicating that the module is corrupted (see Figure 5).

In other words, when ValidLI is zeroed, a *soft deletion* is performed, because the memory is only tagged as invalid, but the content of the module is not deleted. Soft deletions avoid the overhead of erasing flash memory.

Table 2 summarizes the meaning of the status bits, and Table 3 lists all possible states these bits can represent.

Table 2: Modules Information status bits

Bit	= 1	= 0
VirginLI	LI in initial state.	LI modified.
VirginMod	Module ROM in initial state.	Module ROM modified.
EmptyMod	Module not yet in ROM.	Module already in ROM.
ValidLI	Valid LI.[a]	Corrupted LI or deleted module.

[a]Either LI is empty or it refers to a correctly installed module.

Figure 5 shows the startup procedure: MoM iterates over the status bits to load all correctly installed modules , i.e., with ValidLI=1 and EmptyMod=0 (see Figure 12 for loading procedure). Upon detection of installation failures, MoM zeroes ValidLI to mark the module as corrupted, and indicates that the update must be requested again. Invalid modules (ValidLI=0) are simply skipped.

The startup procedure finishes when MoM finds the first virgin LI. Therefore, all subsequent LIs must also be vir-

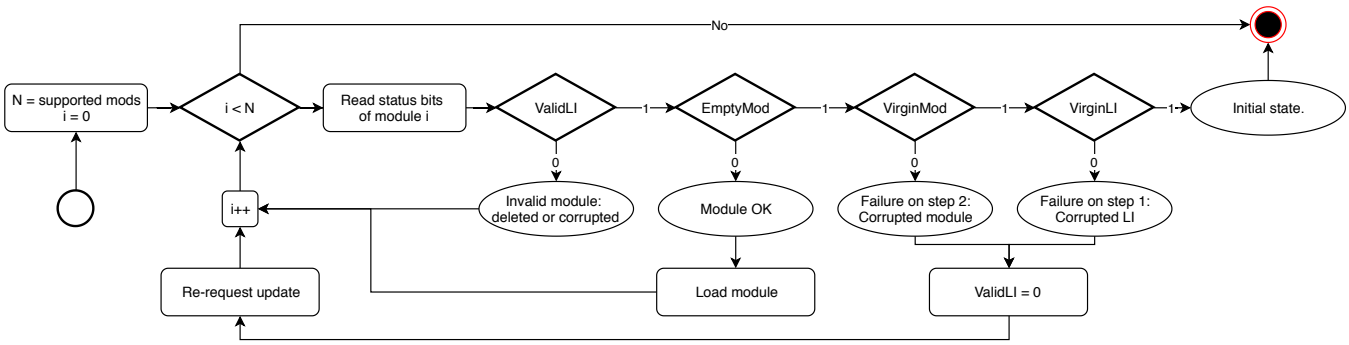


Figure 5: MoM procedure on startup.

Table 3: States of a module with VirginLI-VirginMod-EmptyMod-ValidLI configurations.

Config.	State Description
1-1-1-1	Initial state, empty memory (only 1's).
0-1-1-1	Ongoing installation[a]: writing to LI; else: installation failed while writing to LI.
0-0-1-1	Ongoing installation: writing to module ROM; else: installation failed while writing to module ROM.
0-0-0-1	Installed module.
0-0-0-0	Deleted module.
0-1-1-0	Corrupted LI: MoM detected failure from step 1, and marked the LI as invalid.
0-0-1-0	Corrupted module: MoM detected failure from step 2, and marked the LI as invalid.
Others	Not applicable.

[a]Installation process shown in Figure 4.

gin. MCSmartOS assures this property by updating just one module at a time, and using LIs sequentially. Additionally, MI defragmentation assures that all virgin LIs are in the end of the region (see Figure 6). In case no virgin LI is found, MoM iterates over all LIs.

To conclude the description of Figure 5, the more modules are installed, the longer the startup procedure lasts. The influence of invalid LIs is negligible, since they only increase the number of iterations necessary to load all modules, but do not add any extra operation.

2.3 Memory Manager

As shown in Table 1, LIs have a predefined structure, and consequently the same size. Thus, given the MI region size, it is easy to calculate how many LIs fit, i.e., how many modules are supported. Software modules, on the other hand, have different sizes, and the maximum number of modules that can be installed in a device depends on their individual memory requirements. Therefore, more sophisticated procedures are required, which are performed by the Memory Manager (MeM).

Like MoM, MeM is part of the operating system and runs in kernel mode. The memory management is performed with the following procedures:

Memory tracking: at any point in time, MeM must know how much ROM and RAM is available in the system as well as where free memory is located. It does not need to keep track of which portions of memory are assigned to which modules, since this information is managed by MoM.

Module memory allocation: upon updates, MeM checks for free ROM and RAM to store the module. In case there is enough space, it provides Base_ROM and Base_RAM (see Table 1) for the new module, otherwise MeM rejects the update due to insufficient memory. More specifically, these base addresses are provided in case there are three portions of contiguous free memory: in ROM, (i) at least (Size_TEXT + Size_DATA) bytes; in RAM, (ii) at least (Size_DATA + Size_BSS) bytes, and (iii) at least (Size_Stack + TaskCtr*Size_CTX) bytes. Size_CTX is the size of a task's context, which is stored in the task's stack by MCSmartOS.

Garbage collection: every invalid LI and the respective module ROM are garbage in the system, since they are not used anymore, but still consume memory. While the RAM of a deleted module can be immediately reused, its ROM must first be erased, and only then be used again. MeM is responsible for erasing the ROM of invalid modules. The actual erasure of ROM is called *hard deletion*, in contrast with soft deletion, which only marks a module as invalid, but keeps its content in memory. An important point is that LI hard deletion must always be performed in conjunction with defragmentation, in order to assure that all virgin LIs are adjacent in the end of the MI region, otherwise the startup procedure shown in Figure 5 would fail.

MI defragmentation: a defragmented MI has all virgin LIs consecutively placed in the end of the region, as shown in Figures 6(a) and 6(c). It allows both faster startup (stop at first virgin LI) and efficient search for the next virgin LI (e.g., with binary search). Figure 6(b) shows a fragmented MI, where virgin and valid LIs are interleaved. This configuration must never be in ROM; it is shown only for the sake of illustration, to show that invalid LIs become virgin upon hard deletion. In fact, based on Figure 6(a), the content of Figure 6(c) must be built in RAM, then written back to ROM.

2.4 Comparison with Monolithic Approach

Both the monolithic and the modular approach divides the system memory into regions, as shown in Figure 2.

In the monolithic approach, the amount of memory used by the software is constant during its whole life cycle. Replacing the full software with a new monolithic image is the only way to change the memory configuration and regions.

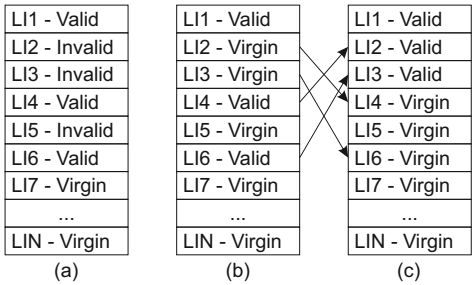


Figure 6: Hard deletion and defragmentation in the MI region.

In the modular approach, the amount of memory used by the software changes upon every partial update. Therefore, regions are dimensioned with free space, to allow the addition or update of modules, as described in Section 3. Since such modifications change the memory content in a non-deterministic way, extra management is required, namely:
(i) tracking of installed modules, performed by MoM;
(ii) tracking of available memory, performed by MeM.

These two items lead to both memory and processing overhead. The memory overhead is constant (the larger the software, the lower the relative overhead), and defined by:

$$Overhead_{mem} = size(MeM) + size(MoM) + size(MI)$$

where *size(X)* is the amount of ROM plus the amount of RAM required by X. The processing overhead, on the other hand, cannot be easily quantified. During updates, the modular approach transmits less data (module instead of full image), but it requires the device to perform tasks that are inexistent in the monolithic approach (see Figure 1). Furthermore, modular code often does not call functions directly, but through some level of indirection, e.g., jump tables. The more often code with indirections is executed, the higher the processing overhead in relation to the monolithic approach (with direct calls).

3 Update Mechanism

In Section 2, we described the design decisions and gave an overview of mechanisms to support partial updates. This section describes the update mechanism in detail, covering the update protocol, module installation, module loading, and involved memory regions in each operation.

3.1 Setup Requirements

Our update mechanism requires the availability of servers, which are resourceful computers acting as software repositories, providing all modules for download.

Servers have enough resources to keep track of which modules are installed in all deployed devices. Thus, they could build new versions of modules without any communication with the devices. However, if privacy is an issue in the future, servers might not be allowed to keep track of installed modules. In this case, a device must provide to the server the necessary information to build a new module tailored for that device. The server, in turn, must discard the information after building the module.

The first scenario (servers keeping track of SW configuration) is straightforward and does not demand elaborated protocols. Therefore, our discussion in this section is based on the second scenario.

3.2 Device Performance Classes (DPCs) and Supported Operations

In order to perform the operations involved in an update, a variety of information must be provided, as shown in Table 4. Furthermore, each operation requires different processing power and amount of memory. Not all embedded devices are able to store all the required information or to process the operations in acceptable time intervals. Therefore, we must adapt the update process to the embedded devices' capabilities.

Table 4: Distributable update operations and required information. Example: updating module M in device D.

Operation	Required Information
Pluggability Check	M's dependencies, modules installed in D and memory configuration in D
Linking	Table of global symbols of modules in D
Relocation	M's base addresses (ROM and RAM) and M's symbol and relocation tables
Interoperability Check	Interoperab. meta-info of M and catalog of D (WCET, WCRT, tasks interactions, etc.)

An arbitrary number of categories can be created, depending on how many operations devices outsource to servers. In the context of this work, we suggest five Device Performance Classes (DPCs) to categorize the devices according to their capability to store the required information and to execute each of the four operations from Table 4. A summary of DPCs and their capabilities is shown in Table 5.

- DPC-0 devices are very resource constrained, often battery powered. They do keep track of installed modules, but do not implement MeM. Therefore, MoM must assume MI defragmentation, and the server must keep track of the memory layout configuration and (optionally) of installed modules. Additionally, the server provides the base addresses for new modules. DPC-0 devices receive these addresses together with the binary, and update the respective LI. Upon updates, they receive only a module's content, without any meta-information (see Figure 8).
- DPC-1 devices are still resource constrained, but they have their own MeM, so they can perform pluggability check. Furthermore, they keep a module's dependency list after installation. This information is necessary to perform pluggability check when removing a module: the device must check if other modules depend on the one being removed. Upon updates, they receive a module's content and its dependency list.
- DPC-2 devices have more processing power and memory than DPC-1, so they can also execute tasks that demand more memory, such as algorithms to relocate modules. Upon updates, they receive a relocatable ELF

file already linked with global symbols, so it only needs to be relocated before being installed.

- DPC-3 devices also offer more memory, so they store the symbol tables of installed modules. This gives them the ability to perform linking as well. Upon updates, they receive a relocatable ELF file, which needs to be linked and relocated before being installed.
- DPC-4 devices have abundant memory and high processing power, so they can afford to store the all the meta-information required by CC, and to execute the complex interoperability check. Upon updates, they receive a relocatable ELF file and its respective interoperability meta-information. All operations are executed at the device's side before installation. The scenario depicted in Figure 1 is possible only with DPC-4 devices.

3.3 Update Protocol

An update can be triggered by the server or by the embedded device. For example, the server can broadcast a module ID, and all devices that contain an older version of that module will request the new one. The devices can also have a periodic update routine, in which they request updates for all installed modules, or they can offer an interface that allows system administrators to manage the modules. For simplicity, in this section we depict the protocol from the the moment the embedded device sends the first message.

The update protocol was designed to support a wide range of devices, from resource-constrained IoT gadgets to high performance systems. This is achieved by performing selected operations in the server when the devices are not able to perform them. The key idea is to tell the server which operations it must perform, and make the necessary information available.

During module request, a device informs the server about its DPC (see Table 5). Thus, the server already knows which operations it must perform, and how the protocol will follow. Figure 7 show the update protocol for DPC-1 devices. The numbered text, above the arrows, identifies the type of message. The text within curly brackets, below the the arrows, shows which information is embedded in the message. Finally, the text within brackets lists the operations performed by the respective side. We present below all the steps and alternative actions in the protocol, using DPC-1 as reference (Figure 7).

Request Module:
As described in Table 1, the *Module ID* uniquely identifies a module, and also encodes the module version. When a device requests the installation of a new module, it sends version 0; for updates, the currently installed version is sent. Thus, the server is able to check if there is a newer version for the requested module. *Architecture* informs the server about the target architecture, and *OS_Info* informs which OS runs in the device, including its version; the suitable module is selected according to the architecture and OS. *DPC* informs the server which operations will be performed by the device, and based on this information, the server knows how to proceed.
Alternative 1: There is a newer version, and the update protocol continues.

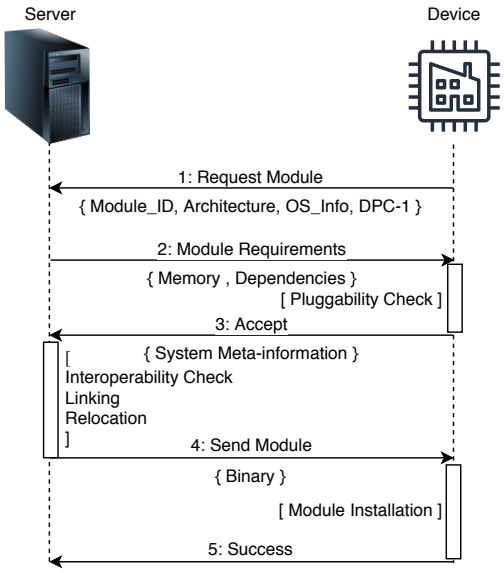


Figure 7: Update of DPC-1 devices

Alternative 2: The module is already up-to-date, the server notifies the device and the update protocol ends.
Module Requirements:
This message contains the necessary information for pluggability check. Within *Memory*, the server informs the device how many tasks the module contains (devices need to check if there are enough free TCBs), and the required amount of RAM and ROM (Size_TEXT, Size_DATA, Size_BSS, Size_Stack). A module usually makes use of the OS and other modules. *Dependencies* contain a list of module IDs and required events and resources, so that the device can check if they are already present in the system.
Alternative 1: Pluggability check is successful, and the update protocol continues.
Alternative 2: Pluggability check finds missing dependencies, and, before proceeding with the current update, the device installs the dependencies.
Alternative 3: Pluggability check fails due to lack of memory, the device reports the error and ends the update protocol.
Accept:
If the device can plug the new module in the system, it informs the server that it accepts and fulfills the module requirements, and sends the necessary information for the operations at the server side: the *System meta-information*. The exact content depends on the DPC. For DPC-1 devices, the system meta-information contains a list of tuples (Module_ID, Base_ROM, Base_RAM), used to relocate and link the modules, and OS parameters (scheduling algorithm, resource management protocol, etc.) for the interoperability check.
Alternative 1: Interoperability check is successful. Then, the module is linked and relocated, and the update protocol continues.
Alternative 2: Interoperability check fails. Then, the cause

Table 5: Overheads and operations of different Device Performance Classes (DPCs).

	Overhead on Devices[a]				Operations performed by				Binary Transmission Step
DPC	**MeM**	**Dep. List**	**Symbol Table**	**Interop. meta-information**	**Pluggab. Check**	**Relocation**	**Linking**	**Interop. Check**	**Data**
0	No	No	No	No	Server	Server	Server	Server	Content[b] + base addresses
1	Yes	Yes	No	No	Device	Server	Server	Server	Content
2	Yes	Yes	No	No	Device	Device	Server	Server	Relocatable ELF linked with global symbols
3	Yes	Yes	Yes	No	Device	Device	Device	Server	Relocatable ELF
4	Yes	Yes	Yes	Yes	Device	Device	Device	Device	Relocatable ELF + interoperability meta-information

[a]MeM adds memory and processing overheads; the others add only memory overhead (they must be kept after installation)
[b]ELF headers and meta-information are stripped (see Figure 8). The content is executable, i.e., fully linked and relocated.

is reported, and the update protocol ends.

Send Module:
In this step, the format of the binary file to be transmitted depends on the DPC. Table 5 lists the binary data for each DPC.
Alternative 1: Installation is successful, and the update protocol continues.
Alternative 2: Installation fails because of data integrity violation (due to, e.g., transmission errors or power outage), and device requests the same module again.

Success:
End of a successful update; this message informs the server that the new module was successfully installed. After receiving it, the server discards the module sent in step 4, since it was tailored for that device.

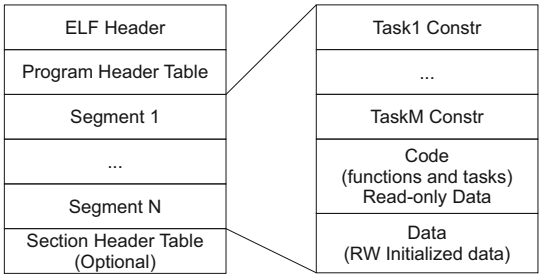


Figure 8: Content of a module: tasks constructors, code and data.

So far, we described the protocol for addition or update. The protocol for module deletion is slightly different, as shown in Figure 9. There is no need for linking or relocation, but CC must still be performed to assure (i) that other modules do not depend on the module being removed (pluggability) and (ii) that all remaining tasks in the system still fulfill their NFPs (interoperability). For the interoperability check, among other information, the server must receive a list of modules installed in the device, so that it can load and combine the corresponding meta-information. This list is all the pluggability check needs. Therefore, the server is able to perform full CC for DPC 0-3 devices.

Optionally, DPC 1-3 devices can perform pluggability check before the *Delete Module* message, in order to avoid unnecessary communication with the server in case the pluggability check fails. DPC-4 devices can delete modules without any communication with the server.

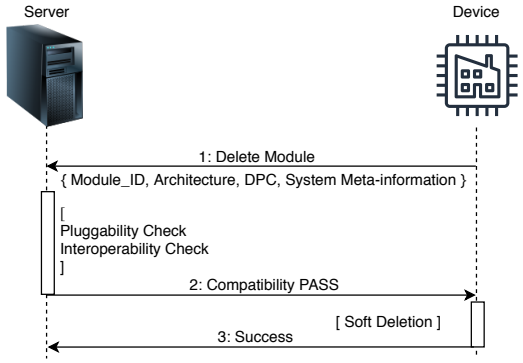


Figure 9: Module deletion on DPCs 0-3

3.4 Pluggability Check

In this section we show what is involved, and which memory regions must be considered during the pluggability check. As described in Section 2.1, applications, middleware (drivers, libraries, etc.), and OS (kernel and essential modules) are placed in different regions. We show how the pluggability check is performed with an application module by referring to Figures from previous sections.

As mentioned in Section 3, a new module is **pluggable** if there is enough **memory** to store and load it, and if all the **dependencies** are present in the target system.

In Table 6 we show how much memory is necessary to store a module. Besides the module content, additional management information must be stored, namely one Load Information (LI) and TaskCtr TCBs. The LI is persistent, and is used to locate and load a module. The TCBs are used by MCSmartOS to properly schedule tasks. MoM is responsible for checking if there is still a virgin LI to store the information about the new module. All other checks are performed by MeM.

Once it is confirmed that the target system fulfills all the memory requirements to store the new module, it is necessary to check if the system contains the modules required by the new one: its dependencies. All modules must be present and their version must be compatible with the ones required by the new module (including the operating system).

Table 6: Memory requirements of an application module

Purpose	Memory Requirement	Affected Memory Region	Managed by	Visualization
Module ROM	Size_TEXT + Size_DATA	Local RO	MeM	Figure 10
Module RAM	Size_DATA + Size_BSS + Size_Stack + TaskCtr*Size_CTX	Local Data	MeM	Figure 11
Management ROM	1 Load Information (LI)[a]	Modules Information (MI)	MoM	Figure 3 and Table 1
Management RAM	TaskCtr TCBs	Task Control Block (TCB)	MeM	Figure 11

[a] Only extra memory requirement in the modular approach. All the others are the same as in the monolithic approach.

The pluggability check is also necessary upon deletion of common modules (libraries, drivers, OS modules, etc.). However, the focus is on dependent modules, i.e., modules that make use of the one being removed. A module cannot be removed in case other modules make use of it, because that would put the system in an unplugged state, which would lead to system crash or undefined behavior. This analysis is not necessary upon removal of application modules, since applications cannot be used by other modules.

3.5 Module Installation

After a positive compatibility check, linking and relocation, a module is ready to be installed. The installation is straightforward: it consists in transferring the module to the appropriate region in ROM. Figure 10 shows an installed application module; it belongs to the "Local RO" region. A driver would be installed in the "Global RO" region, for example. The base address (Base_ROM) and the number of bytes to be copied (Size_TEXT + Size_DATA) are stored in the respective LI.

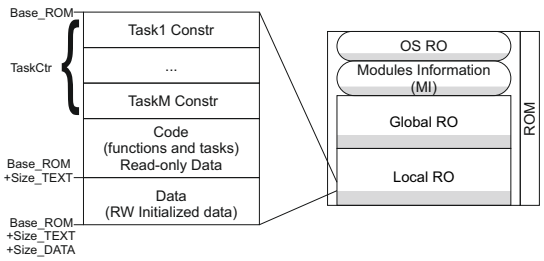


Figure 10: Installed application module

3.6 Module Loading

Every module is loaded on startup, as shown in Figure 5. Additionally, a module can be loaded immediately after its installation. Either way, the process is the same: initializing the module data segment and OS data structures, as depicted in Figure 12 and described below.

Module data: For this step, the module's ROM and respective LI are necessary. The initialized data is stored in ROM (as shown in Figure 10), while the addresses and sizes are stored in the LI. Size_Data bytes are copied from (Base_ROM + Size_TEXT) to Base_RAM, Size_BSS bytes are zeroed, and (Size_Stack + TaskCtr*Size_CTX) bytes are allocated for all tasks' stacks.

OS data structures: The OS initializes the TCBs and stacks of all tasks belonging to the module, and schedules all tasks. As shown in Figure 11, the TCBs are initialized according to the tasks constructors stored in the beginning of the module's ROM.

Figure 11 also shows the RAM content of a loaded application module. The TCB region stores one TCB for each task in the module; the Local Data region stores the data segment of the module (data common to all tasks) and the stacks of each of the tasks. The exact addresses can be retrieved from the corresponding module LI.

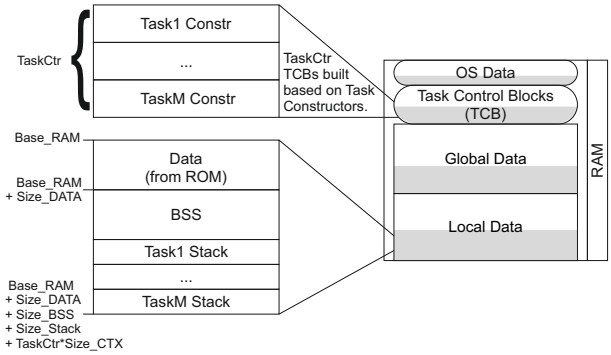


Figure 11: RAM of a loaded application module

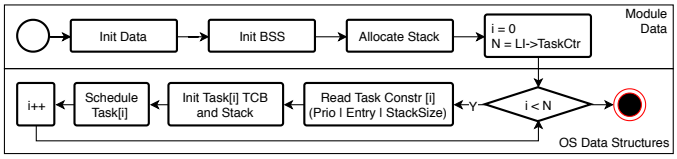


Figure 12: Module loading

4 Analysis and Evaluation

To evaluate our solution, we analyze the update regarding processing time and amount of exchanged data. Furthermore, we compare aspects of the modular and monolithic approaches.

4.1 Goal and Setup

Figure 14 shows our target SW configuration, and respective dependencies. LED-App toggles an LED periodically, and prints a string to the serial port every time the LED is toggled. LED-Drv uses the OS GPIO functionality to operate the LEDs. Initially, only the OS is present in the device; the final state is achieved through an update, which is performed without disrupting the normal operation of the device.

Our proof of concept was implemented on an MSP430F5529 LaunchPad [16], configured as DPC-1,

Table 7: Transmitted bytes and elapsed time during the update shown in Figure 13

Step	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	**Total**
Bytes	7	-	13	-	7	-	11	-	5	-	91	-	1	-	11	-	162	-	1	309
Time (ms)	81.60	0.02	578.86	0.18	81.60	0.002	522	69.6	6.23	18.03	608.11	2.08	0.13	69.58	15.49	22.93	1116.09	3.35	0.13	3195.83

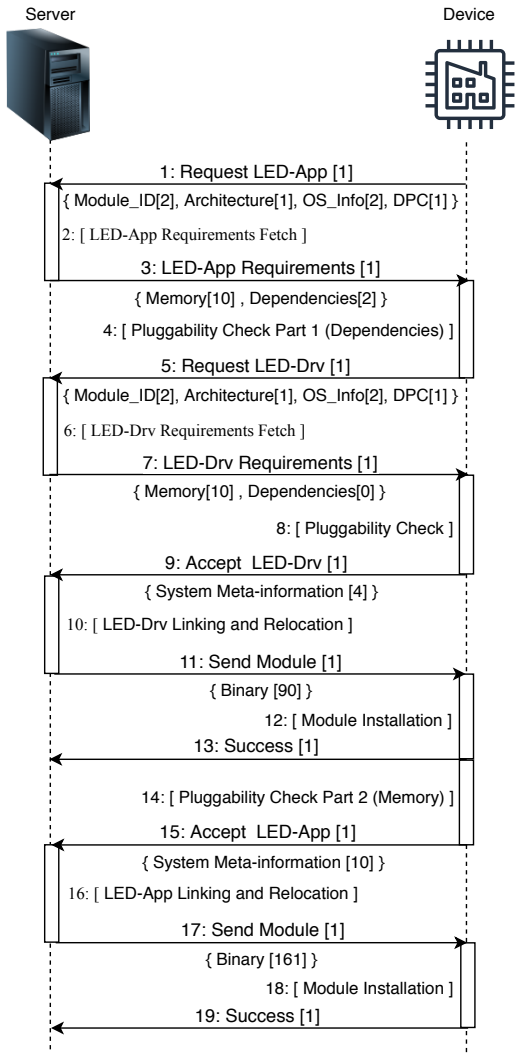


Figure 13: Updating the device with LED-App

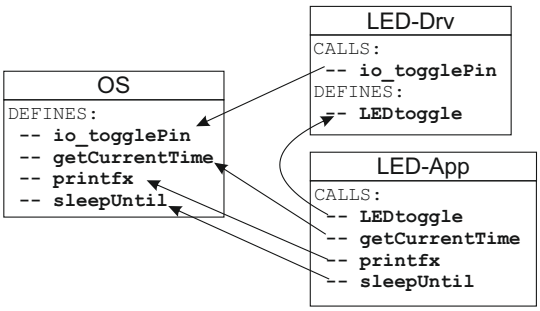


Figure 14: Dependencies between modules and OS

and driven by a 24MHz CPU clock. It offers 128KB of ROM and 8KB of RAM. Our server was an Arch Linux x86_64 machine with an Intel(R) Core(TM) i7-4750HQ CPU @ 2.00GHz and 8 GB of RAM. The MSP and the server communicate via RS232 9600/8-N-1. We compiled the software with `msp430-gcc 7.3.2.154`, optimized for size and without debug information (flags `-g0 -Os`).

4.2 Update

Figure 13 depicts the complete update process. The numbers within brackets describe how many bytes are used to represent the information, e.g, 2 bytes for Module_ID on step 1 (Request LED-App); message types are always represented with 1 byte. In summary, the device requests the installation of LED-App (step 1), but because the dependency "LED-Drv" is not installed (detected in step 4), the device first installs it (steps 7 to 13), and only then proceeds with LED-App installation (from step 14).

On steps 3 and 7 (module requirements), the memory requirements are sent in 5 parts of 2 bytes each: Size_TEXT, Size_DATA, Size_BSS, Size_Stack and TaskCtr. Thus, the device already starts building the LI for the coming module. The module IDs of each dependency are also sent within these steps, i.e., 2 bytes per dependency are sent.

On steps 9 and 15 (Accept), the system meta-information is composed by Base_RAM and Base_ROM of the module being installed (4 bytes, used for relocation), plus 6 bytes per dependency (Module_ID, Base_ROM, Base_RAM), used for linking. Because we do not perform the interoperability check yet, no extra information is needed.

4.2.1 Data Exchange and Timing Analysis

The amount of bytes exchanged and the time spent during the update are shown in Table 7.

The code and data from LED-App and LED-Drv sum up to 251 bytes. During the update, 309 bytes were exchanged. In other words, for our proof of concept, our protocol data transmission overhead was only 57 bytes, or 18.4%.

Table 8 shows that the modular update time was 62.5% faster than the baseline update (3195.83ms against 8537ms). The baseline measures the time the server needs to link all object files to create the monolithic image, plus the time required to flash this image into the device. Since communication is responsible for 94.2% of the modular update time, we could achieve even better results with faster transmission protocols, but this is not the focus of this work. Without the interoperability check, there is nothing to evaluate at the server side, since it performs only basic operations. Therefore, we focus on the device side.

As shown in Table 7, the most time-demanding operations were the pluggability checks (steps 8 and 14). In these steps, MeM allocates memory for the modules. In its current implementation, MeM stores the allocated addresses in ROM, so that nothing is lost in case of reboot. When memory is

allocated, these addresses must be updated, which requires flash erasure before writing. With a MeM that does not require flash erasure, steps 8 and 14 run in less than 2.5 ms each (similar to step 12). With such a MeM, the device would reduce its processing time from 144.61 ms to around 11 ms.

Step 4 is fast because there are no flash writes; MoM simply checks if the dependencies are present. Steps 12 and 18 install the modules by writing into flash memory 91 and 161 bytes, respectively.

Table 8: Time spent on data transmission and processing (server and device sides) compared with the baseline

	Time (ms)	% of Total Update Time	Involved Steps
Communication	3010.24	94.2	1,3,5,7,9,11,13,15,17,19
Server	40.98	1.3	2, 6, 10, 16
Device	144.61	4.5	4, 8, 12, 14, 18
Total	3195.83	100	1 to 19

Baseline (ms)	**Total**	**Flashing**	**Linking**
	8537	8507	30

4.3 Loading Time

After a module installation and before its execution, it needs to be loaded, as depicted in Figures 11 and 12. The load time depends on the RAM requirements, described in Table 6. The load times and memory requirements for our proof of concept are shown in Table 9.

Table 9: Loading time (in μs) and memory requirements (in bytes)

	LED-App	LED-Drv
Loading Time	115.6	8.4
Module ROM	161	90
Module RAM	438	0
Management ROM	15.5 (1 LI + 4 status bits)	15.5
Management RAM	29 (1 TCB)	0

4.4 Comparison with Monolithic Updates

Besides the update time (see Table 8), we compare the startup time (from resetting to scheduling the first task) and data transmission of the modular and monolithic approaches. The reference to our modular software is a monolithic image containing MCSmartOS, LED-App, and LED-Drv.

We measured the startup time of the monolithic software (Monolithic SW), and of the modular SW in its initial state (Modular OS), and after the modules were installed. As shown in Table 11, the startup time in the modular approach is slightly higher than in the monolithic. This overhead is due to the initialization of three items absent in the monolithic version: update task, MeM, and MoM. Nonetheless, the startup time of "Modular OS + LED Modules" is only 0.9% (0.41 ms) slower than "Monolithic SW". Furthermore, the more modules are installed, the lower the relative overhead, since they must be loaded in both approaches; the only difference is that in the modular approach MoM loads the modules (after the OS is already loaded), and in the monolithic approach, OS and modules are loaded by the same startup routine.

Regarding data transmission, a full image replacement would require the transmission of at least 20136 bytes (stripped monolithic image from Table 10). As shown in Table 7, our protocol exchanged 309 bytes, i.e., only 1.5% of the stripped monolithic image size. Besides not disrupting the normal operation of the device during updates, our modular approach for DPC-0 and DPC-1 devices transmits very little data in comparison with the monolithic approach. This feature can, e.g., enable a higher update frequency on battery-powered devices, since the data transmission and flash operations involve less data, and consequently consume less energy.

Table 10 shows that an ELF file has high meta-information overhead, which is more than 80% of our modules, even for the stripped versions. Therefore, by not sending this meta-information during updates, as proposed for DPC-0 and DPC-1 devices, we transmit much less data during the update.

5 Related Work

The idea of performing modular updates in embedded devices is not new. However, most of the solutions tackle the application layer. In 2002, Maté [8] built a virtual machine solution on top of TinyOS [9]. In 2005, SOS [7] was proposed, supporting modular updates at application level using Position Independent Code (PIC) modules. In 2006, [4] added dynamic linking capability into Contiki [5], which enabled modular updates, also at application level, but with no PIC. In the same year, FlexCup [10] was built on top TinyOS. It supports modular updates by splitting the update process in two phases: code generation (at compile time, relevant information is generated) and linking (modified modules are combined with other modules in the devices at runtime). In 2008, FiGaRo [11] and OpenCom[2] were proposed. FiGaRo is implemented on top of Contiki, to handle dependencies checking, version control, and code distribution strategies. OpenCom proposes an extensions layer on top of a minimal kernel layer, to achieve tailorability and extensibility. In 2010, Dynamic TinyOS [12] was proposed. It is similar to Flexcup, but it allows multiple components into a single object. In 2016, GITAR[13] went a step further, and enabled dynamic updates also at network layer.

Our update protocol was partially inspired in SenSpireOS [3], which sends ROM and RAM base addresses to the update server, so that a module can be relocated before it is transmitted. In our protocol, the DPC is used to decide what information must be exchanged and which operations will be performed at the server side.

The works mentioned so far cannot assure dependability, since they offer no mechanism to check if NFPs would hold after the update. Real-time, a subset of dependability, is at least tackled in [15]: upon every update, there is a schedulability test and the update process is assured to be finished within two hyper-periods. However, the target system is required to use rate-monotonic scheduling.

We aim to tackle dependability properties during the interoperability check, which requires the generation and processing of extra meta-information, e.g., Atomic Basic Blocks (ABBs) [14] and COntrol Flow and Interaction Expression

Table 10: Sizes in bytes of monolithic image and modules, and respective meta-information overhead

	Standard Compilation (`-g0 -Os`)				Symbols Removed (`--strip-all`)			
	OS only	**LED-App**	**LED-Drv**	**Mon. Image**	**OS only**	**LED-App**	**LED-Drv**	**Mon. Image**
ELF[a]	123048	6088	5840	92452	23468	836	784	20136
Content-only	20848	161	90	17578	20848	161	90	17578
Meta-information overhead (%)	83.06	97.36	98.44	80.99	11.16	80.74	88.39	12.70

[a] Linked and relocated.

Table 11: Startup time in ms

Monolithic SW (OS + LED SW)	Modular OS (Without Modules)	Modular OS + LED Modules
46.09	46.37	46.50

(COFIE) [1]. ABBs describe the dependencies within a real-time system; COFIEs describe interactions among tasks and their control flow regarding the interaction primitives.

6 Summary and Future Work

In this paper, we showed an overview of our envisioned concept for module-contained development and automatic integration in dependable embedded systems. We presented how MCSmartOS supports dynamic updates and how our update protocol copes with the high diversity of embedded devices. However, the concepts are not restricted to MCSmartOS. Any OS can add our modular update support, and use our update protocol to perform Compatibility Check (CC), alone or in conjunction with an update server, thus achieving the same goal.

The key idea of the update mechanism is that embedded devices outsource selected operations to servers, according to their Device Performance Class (DPC). The lower the DPC, the more resource-constrained the device, and the more operations are performed by the server.

We showed that the meta-information of ELF files takes a considerable portion of a module (97% in our LED application module), and that we can reduce data transmission by not sending this meta-information, since it is not required by MCSmartOS to load modules. Furthermore, we showed that the startup time in the modular approach is only 0.1% higher than the monolithic approach, showing that the extra code required by the modular approach introduces almost no processing overhead on startup.

Regarding improvements, our future work can be divided in two parts: (i) support the update of the OS or shared modules (drivers, libraries and services) without requiring the modification of dependent modules, and (ii) achieving the concept depicted in Figure 1. Our goal is to offer a solution that does not rely on specific HW or compiler features. Furthermore, we will work on efficient mechanisms to handle updates of events and shared resources, and tackle security aspects.

We will also run a detailed comparison between MCSmartOS and other OSs that support partial updates, regarding, e.g., energy consumption and memory footprint.

For the overall concept, we will focus on the interoperability check and further meta-information generation and processing.

7 References

[1] L. Batista Ribeiro and M. Baunach. COFIE: a regex-like interaction and control flow description. In *2019 IEEE Industrial Cyber-Physical Systems*, 5 2019.

[2] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Trans. Comput. Syst.*, 26(1):1:1–1:42, Mar. 2008.

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

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

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

[6] R. M. Gomes, M. Baunach, M. Malenko, L. B. Ribeiro, and F. Mauroner. A co-designed RTOS and MCU concept for dynamically composed embedded systems. In *OSPERT'17*, 2017.

[7] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, MobiSys '05, pages 163–176, New York, NY, USA, 2005. ACM.

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

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

[10] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *European Workshop on Wireless Sensor Networks*, pages 212–227. Springer, 2006.

[11] L. Mottola, G. P. Picco, and A. A. Sheikh. Figaro: Fine-grained software reconfiguration for wireless sensor networks. In *Wireless Sensor Networks*, pages 286–304. Springer, 2008.

[12] W. Munawar, M. H. Alizai, O. Landsiedel, and K. Wehrle. Dynamic tinyos: Modular and transparent incremental code-updates for sensor networks. In *2010 IEEE International Conference on Communications*, pages 1–6. IEEE, 2010.

[13] P. Ruckebusch, E. D. Poorter, C. Fortuna, and I. Moerman. Gitar: Generic extension for internet-of-things architectures enabling dynamic updates of network and application modules. *Ad Hoc Networks*, 36:127 – 151, 2016.

[14] F. Scheler. Atomic basic blocks: eine abstraktion für die gezielte manipulation der echtzeitsystemarchitektur. In *Ausgezeichnete Informatikdissertationen*, 2011.

[15] H. Seifzadeh, A. A. P. Kazem, M. Kargahi, and A. Movaghar. A method for dynamic software updating in real-time systems. In *2009 Eighth IEEE/ACIS International Conference on Computer and Information Science*, pages 34–38. IEEE, 2009.

[16] Texas Instruments. *MSP430F552x Datasheet*, 2009. revised in 2018.