2025 21st International Conference on Distributed Computing in Smart Systems and the Internet of Things (DCOSS-IoT) | 979-8-3315-4372-3/25/$31.00 ©2025 IEEE | DOI: 10.1109/DCOSS-IOT65416.2025.00132

Droopy: A Dynamic Runtime Platform for Micro-Controller Units supporting Partial and Incremental Updates of Modularized Firmware

Huy Dat Nguyen*, Nicolas Le Sommer*, Yves Mahéo*, Lionel Touseau*†
*Université Bretagne Sud, UMR CNRS 6074, IRISA, F-56000 Vannes, France
†Saint-Cyr Coetquidan Military Academy, CReC Saint-Cyr

***Abstract*—Long-range and low-power wireless communication technologies, such as LoRa, Sigfox, myiot and Wi-Fi HaLow, have emerged to massively connect things to the Internet. The ability to effectively update the firmware of these things over the air is becoming a necessity, particularly when resources are restricted.**

This paper presents Droopy, a runtime platform for things powered by MCUs that supports the deployment and execution of firmware designed as a composition of software modules that can express dependencies on other modules. Droopy currently enables partial and incremental over-the-air updates of modules using LoRaWAN, and does not necessarily require a reboot to apply these updates. Modules can be stored in flash memory, or copied to RAM to speed up execution. This paper outlines an implementation of Droopy on STM32 MCUs and details an evaluation of its performance.

I. Introduction

Advances in micro-controllers, sensors, actuators and long range low power communication and cloud technologies have enabled the emergence of the Internet of Things (IoT), with application in various domains, such as smart home, smart agriculture, healthcare, etc. Billion of devices have already been deployed in our daily environment, and billion more will be installed in the future, requiring solutions to facilitate both the remote update and the development of firmware running on these devices. In IoT systems composed of a large number of devices, the firmware update represents a major challenge, especially in Low-Power Wide-Area Networks (LPWANs) where data throughput is limited.

In order to address this challenge, one of the most promising solutions is to develop the firmware as an assembly of pre-existing and reusable off-the-shelf software modules, which can be developed and updated independently. Such a solution will emerge only if it exists embedded runtime platforms than can support the execution and the secure deployment of such modules over the air.

In this paper, we present Droopy, an embedded platform for devices powered by MCUs that provides such features. Droopy stands for "Dynamic Runtime platfOrm for mOdule-Programmed firmware sYstems". Droopy can dynamically load and unload modules, can check their authenticity and integrity, can resolve their dependencies, and can update modules at runtime without necessarily requiring a reboot. In Droopy, modules do not communicate directly with each other, but through proxies, so that Droopy can identify when a module is no longer in use and can be replaced with a new version, or unloaded to reclaim the memory it occupies for other modules. In doing so, it provides some form of isolation between modules. Modules can be stored in flash memory, or copied in RAM by Droopy to speed up the execution of the firmware. In Droopy, a module is described by a set of metadata (e.g., type, identifier, version, provider, dependencies) and usually includes code and data sections that expose variables, functions and routines for other modules. It may also include additional files. The code contained in modules is compiled as position independent code so as to be executed from any address in memory. It is dynamically linked at runtime by the platform.

The rest of the paper is organized as follows. Section II gives an overview of the features and of the architecture of Droopy. Section III describes the different types of modules that are considered in Droopy and their structure. Section IV describes how modules are managed in Droopy. Section V gives implementation details of Droopy on STM32 MCUs and of a firmware update architecture that supports the update of these devices in LoRaWAN. Section VI presents the evaluation results we obtained with Droopy on a concrete scenario, showing that it helps to significantly reduce the network load induced by updates. Section VII compares Droopy with related works. Section VIII concludes this paper by summarizing our contribution and by mentioning suitable features that should be integrated in Droopy in the future.

II. Overview of Droopy

Firmware of IoT devices is usually developed in a monolithic way, introducing constraints and limitations in the firmware update process, the network usage and the power budget and memory usage of devices. Indeed, new firmware must fully be sent on the network in order to be received and stored on a given flash partition by targeted devices that will load this new firmware after a reboot. At least three memory partitions must be reserved: one for the running firmware, another one for the new firmware and a last one for the boot program. Devices whose firmware must be updated must remain up to receive the network packets that contain the firmware fragments, making them consume more energy than necessary, especially if only a small part of the firmware needs to be updated. The network is also unnecessarily overloaded by

2325-2944/25/$31.00 ©2025 IEEE
DOI 10.1109/DCOSS-IoT65416.2025.00132

Authorized licensed use limited to: Universidad Nacional de Colombia (UNAL). Downloaded on November 10,2025 at 19:39:29 UTC from IEEE Xplore. Restrictions apply.

the transmission of the entire firmware. A modular firmware architecture makes it possible to address these drawbacks, by supporting partial and incremental update of firmware through the individual update of the modules that compose the firmware. Not only the update time and the network load can be then drastically reduced, but pre-existing off-the-shelf modules can also be reused for the development of firmware. Providers of modules may trigger their update by multicasting new versions of these modules, allowing different firmware containing common modules to be partially updated simultaneously, without this task being the sole responsibility of the developer who created a given firmware by assembling modules.

A. Features of Droopy

To promote such a development approach, we have designed the Droopy embedded platform, which provides the following features:

- integrity and authenticity checking of modules, based on the module checksum and provider signature;
- dynamic loading/unloading of modules;
- dynamic linking of code provided by modules and compiled as position independent code, so as to be executed from any address in memory;
- memory management, to allocate memory segments to data and code sections of modules, and to dynamically swap module code sections from flash and to RAM for speed up purposes;
- dependency module management, to load modules only when their dependencies are satisfied;
- module isolation, by proxifying the function calls between modules and by providing a loosely coupled binding between modules;
- update functions, to replace a module by a new version while ensuring the compatibility of versions and dependencies.

B. Architecture of Droopy

Droopy, whose general architecture is illustrated in Figure 1, is composed of six main elements, namely a Bootstrapper, a Module Manager, a Dynamic Loader and Linker, a Memory Allocator, an Update Manage and an Update Provider. Droopy can be installed directly on device hardware or be deployed together with an operating system for embedded devices such as RIOT or FreeRTOS.

The bootstrapper (BS) is responsible for installing the main module that serves as the entry point to the application part of the firmware, and for initializing the components of Droopy.

The Module Manager (MM) is in charge of managing modules, resolving their dependencies and managing their life cycle. It relies on the dynamic loader and linker to load and unload modules in and from memory, and to bind, in a loosely coupled manner, the modules together (i.e. to resolve symbol references and to instantiate proxies between modules).

The Dynamic Loader and Linker (DL) ensures the on-demand loading/unloading of modules and resolves at runtime

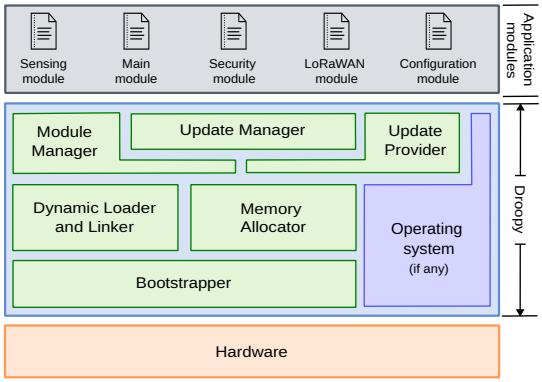

Figure 1. Architecture of Droopy.

symbol references by looking at the symbol table included in each module. Modules can be either executed directly from the flash memory (execute in place) or loaded into RAM. The first strategy aims to optimize the RAM usage by copying only the .data section into RAM. Alternatively, the second strategy enables the module to be entirely copied and executed in RAM instead of the flash memory. In doing so, it speeds up the execution of performance-critical sections of the application. The loading strategy of modules is specified in their properties and is managed by the MM. Our design allows to dynamically disable certain modules or functions by unloading them when they are not used, which improves the memory consumption.

The Memory Allocator (MA) optimizes the memory usage. Our proposition requires to split the flash memory into two main partitions. The first one is dedicated to Droopy ; it contains the operating system (if any), the hardware drivers and device libraries. The second one is used for storing loadable modules, including the main module, data modules and function modules. To each module is allocated a memory block in flash, before deployment, depending on its size. If the module to be updated requires a larger block, the MA looks for a new available memory portion that is suitable for this modified module. This new memory block must be adjacent to a non-empty block to reduce the impact of the memory fragmentation issue. Memory fragmentation occurs when the memory is allocated in a large number of non-contiguous blocks. If the number of non-contiguous blocks is greater than a pre-defined threshold, the main module is stopped and modules are unloaded from the RAM, before the system is rebooted to apply a memory defragmentation algorithm. Then, the main module is reloaded to restart the execution of the application part of the firmware.

The Update Manager (UM) controls the firmware update process. It checks the validity of an update request coming from the UP and, in the case of a valid request, waits for the availability of the the incoming update before verifying its integrity and authenticity based on the checksum and the signature respectively.

The Update Provider (UP) relays update requests to the UM and ensures that an update is properly accessible, and informs the UM when it is so. The behavior of the UM may

Authorized licensed use limited to: Universidad Nacional de Colombia (UNAL). Downloaded on November 10,2025 at 19:39:29 UTC from IEEE Xplore. Restrictions apply.

vary according to the method used to obtain new versions of modules. Typically, a FUOTA method will be applied, as the one we have implemented that receives fragments multicasted over LoRaWAN, reassembled to form a module (see section V for a description of the main features of this FUOTA method).

III. Structure of Modules

This section presents the three types of modules that are considered in Droopy, namely data modules, function modules and main modules. It also shows how modules express their dependencies regarding other modules. Modules are uniquely identified by an ID, a version number and a provider ID. A module ID is defined as a sequence of lower-case characters, separated by dots if needed. The use of dots allows to define a hierarchy of groups of modules. The provider's ID is also defined as a sequence of dot-separated lower-case characters. It is expected to be the provider's domain name. Version numbers are expected to be defined using the traditional version numbering X.Y.Z, where X, Y and Z are respectively the major, mini and micro numbers. Dependencies are noted as a list of module dependencies separated by commas, each module dependency being defined by the ID of the module, the ID of the provider and the version number, separated by a colon. For the sake of illustration, let us consider a firmware installed on a sensor which periodically measures pollution levels in a river and transmits measurements *via* LoRaWAN and The Things Network (TTN)[1]. This firmware is composed of five modules:

- a main module, which is the "root" of the firmware and is loaded, started, suspended, resumed and stopped by Droopy;
- a sensing module: a function module that contains the code for the pollution sensor driver and the measurement functions;
- a security module: a function module that contains checksum, signature and cryptography functions to notably encrypt the measurement values;
- a configuration module: a data module including the encryption keys, the TTN access keys and the configuration properties needed to load modules;
- a LoRaWAN module: a function module that provides functions to process messages that will be sent and received using the LoRaWAN system library conjointly installed with Droopy.

An illustration of this firmware structure and of the manifest of its modules is given in Figure 2.

Modules are composed of a manifest, a *.data* section, a *.code* section and a symbol table. As shown in Figure 2, the manifest defines the ID, the version number and the provider ID of the module. It also specifies the type of devices on which the module can be installed, the dependencies of the module, as well as the signature and checksum of the module to check its authenticity and integrity. Sections *.data* and *.code* define the memory organization of the compiled code of the module.

[1]https://www.thethingsnetwork.org/

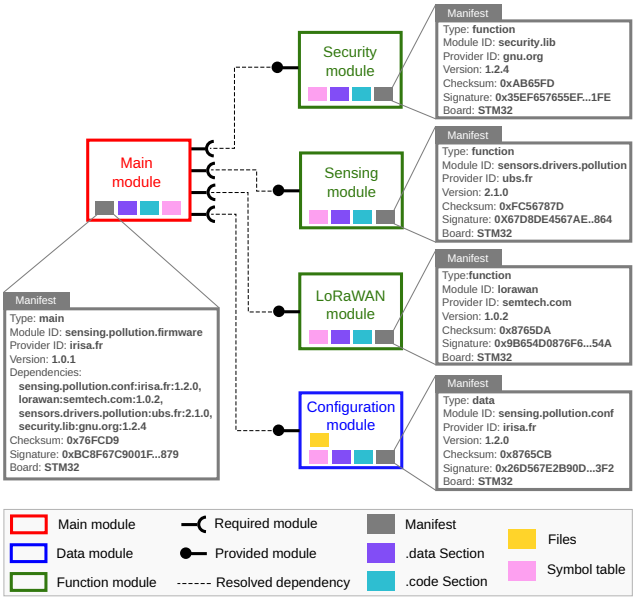

Figure 2. Structure of a modular firmware that can be executed on Droopy.

Section *.code* contains the native machine code of the module, including the functions and the routines that will be executed by the processor. This section is stored on flash memory and is in read access only. It must be noticed that this section can be loaded in RAM by Droopy to speed up the execution of the code. Section *.data* contains global and static variables that are initialized before the execution of the code defined in the *.code* section. The *.data* section is stored in RAM with read and write access. Symbol references that are contained in the machine code of the module must be resolved to the physical memory addresses before the execution. In Droopy, a module is dynamically linked and self-contained, that is, it embeds a symbol table containing all the functions and non-static variables that are referenced to in the module itself.

The rest of this section details the special features of the data, function and main modules.

A. Data modules

Data modules can be simple modules that only define static variables, or more complex ones that include files and functions to process these files. Files contained in a data module are extracted and copied to the file system of a flash partition by Droopy. A ROOT_FS variable is defined in the *.data* section to indicate where the files are stored, thus allowing functions defined in the module to access files. For instance in the example illustrated in Figure 2, encryption keys and TTN account information can simply be defined in the configuration module as static variables.

B. Function modules

Function modules are the most common type of modules. They provide drivers, application and system libraries, and functions and routines created by developers for their own needs. In order to avoid side effects during the update process,

Authorized licensed use limited to: Universidad Nacional de Colombia (UNAL). Downloaded on November 10,2025 at 19:39:29 UTC from IEEE Xplore. Restrictions apply.

function modules that must be hot-replaced (i.e., replaced without requiring a reboot of the system) are expected to be stateless modules. Unlike data modules, function modules are only composed of a manifest, a symbol table and *.data* and *.code* sections, and do not include additional files. If such files are required for the execution of some function modules, they have to be provided by data modules, and a dependency should be defined between the function and a data module.

C. Main module

The firmware deployed on Droopy is defined as a set of modules, one of which is a main module. This main module is the entry point of the firmware. It is automatically loaded and executed by Droopy when the device is started or woken up. It expresses dependencies regarding other modules, and must implement six functions serving as hooks in the module life cycle, namely *on_load()*, *on_start()*, *on_resume()*, *on_suspend()*, *on_stop()*, and *on_unload()*. These functions are called by Droopy when the device is started for the first time or after a deep sleep phase (*on_load()* and *on_start()*), when it is resumed after a light sleep phase (*on_resume()*), when it is put into a light sleep (*on_suspend()*) and when it is put into a deep sleep (*on_stop()* and *on_unload()*). Functions *on_stop()* and *on_load()* are also invoked by Droopy before replacing the main module by a new one. Functions *on_load()* and *on_start()* are called by Droopy after updating the main module. Developers can implement these functions to perform specific tasks. For instance, in the example depicted in Figure 2, a developer can implement in functions *on_start()* and *on_resume()* the code necessary to obtain the pollution level using the functions provided by the sensing module, to transmit this measurement to TTN using the LoRaWAN module, to schedule the next wake-up phase and to put the device in sleep mode for a given time.

D. Module dependencies

A module can be loaded by Droopy only if its dependencies are satisfied recursively, i.e., the dependencies of the required modules are also satisfied, and so on, down to the modules without dependencies. Droopy currently implements a simple dependency resolver that does not yet support complex dependencies like those expressed in packaging systems such as Debian, Apache Maven or Node Package Manager (NPM) (It does not allow for example version number intervals or a minimal version number to be specified.) In Droopy, we currently consider that a new version of a module is compatible with the previous ones (i.e., with a smaller version number), that is, it implements the same functions or defines the same variables. As a consequence, no order is imposed when loading a set of modules in a hierarchy of modules. However, the developer may specify in the manifest that a module breaks the previous ones (*compatibility: no*). If so, the module will be loaded/updated only when all the modules that depend on it have been received by Droopy and can be loaded without breaking backward compatibility.

IV. Management of modules in Droopy

A. Modules' life cycle

Each module has its own life cycle, whose an illustration is given in Figure 3. Depending on their state, the modules can execute specific operations. The following states are considered. Note that states RUNNING and SUSPENDED serve only to the main module, whereas states BOUND and USED serve only to function and data modules.

- UNINSTALLED: The module is in pre-deployment phase. It has not been copied to flash, and thus is not ready to be loaded yet.
- INSTALLED: The module is installed (copied to the flash memory of the device) and now ready to be loaded, but not yet loaded by Droopy.
- INITIALIZED: The module has been loaded and initialized by Droopy (symbol references have been resolved), and is ready to be used by other modules if required.
- RUNNING: The main module is currently running on the platform, and uses the other modules installed and initialized on the platform.
- SUSPENDED: The main module is on standby mode for a light sleep phase. Other modules depending on it are therefore in state INITIALIZED and can be unloaded.
- BOUND: A module (either function or data module) is currently bound to at least one another module.
- USED: A module (either function or data module) is currently used by at least one another module.

The transitions between these states are the following:

- UPLOAD: The firmware of module is copied from the external source/memory to the flash memory of the device.
- REMOVE: The firmware of module is removed from the flash memory because it is no longer used.
- LOAD or UNLOAD: The module is being loaded/unloaded by Droopy.
- START: The main module is started for the first time, woken up after a deep sleep phase or restarted after an update session.
- STOP: The main module is stopped as the device is put on a deep standby phase or must be updated.
- SUSPEND: The main module is put on a light standby phase.
- RESUME: The main module is resumed after a light standby phase.
- BIND: A module (either function or data module) is bound to another one.

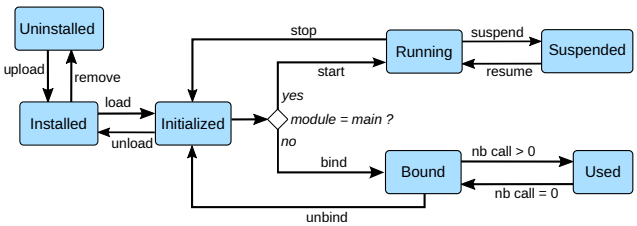

Figure 3. Life cycle of a module.

Authorized licensed use limited to: Universidad Nacional de Colombia (UNAL). Downloaded on November 10,2025 at 19:39:29 UTC from IEEE Xplore. Restrictions apply.

- UNBIND: A module (either function or data module) is unbound from another one.

The START, STOP, SUSPEND and RESUME transitions are dedicated to the main module. Droopy allows indeed the main module to be put on a deep standby phase which is different from a light standby phase. In terms of the update strategy, the management of states and transitions of different types of modules is distinct. Regarding the main module, before it is replaced by an updated one, Droopy stops its operation (if the main module is currently in a light sleep phase, Droopy will wait it to resume the operation first) and sets it to INITIALIZED. After the update session, the main module is restarted to apply the new code. Concerning other types of module (either function or data module), when a new version is ready to replace the active one, Droopy checks if the active module is bound and used by another one. If the module is currently used by some others, Droopy will wait for calls to the module to terminate. If not used, but only bound, Droopy temporarily blocks the module calls and sets the module to INITIALIZED before replacement. After replacement, the module is bound again and its calls are unblocked.

B. Module update

Droopy-based devices support a module-based design, with the capability of loading, unloading, executing, exchanging and replacing independent modules. In terms of firmware updates, this enables a single modified module to be downloaded, replaced and re-executed silently without affecting the operation of other parts of the system. On one hand, allowing only modified modules to be exchanged over a network operator instead of the entire firmware tends to optimize the size of the update, the network overhead and the energy consumption. On the other hand, by adopting a dynamic linking technique, Droopy also enables dynamic modifications to be applied without the necessity of a system reboot. Droopy-based FUOTA protocols ensure a proper update process in four distinct aspects:

1) *Validity.* Droopy determines an incoming update is valid if this update specifies a module with the same identifier, a greater version number comparing to the targeted one that is present on the device, and its dependencies are satisfied.

2) *Integrity.* By computing a checksum, Droopy ensures the module received contains the completely-identical content with the one that is transmitted by the sender.

3) *Authenticity.* Droopy adopts a signature verifier to guarantee the update comes from a trusted authority.

4) *Consistency.* Before a running module is replaced by a modified one, Droopy blocks the calls to all the functions that are implemented in this module to ensure it can not be used during the replacement. Droopy then brings the module back to operational state when the update process finishes.

V. IMPLEMENTATION

We have implemented the Droopy platform for the STM32WLE5JC chip, a high-performance ARM Cortex-M4 generation. This implied developing original code for all the components of Droopy, except the Dynamic Loader and Linker. For this component, we integrated Udynlink[2], a dynamic loader and linker for ARM-based MCUs, that compiles code and produces binary images that can be dynamically loaded, linked and executed at runtime.

The chosen STM32-based device embeds 256 kB of single-bank flash memory, including 128 pages of 2 kB each, as shown in Figure 4. This memory space is reserved for the allocation of the Droopy code and the loadable modules (i.e., the main module, data modules and function modules). Here, our proposition is to allocate the first 50 memory blocks (100 kB) for Droopy, the drivers and libraries needed to program STM32-based devices. The last 78 pages (156 kB) are dedicated to module images. The memory addresses allocated for modules are read from the configuration module that contains configuration properties. Note that the memory allocation strategy depends strictly on the memory organization defined by the chip provider as well as the programmer intention.

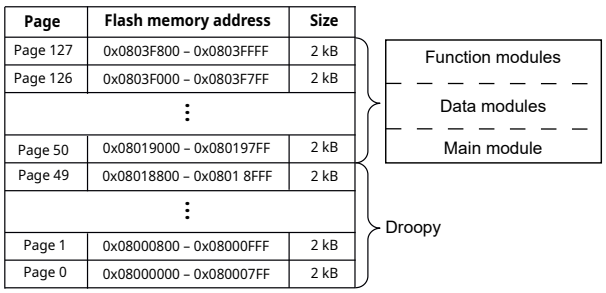

Figure 4. Flash memory of the STM32WLE5JC MCU.

In order to demonstrate the feasibility of Droopy in the context of firmware update for IoT devices, we developed a FUOTA method for Droopy-based devices working in LoRaWAN. This method leverages the modularized firmware of such devices to facilitate partial and incremental updates, that improve the network overhead and the energy consumption, while accelerating the update speed itself. As depicted in Figure 5, the Droopy-based FUOTA architecture is composed of a set of end devices and gateways, a LoRaWAN server and a firmware update server. End devices whose firmware is driven by Droopy are connected directly to the gateways. Each gateway is registered to a LoRaWAN server beforehand. The firmware update server produces the firmware updates, fragments them and propagates the fragments to a set of end devices in a multicast manner, via the LoRaWAN server. The detailed description of the underlying protocol we developed is not given in this paper for reason of space, but Section VI provides a detail comparison between the Droopy-based FUOTA approach and the existing solution in LoRaWAN developed by LoRa Alliance [1].

VI. PERFORMANCE EVALUATION

With the objective to assess the performance of the Droopy platform, in particular in a FUOTA process, we have built a

[2]https://github.com/bogdanm/udynlink/

Authorized licensed use limited to: Universidad Nacional de Colombia (UNAL). Downloaded on November 10,2025 at 19:39:29 UTC from IEEE Xplore. Restrictions apply.

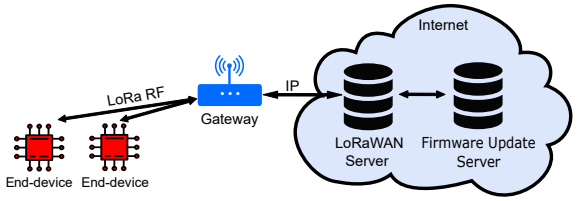

Figure 5. Droopy-based FUOTA architecture

testbed formed of three STM32WLE5JC-based end-devices, a SX1302 868 Mhz gateway and TTN. We considered the deployment of the application mentioned in section III, that monitors pollutant levels in a river and transmits these measurements via LoRaWAN and TTN. When running on Droopy, this application is composed of a main module and four other modules, each fulfilling a specific functionality (sensing, security, configuration, LoRAWAN communication).

This section presents preliminary experimental results by comparing the Droopy-based approach with the traditional monolithic one. Unlike Droopy, dedicated to firmware containing a set of modules, a monolithic design constructs a software system as a single and self-contained unit, that is therefore updated as a whole.

We first focus on the impact of the modularization of Droopy on the execution time, the memory footprint, the boot time and the load time before presenting the benefits of the Droopy-based FUOTA approach.

A. Performance of Droopy-based firmware

Droopy provides a way to accelerate one or more sections of the code that are performance-critical by loading and executing modules in RAM instead of in flash. We consider a cryptography function (in security module) to encrypt the measurement value using the Advanced Encryption Standard 128 bits algorithm. The experiment results report that the time to execute this function in flash is 337 ms in average, which is higher than 278 ms observed in RAM, mainly explained by the fact that resources (e.g., code instructions, data) in RAM can be accessed at a byte level, which is faster than accessing them in pages (2 kB) in flash memory.

The Droopy modular approach, however, comes at a cost in terms of overall firmware size compared with the monolithic design that produces a single self-contained program. Table I shows the modularization overhead of Droopy by comparing the Droopy-based firmware and the monolithic one in terms of memory usage and system boot time. The memory footprint (both flash and RAM) of the Droopy-based program is slightly higher than the monolithic one. This can be explained by the fact that the Droopy micro-system includes the BS, DL, MM, MA, UM and UP core components in addition to the program-specific modules, which also include additional metadata.

The modular approach also results in a slightly slower boot process as shown by the experiment results that report respectively around 1.2 ms and 1.0 ms to completely initialize the Droopy-based program and the monolithic one. This because the Droopy-based boot process contains, not only the configuration of the system clock and of the device drivers as in the monolithic one, but also the initialization of Droopy components. This disadvantage is mitigated by the fact that the Droopy-based system is supposed to reboot less often due to its dynamic update capabilities.

Table I
FIRMWARE MEMORY FOOTPRINT AND BOOT TIME

	Droopy	Monolithic
Flash required	108.3 kB	101.9 kB
RAM required	25.8 kB	23.9 kB
System boot time	1.2 ms	1.0 ms

Table II presents the size and the load time of the considered modules in the Droopy-based program. As can be observed, the larger module, the higher load time. The module sizes including the manifest range from 0.71 kB to 23.76 kB, whereas their load times vary between 0.8 ms and 4.9 ms respectively. The module load overhead is, in our opinion, acceptable due to the fact that modules are not loaded/reloaded frequently.

Table II
MODULE SIZE AND LOAD TIME IN DROOPY

	Size	Load time
Main module	1.0 kB	1.1 ms
Security module	23.76 kB	4.9 ms
Configuration module	1.48 kB	2.19 ms
Sensing module	0.71 kB	0.8 ms
LoRaWAN module	1.95 kB	2.4 ms

B. Performance of Droopy-based FUOTA approach

We have compared the Droopy-based FUOTA approach with the standard FUOTA solution in LoRaWAN that is commonly used for devices based on a monolithic firmware architecture. For comparison purposes, the experiments involved a modification of the encryption key that is to be propagated to end devices. This modification imposed the update of the entire (monolithic) firmware whereas, with Droopy, it only required the update of the configuration module. The system configuration and the LoRaWAN parameters are presented in Table III.

Evaluation Metrics. The following metrics are used for the comparison between the two approaches:

- *Update Size* is the total size of the code that needs to be updated.
- *Update Time* is the total time needed to update the firmware, between the moment the update action is invoked and the moment when the new firmware is successfully delivered and applied to every device.
- *LoRa Overhead* comes from the LoRa preamble, the LoRa header and the cyclic redundancy check that are automatically inserted into the physical payload to be broadcasted in LoRa networks.

Authorized licensed use limited to: Universidad Nacional de Colombia (UNAL). Downloaded on November 10,2025 at 19:39:29 UTC from IEEE Xplore. Restrictions apply.

Table III
EXPERIMENT PARAMETERS

Number of devices/gateway	3/1
Frequency band	868 MHz
LoRaWAN device class	Class C
Spreading factor	7
Coding rate	4/5
Bandwidth	125 kHz
Duty cycle	1%

Results. Table IV compares the proposed Droopy-based FUOTA approach with the standard one in terms of the update size, the update time and the LoRa overhead. In order for the firmware to apply the new encryption key, the Droopy-based solution requires to exchange 1.48 kB compared to 101.9 kB observed in the standard one: the benefit from being able to transmit only a module instead of the entire firmware is clear. The difference in the update size is the main reason for the difference in the update time and LoRa overhead. Indeed, a successful update session for Droopy-based devices demands in average a period of approximately 5.6 minutes, which is 74 times faster than for monolithic-based devices. The gain on the LoRa load is also remarkable. Due to the low bandwidth and payload size of the LoRa technology, reducing the number of packets, and thus the LoRa overhead is particularly advantageous. As shown in the table, the average LoRa overhead decreases from 184.5 kB for the standard solution to 2.46 kB when updating a single module using Droopy. Due to their smaller size, updating other modules (main module, sensing module, security module or LoRAWAN module) is also expected to achieve a significantly higher performance compared to the existing solution in LoRaWAN in terms of the update time and the LoRa overhead.

Table IV
PERFORMANCE OF THE DROOPY-BASED FUOTA APPROACH AND THE STANDARD ONE

	Droopy-based approach	Standard solution
Update size	1.48 kB	101.9 kB
Average update time	5.6 minutes	415.0 minutes
Average LoRa overhead	2.46 kB	184.5 kB

VII. RELATED WORK

Some works dealing with firmware update over-the-air, such as [2], [3] and [4], have shown that partial and incremental updates of firmware allow to preserve the power budget of end devices and significantly reduce the network traffic and the transmission time. In this section, we present systems that support partial and incremental updates of firmware without necessarily requiring a system reboot, notably by implementing dynamic linking techniques, strict or loosely coupled binding models, or containerization techniques, and we compare them with Droopy.

Nguyen et al. [4] have proposed a module-based approach for firmware update process that might reduce the size of the updates, the network overhead and the energy consumption of end devices, while accelerating the update speed itself. However, unlike [4], Droopy considers all types of module (data, function and main modules), supports dependency and module's life cycle management, and provides additional features for ensuring a proper update process in terms of validity, integrity, authenticity and consistency.

In [5], Dunkels et al. have introduced a runtime dynamic linker and loader to more easily reprogram wireless sensor devices running on Contiki [6], a lightweight operating system for tiny networked devices. This dynamic linker and loader relies on the Contiki static symbol table, which is generated beforehand and that cannot be modified afterward. It is therefore limited and of little use for dynamic updating of firmware modules whose functions may evolve over time. In DyTOS [7], Munawar et al. have replaced the static symbol table by a dynamic and extendable one in order to overcome limitations such as those encountered with [5]. However, unlike Droopy in which the symbol table is assembled to the module itself making it self-contained and reusable, both Contiki and Dynamic TinyOS adopt a global symbol table for every module.

GITAR [8] goes one step forward by proposing an embedded platform that supports the dynamically linking of components at runtime; these components may be those forming the application and those forming the network protocol stacks. To this end, it distinguishes three levels of architecture for the firmware, namely the system level, the kernel level and the component level. The system level implements the drivers and the core OS functionality. It is compiled statically and can only be updated by performing a firmware update. The kernel level provides an interface between the system level and the component level. Its role is to dynamically bind the components to each other and to the system level functionality at runtime. In GITAR, network protocol stacks are grouped in dynamically update-able components. Components contain empty references to other components that are resolved at runtime and control functions used to enable indirect interactions between different components. Once these references have been updated by the kernel, a component is able to redirect a call to any other component (or system level functionality) without needing the kernel. Like in Droopy, in GITAR the code is compiled as an independent-position code. In GITAR, components are identified by their name and their version number. They do not exhibit dependencies regarding other components for their execution, thus limiting the coherent update of a firmware composed of several modules (some components may become incompatible with others as updates are released). Moreover unlike Droopy, GITAR does not address the over-the-air update of the components using long range and low-power wireless communication technologies.

In SOS [9], each dynamic module encapsulates its external

Authorized licensed use limited to: Universidad Nacional de Colombia (UNAL). Downloaded on November 10,2025 at 19:39:29 UTC from IEEE Xplore. Restrictions apply.

functions in function control blocks (FCBs). In order to execute a function of a given module, other modules require a pointer to the FCBs and rely on the SOS kernel to execute the function pointer call, thus introducing a CPU overhead. To limit the CPU overhead introduced by FCBs, Enix [10] and RemoWare [11] use a jump table that provides a direct mapping between the name and the pointer of functions. The function pointer call is also delegated by the components to the kernel, but with less overhead. To reduce the memory overhead introduced by the jump table, both Enix and RemoWare combine a strict binding model and loosely coupled binding model for respectively the system level and the dynamic component level. Like in Droopy, in RemoWare, components can express dependencies on other components, which is not the case for Enix and SOS. Neither Enix, nor SOS, nor RemoWare provide any means of dynamically updating components using long range and low-power wireless technologies. DyTOS [7], a TinyOS based system, supports incremental remote firmware updates, by relying on the dynamic exchange of software components forming the operating system and its applications. However, unlike Droopy and RemoWare, DyTOS does not provide a real component model with dependencies, but only a dependency management based on the symbol tables.

Zandberg et al. [12] have introduced a novel container-based middleware, named Femto-Containers, that supports the secure deployment, execution and isolation of small pieces of software on low-power IoT devices. The Femto-Containers implement an event-driven programming model. Pieces of software are only executed when they are triggered by events in the operating system. For that, these pieces of software specify their entry points and the operating system's hooks they are to be attached. Pieces of code that can be deployed in in Femto-Containers are limited to the eBPF architecture. Network transmissions and power consumption can be reduced if software updates are applied to a piece of code running in a Femto-Container instead of to the full firmware. In [12], software update metadata are defined by SUIT [13], which allows authentication, integrity checks, rollback options and specification of dependencies. Nevertheless in [12], authors do not describe how dependencies between eBPF codes forming an application and running in different containers are managed and updated in a coherent manner.

VIII. Conclusion

In this paper, we have proposed a dynamic runtime platform for MCU-based embedded devices called Droopy. This platform supports the execution of firmware implemented as a composition of software modules. It provides several advanced features to dynamically load and unload modules, to manage module dependencies, to speed up the execution of the modules and to update them over-the-air without necessarily requiring a reboot of the device. The performance evaluation of Droopy shows that its memory footprint and additional computing overhead are negligible, and that it can significantly speed up the execution of modules by putting their code in RAM instead of in flash.

Although other works have proposed modular approaches for embedded micro-systems, the module dependency management is often limited or they do not address remote firmware update over-the-air (FUOTA), which is a major challenge in LPWAN, and more generally in IoT. Droopy proposes a FUOTA mechanism over LoRaWAN that drastically reduces update times, preserving both the network load and the energy of devices, and easing the burden on developers during this time-consuming task.

Droopy is currently under active development, and we plan to introduce new features and enhancements in the future, such as a better memory isolation between modules, a no-reboot memory defragmentation mechanism and a better dependency management by adopting something close to the dependency management systems of Debian packages, Apache Maven modules or Node.js modules. It is also planned to implement Droopy on real-time operating systems like FreeRTOS or RIOT.

References

[1] N. Sornin, "LoRaWAN®: Firmware Updates Over-the-Air," Semtech, Tech. Rep., Apr. 2020.

[2] Z. Sun, T. Ni, H. Yang, K. Liu, Y. Zhang, T. Gu, and W. Xu, "FLoRa: Energy-Efficient, Reliable, and Beamforming-Assisted Over-The-Air Firmware Update in LoRa Networks," in *22nd International Conference on Information Processing in Sensor Networks (IPSN 2023)*. San Antonio, TX, USA: ACM, May 2023, pp. 14–26.

[3] B. P. Neves, A. Valente, and V. D. N. Santos, "Efficient Runtime Firmware Update Mechanism for LoRaWAN Class A Devices," *Eng, MDPI*, vol. 5, no. 4, pp. 2610–2632, 2024.

[4] H. D. Nguyen, N. Le Sommer, and Y. Mahéo, "Over-the-Air Firmware Update in LoRaWAN Networks: A New Module-based Approach," *Procedia Computer Science*, vol. 241, pp. 154–161, 2024.

[5] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-Time Dynamic Linking for Reprogramming Wireless Sensor Networks," in *4th International Conference on Embedded Networked Sensor Systems (SenSys 2006)*. Boulder, CO, USA: ACM, Oct. 2006, pp. 15–28.

[6] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors," in *29th Annual International Conference on Local Computer Networks (LCN 2004)*. Tampa, FL, USA: IEEE, Nov. 2004, pp. 455–462.

[7] W. Munawar, M. H. Alizai, O. Landsiedel, and K. Wehrle, "Modular remote reprogramming of sensor nodes," *International Journal of Sensor Networks, Inderscience*, vol. 19, no. 3/4, p. 251–265, Nov. 2015.

[8] P. Ruckebusch, E. De 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, Elsevier*, vol. 36, pp. 127–151, Jan. 2016.

[9] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A Dynamic Operating System for Sensor Nodes," in *3rd International Conference on Mobile Systems, Applications, and Services (MobiSys 2005)*. Seatle, WA, USA: ACM, Jun. 2005, pp. 163–176.

[10] Y.-T. Chen, T.-C. Chien, and P. H. Chou, "Enix: A Lightweight Dynamic Operating System for Tightly Constrained Wireless Sensor Platforms," in *8th ACM Conference on Embedded Networked Sensor Systems (SenSys 2010)*. Zürich, Switzerland: ACM, Nov. 2010, pp. 183–196.

[11] A. Taherkordi, F. Loiret, R. Rouvoy, and F. Eliassen, "Optimizing sensor network reprogramming via in situ reconfigurable components," *ACM Transaction Sensor Networks*, vol. 9, no. 2, pp. 1–33, Apr. 2013.

[12] K. Zandberg, E. Baccelli, S. Yuan, F. Besson, and J.-P. Talpin, "Femto-containers: lightweight virtualization and fault isolation for small software functions on low-power IoT microcontrollers," in *23rd ACM/IFIP International Middleware Conference (Middleware 2022)*. Quebec, QC, Canada: ACM, Nov. 2022, pp. 161–173.

[13] B. Moran, M. Meriac, H. Tschofenig, and D. Brown, "A Firmware Update Architecture for Internet of Things Devices," RFC 9019, 2021.

Authorized licensed use limited to: Universidad Nacional de Colombia (UNAL). Downloaded on November 10,2025 at 19:39:29 UTC from IEEE Xplore. Restrictions apply.