Implement Updates Over the Air for ESP8266 microcontrollers

Dustin Frisch
C/C++ for Embedded Systems and Physical Computing
Angewandte Informatik
University of Applied Sciences Fulda
dustin.frisch@informatik.hs-fulda.de

February 8, 2017

This article describes the implementation of durable and stable system for building firmware updates for embedded systems based on *ESP8266*[1] microcontrollers. This includes the mechanisms used to build the updates, distribute and install them on the device.

1 Introduction

In embedded systems, the software, also known as firmware, is an essential part of the system. On one side, it interacts with the hardware in a system specific way by implementing the specifications required by the components used in the system. On the other side, it provides use-case dependent functionality in interaction with general purpose hardware components.

Embedded systems are often thought as systems that never change their requirements or functionality. However, practical use shows that the environment in which these systems run do, in fact, change. These changes include, but are not limited to, modifications of to the expected behavior or additions to it, reconfiguration of parameters related to communication with other systems and the users as well as correcting errors that have been reported after deployment. In almost all cases, the requirements can be accomplished by changing the firmware and do not need any modification of the hardware. For updating the firmware on a system being deployed, the system must provide an interface for altering the firmware. In addition, such an interface should provide mechanisms to check which firmware is currently installed and which configuration parameters are used.

Even if systems are equipped with an interface for applying updates, the maintenance cost can still be enormous because administrators have to interact with each device physically. For systems that are located in areas where reachability is limited the cost is increased even more. If a system is already able to communicate over a network interface, this can be leveraged to apply updates on these system - this is typically referred to as *Over the Air (OTA)*. By reusing the existing communication channels, the dedicated update interface can be omitted which leads to smaller packaging and reduces production cost. It also decreases the maintenance cost drastically because updates can be installed remotely.

OTA updates enables administrators to apply automation methods on the update process allowing to roll out updates in a controlled fashion. I.e. updates can be done on testing devices first, followed by security-critical deployments and subordinate ones can be delayed to times where the device is not utilized. Information about the update status provided by the devices allows administrators to apply monitoring techniques ensuring all updates are installed and devices ar in the desired state.

[1] ESPRESSIF. *ESP8266EX Overview.* 2017. URL: `http://web.archive.org/web/20170130001257/http://www.espressif.com/en/products/hardware/esp8266ex/overview` (visited on 01/30/2017).

2 Environment

The home-automation projects developed by *Magrathea Laboratories e. V.*,[2] the local hackerspace in Fulda, are used to provide control over the different actors and sensors in the foundations rooms to visitors and members locally and remotely.

The different components available (like the door status, power sockets, projectors and screens, temperature sensors, etc.) are all managed by the home-automation controller driven by the software *home-assistant*.[3] It provides direct control over all existing components using a web UI and allows to define rules and automations on how these components interact.

The hackerspace has developed a common software and hardware platform for its home-automation projects called *ESPer*.[4] For the hardware, boards based on the *ESP8266* micro-controllers, mostly *ESP-01s*[5] boards, are used in combination with self-developed power supplies and use-case specific hardware components.

These boards provide a Microcontroller Unit (MCU) fast enough for all required scenarios and integrate WiFi without requiring any extra components. The software is based on the *Sming*[6] library, which in turn is based on the open source SDK for *ESP8266* and integrates a lot of other software components for easy use. To build the software, a *Makefile*[7] is used, which provides a simple way for reproducible builds.

For communication with the controller, the *MQTT*[8] protocol is used. It provides a lightweight messaging mechanism implementing the publish-subscribe pattern that allows devices to listen for commands and publish their current state to the controller and other interested parties. The controller software has out-of-the-box support for this protocol, which allows easy integration of all different device types using the same patterns.

The components all share the same configuration in regard to the network access and the controller to communicate with. The configuration is provided during build time, which eschews the need for a configuration interface and reduces the management overhead, thus minimizing security leaks.

[2]Magrathea Laboratories e.V. *Magrathea Laboratories - Creating new Worlds*. 2016. URL: `http://web.archive.org/web/20161116123421/https://maglab.space/` (visited on 11/16/2016).

[3]Home Assistant. *Awaken your home*. 2017. URL: `http://web.archive.org/web/20170102023619/http://home-assistant.io/` (visited on 01/02/2017).

[4]ESPer. *ESPer - Space Automation Firmware for ESP8266*. 2017. URL: `https://git.maglab.space/esper/esper` (visited on 02/02/2017).

[5]SparkFun. *WiFi Module - ESP8266*. 2017. URL: `http://web.archive.org/web/20170104002307/https://www.sparkfun.com/products/13678` (visited on 10/28/2017).

[6]Sming. *Sming - Open Source framework for high efficiency native ESP8266 development*. 2016. URL: `http://web.archive.org/web/20170206144443/http://sminghub.github.io/Sming/about/` (visited on 11/25/2016).

[7]The IEEE and The Open Group. *The Open Group Base Specifications Issue 6 - make - maintain, update, and regenerate groups of programs*. 2004. URL: `http://pubs.opengroup.org/onlinepubs/009695399/utilities/make.html` (visited on 11/27/2016).

[8]OASIS Standard Incorporating. *MQTT Version 3.1.1 Plus Errata 01*. 2015. URL: `http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/errata01/os/mqtt-v3.1.1-errata01-os-complete.html` (visited on 01/15/2017).

3 Requirements

The following requirements are defined as global project goals and have been refined during the work on the project multiple times.

- The systems should be able to perform updates on the release of new software without administrative interaction. If a new version of the firmware is published, it should be prepared automatically for installation to the target devices. All these devices should then download and install the new software version and start using it subsequently, if no errors have occurred during the process.
- To ensure minimal maintenance effort, the update process should be insusceptible to errors as most as possible. Even if the installation of an update fails in the middle of reprogramming the controller, the system should continue to work fully functional immediately and after a reboot.
- Downloading the updated firmware should be done over the WiFi interface using the same network connection as used during normal operation. Fetching the firmware should be done side-by-side with other traffic produced during operation.
- Reducing network load and aiming for the maximum possible device uptime is critical. Therefore, the update process should only be done if a new version is available. In contrast, the release of a new update should be rolled out to all devices as fast as possible. While checking for available updates and downloading such an update, the device should continue to work as usual.
- For easy maintenance and monitoring, each device should provide detailed information about the currently installed firmware version and other details relevant for the update process.
- Devices are categorized by types. Each type runs the same software and therefore provides the same functionality. As the device type is hardly coupled to the hardware and the software interacts with it on a specific way, the update process must ensure that the correct firmware is used while reprogramming. The according device type is provided as a string through a global constant at compile time and it must never be changed during operation.

4 Implementation

Implementing *OTA* updates under the given requirements involves three different components, which interact closely.

The first component implements the update mechanism in the firmware running on the embedded device. It is responsible for checking and downloading the updates and installing them. Second, the build system is in charge of building the firmware from

source and publishing the built binary images. Finally, the deployment provides infrastructure for downloading the binary firmware images and triggering the update on all devices.

4.1 The update mechanism

The implementation of the update mechanism consists of three parts which interact closely: checking for updates, reprogramming the device and reconfiguring the boot process. This sections describes all three of these parts in detail.

The build-time configuration was extended to include a new option called `UPDATER_URL`, which is the base URL used to query the update server. Each device requires to have this option set to make the update work. If the option is skipped, the code for update management is excluded during the build.

4.1.1 Checking for updates

Initially, each device queries the update server regularly for the current firmware version and initializes the update process if remote and local versions differ. To do so, the update server provides a file for each device type containing the available version identifier, which is stored beside the firmware binary files. These version identifier files are provided by the update server using *HTTP 1.1*[9] under the following path pattern: `$DEVICE.version` (whereas `$DEVICE` is the device type name). The version identifier can be an arbitrary string as the content is not interpreted semantically but only compared to the version identifier used during build time.

Each device tries to fetch the version identifier file once every hour. After the version identifier file has been downloaded successfully, the whole file content is compared to the version identifier provided during build time. If the version identifiers differ, the update process is initialized; in cases where the download has failed, the update server or the network connection was not available or any other error occurred, another attempt will be made at the next interval.

In addition to the interval, a special *MQTT* topic shared by all devices is subscribed on device startup: `$MQTT_REALM/update`. Every time a message is received on this topic, a fetch attempt for the version identifier file is triggered. This allows faster roll-outs of updates and finer control for manual maintenance.

4.1.2 Reprogramming the device

As the binary to download and flash possibly exceeds the size of free memory heap space, the received data must be written to the flash directly. In contrasts, executing the code from the memory mapped flash while writing the same area with the downloaded update leads to errors, as the executed code changes immediately to the updated one. To avoid this, the flash is split into half to contain two firmware ROMs with different versions, one

[9] The Internet Society. *Hypertext Transfer Protocol – HTTP/1.1.* 1999. URL: `https://www.w3.org/Protocols/rfc2616/rfc2616.html` (visited on 01/15/2017).

being executed and one which is being downloaded. This standby firmware also acts as a safety mechanism if the download fails or is interrupted as the previous version stays intact and can still be used. In case of an error the old firmware is kept unchanged and will be used until the successful download of a newer firmware succeeds.

Listing 1: Linker script to build firmware for two ROM slots.

```
irom0_0_seg : org = ( 0x40200000         // The memory mapping address
                    + 0x2000             // Bootloader code and config
                    + 0x10               // Data offset after header
                    + 1M / 2 * ${SLOT}   // Offset for the ROM slot
                    ),
              len = ( 1M / 2 - 0x2010 )  // Half ROM size excl. bootloader
```

Microcontroller boards based on the *ESP8266* MCU are mostly following the same layout: the MCU is attached to a flash chip which contains the bootloader, firmware and other application data. The memory mapping mechanism of the MCU allows only a single page of 1 MB of flash to be mapped at the same time[10] and the selected range must be aligned to 1 MB blocks. As the *ESP-01s* is only equipped with 1 MB of flash, this means that the whole memory is mapped to a contiguous address space. Therefore, the second ROM can not be re-mapped to have the same start address as the first ROM. While the firmware is executed without any dynamic linking mechanism and the chip does not support position independent code, the addresses used in the ROMs are dependent to the offset at which the firmware is stored. This arises the need for building two firmware binaries, one for each target location. To do so, a linker script for each of the two ROM slots was created, which is used to create two variations of the same firmware, only differing in ROM placement. The two resulting firmware binary files are both provided for download via HTTP 1.1 - which one to download depends on the target ROM slot and is selected by the device during the update process. Listing 1 shows the only difference between the two linker scripts, where `$SLOT` must be replaced with the slot number according to the current build.

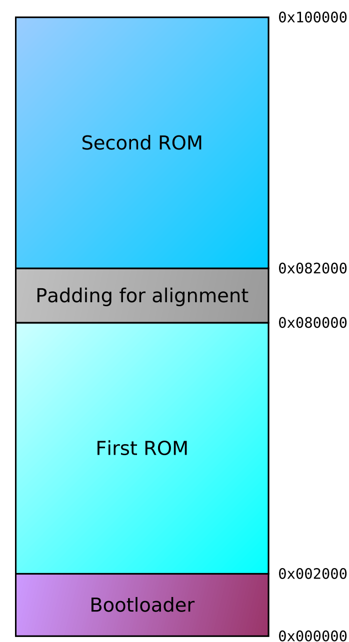

Figure 1: The flash layout used for two ROMs.

In addition to the two ROMs, the flash must provide room for the bootloader and its configuration. *rBoot*[11] has been choosen as it is integrated within the *Sming* framework

[10] ESP8266 community wiki. *ESP8266 Memory Map*. 2016. URL: `http://web.archive.org/web/20161118224802/http://www.esp8266.com/wiki/doku.php?id=esp8266_memory_map` (visited on 01/30/2017).

[11] Richard Antony Burton. *An open source bootloader for the ESP8266*. 2016. URL: `http://web.archive.org/web/20160611044740/https://github.com/raburton/rboot` (visited on 01/30/2017).

and allows to boot to multiple ROMs. For configuration, an *rBoot* specific structure is placed in the flash at a well-known location directly after the space reserved for the bootloader code. This structure contains, among other things, the target offsets for all known ROMs and the number of the ROM to boot from on next reboot.

The full memory layout of this approach is shown in figure 1. To calculate the origin of application data for each slot, the available memory of 1 MB is split in half and an offset of the size of the bootloader code and its configuration (0x2000 bytes) is added. For alignment and easy debugging, the second block is also shifted by the same amount ob bytes as the first block. The unused gap of 8192 bytes is used by some applications to store data which can persist over application updates.

Listing 2: The flash layout used for two ROMs.

```
#define UPDATER_URL_ROM(slot) (( UPDATER_URL "/" DEVICE ".rom" slot ))

// Select rom slot to flash
const auto& bootconf = rboot_get_config();

// Add items to flash
if (bootconf.current_rom == 0) {
  updater.addItem(bootconf.roms[1], UPDATER_URL_ROM("1"));
  updater.switchToRom(1);
} else {
  updater.addItem(bootconf.roms[0], UPDATER_URL_ROM("0"));
  updater.switchToRom(0);
}
```

For installing a firmware update, the new firmware binary file is downloaded using an HTTP GET request. The update server provides these files in the exact same way as it provides the version identifier files, but the path pattern differs: the suffixes `.rom0` and `.rom1` are used to provide the firmware binary files for the first and second slot respectively. The firmware files provided on the update server are the exact same ones as used to initially flash the chip for the according version. Using the same files for flashing and updating allows better debugging by eliminating errors related to the update process itself and makes development and initial installation very easy. Listing 2 shows the algorithm used to determine the download address and reconfigure the bootloader.

After the download of a new ROM has been finished successfully, the bootloader configuration is altered to boot to the new ROM slot and the device is rebooted.

4.1.3 Publish device information

For monitoring and maintenance purposes, each device publishes a set of information to a well-known *MQTT* topic after connecting to the network. Beside already existing dates like device type, chip and flash ID, the information block has been extended with details about the bootloader, SDK and firmware version as well as relevant details from the bootloader configuration, like the currently booted ROM slot and the default ROM

slot to boot from. This allows administrators to find devices with outdated bootloaders and helps to find missing and failed updates.

4.2 Multi-Device build infrastructure

The firmware for all *ESP8266* based devices in the hackerspace are all based on the same framework. *Sming* provides the base library for this framework. In addition, components and functionality shared by all devices has been identified and are providing a framework for the existing and possible further devices. This framework provides a functional base for all devices and allows to reuse code providing functionality which is common in multiple devices. The framework also includes a build system, which allows to configure some basic parameters for all devices. Including, but not limited to, the Wi-Fi access parameters, the *MQTT* connection settings and the updater URLs. By sharing the same code, all devices ensure to have a common behavior when it comes to reporting the device status or interacting with the home-automation controller. This eases configuration and allows to collect information about all devices at a central location.

Each device firmware exists as a separate project and includes a link to the framework. As development on these devices happens in cycles, older projects are missing updates of the framework and therefor do not benefit from newly added features or fixed problems. Updating the framework version and rebuilding the firmware would often result in an easy gain of these benefits, but requires manual interaction. More problems will arise if the application programming interface (API) of the framework changes. In this situation, the device firmware must be updated to use the changed API, which can be an unpleasant and complex task and leads to higher latency for firmware updates.

To prevent these problems the device firmware of all devices in the hackerspace is now integrated with the framework into a larger project. By doing so, each device specific code is always linked to the latest version of the framework. Device specific code is now organized as a folder for each device type. The build system has been modified to scan for all device specific folders and call the original build process for each of them.

4.2.1 Framework integration

The framework has been changed to keep control over the application life-cycle. It ensures that the device unspecific code is executed at the right time and provides an API for device specific functionality.

The framework specifies a simple interface, which must be implemented by each device. A single function `Device* getDevice()` must be defined exactly once in each device specific folder. To implement this interface, a static instance of `Device` is created and returned. Each `Device` is populated with device specific `Feature` instances. While the `Feature`-API leverages common runtime polymorphism to share functionality between features, the initial `Device` creation uses compile-time polymorphism which reduces the need for memory management and increases performance by avoiding virtual function tables. Listing 3 shows the complete device specific code used for a simple power socket.

Listing 3: Device specific code for a socket driver.

```
#include "Device.h"
#include "features/Socket.h"

Device device:

constexpr const char SOCKET_NAME[] = "socket";
constexpr const uint16_t SOCKET_GPIO = 12;
OnOffFeature<SOCKET_NAME, SOCKET_GPIO, false, 1> socket(&device);

Device* getDevice() {
  return &device;
}
```

4.2.2 Build system

The existing *Makefile* has been refactored to accept a parameter for device type identifiers called `DEVICE` and to create its whole output inside a subdirectory specific to the device type. Another *Makefile* has been created which scans a project subdirectory and uses each directory in there as container for device specific code. For each of these directories, the other *Makefile* is called and the subdirectories name is used as `DEVICE` parameter. By splitting the build and recompiling the framework each time before intermixing it with the device specific code, the device type identifier can be used inside the shared framework code.

While building a devices firmware, the version identifier file used during updates is also created and stored beside the binary firmware image.

For development, each device can be build separately by using the device type identifier as *Makefile* target. In addition the prefix `/flash` can be used to flash a specific firmware.

4.3 Automatic deployment and roll-out

The source code of the *ESPer* project is published into a *GIT* source code repository which is provided by the hackerspace. To avoid interferences between different build environments on developers computers and roll out new versions as early as possible, the code has been integrated into a continuous integration (CI) system. The CI, which is based on *drone*[12] and provided as part of the hackerspace infrastructure, allows to execute commands on each version published into the *GIT* repository. Therefore a *drone* configuration file as shown in Listing 4 has been added to the source code as `.drone.yml`.

As shown in the configuration Snippet, the build environment includes some special settings. First, the `CONFIG=maglab` option lets the build system use `Configurion.mk.maglab` instead of the default configuration file. This configuration

[12] Drone. *Drone is a Continuous Delivery platform built on Docker, written in Go*. 2016. URL: `http://web.archive.org/web/20160705005808/https://github.com/drone/drone` (visited on 02/05/2017).

file is stored inside the repository, too. To keep the WiFi password secret, it is not written down in the configuration, but must be specified in the environment. To include secrets into a build process while allowing to keep the configuration public, *drone* allows to encrypt these with a repository specific key. Using this method, the password is stored as `.drone.sec` file inside the repository from where it is injected into the build environment. At last, the firmware version is configured to be made out of the first 8 letters of the *GIT* commit hash, which uniquely identifies a version of the source code.

Listing 4: The *drone* configuration for the *ESPer* project.

```
build:
  image: maglab/sming
  environment:
    - CONFIG=maglab
    - WIFI_PWD=$$WIFI_PWD
    - VERSION=$${COMMIT:0:8}
  commands:
    - make clean
    - make
publish:
  sftp:
    host: eddie.maglab.space
    username: esper
    files:
      - dist/*
    destination_path: './'
    when:
      branch: master
```

For deployment, only the master branch is considered. After a successful build, all distribution files (the binary firmware files and the version files) of all devices are copied to the machine running the home automation controller software into a directory served by a HTTP server. The used configuration file references this server as source of updates.

5 Conclusion

The project has been successfully deployed in the hackerspace and is now an essential part of home-automation development and deployment.

The update infrastructure has been the crucial point for decisions towards the framework for most members. Enabling the developers to do updates in combination with the shared configuration and behavior provided by the framework resulted in a massive speedup when it comes to project deployment. Before that, the cost for an change after deployment was estimated so high, that most projects tend to delay deployment until all required and wanted features are implemented. Now, as the devices are deployed as soon as the hardware is considered stable, these devices start to provide functionality early and therefore the developers can get better feedback on the provided functionality.

Most of the devices running the update-enabled firmware have undergone multiple major updates without any problems. This includes a major network configuration change and a big stability fix for network communication. All devices applied the update successfully and started to work without any manual interaction required afterwards.

The project will be continued to extend the functionality with features already being in development. The latest development includes enhanced checksum verification where the firmware can be signed using cryptographic methods and will be verified during the update process. In addition, the information provided by the device about the firmware status will be enhanced to allow better control and reduce maintenance effort even more.