Remote firmware update management of embedded devices

Oliver Hollý

Department of science and innovations, FEI STU in Bratislava
Institute of automotive mechatronics

xhollyo@stuba.sk

Abstract – Publication describes concept and practical implementation of remote firmware updates management for embedded and internet of things devices or systems. Firmware update feature is fundamental operation in development and operation process to reach required system nature. This paper has been created as a summary of knowledge acquired during participating on a real commercial product in my part time job

1 Introduction

1.1 Motivation

The general motivation for developing remote firmware updates management infrastructure is to achieve the possibility to manage, upgrade or check firmware version of multiple devices connected to one centralized remote server. Reliable development and operation of embedded systems is based on testing and debugging firmware and its new versions and utilities**.**

We can split a software development process into two main stages. First stage begins with planning, analysis, design and finishes after development and implementation. Second stage of the software development process consist of testing and maintenance or service. First state is the best period for time and cost efficiency of software development process. In this first state we can afford to do lot of changes and adjustments based on client's needs.

It is a good practice to use debugging tools and boards with lots of interfaces and GUIs. But these tools are not available in state of maintenance and service, because of PCB size constrains, power consumption, computing capacity etc. And this is where our solution takes meaning. We need way to upgrade features of final product, deployed in operation miles away or just without needed additive wires and hardware.

1.2 State of the art

Software distribution techniques are today common and standardized in the whole IT sector. Embedded systems software distribution feature, also called bootloader, must physically handle process of writing program to flash memory. Difference between boot-loading process of commons embedded development platforms and our proposal is distribution complexity, depends on communication interface. The most common concept of firmware distribution use short distance serial communication, but our concept is based on communication over the internet. Related work called Firmware Over The Air [1], is technique widely used on embedded devices with high computing power and running operating system. So, there are differences and challenging opportunity to create remote firmware management system for the pure embedded system without operating system.

1.3 Terminology

Firmware is a low-level control software for embedded systems Firmware exactly determines the function of the microcontroller, while it is reading sequence of tasks and commands in machine code from the program memory. To achieve real time conditions and security, firmware depends on the efficiency and robustness. In the following sections we will use term firmware for the application software.

Bootloader is a small program which writes firmware in binary format to the embedded device flash program memory during the boot phase. Most microcontrollers (MCU's) had the bootloader in their memory by default. These bootloaders are running after setting up MCU or after reset, and they are waiting for user request event to write new firmware, usually listening on serial interface for short time. Our custom written bootloader must be more sophisticated, because we want to communicate over interface supported by the server. Available interfaces in our project are industrial Ethernet and internet UDP socket connection. Another communication possibilities are e.g. Wi-Fi, Bluetooth or USART. Bootloader design and an its implementations is described in the rest of this paper.

1.4 Initial technological requirements

The fundamental aspect to start developing remote firmware managed system is at least one server-client communication interface. Enabled write and read from client's program flash memory with enough free space for bootloader and firmware program binary file.

2 Technical design and implementation

2.1 Client implementation

In our case the client is an embedded device based on ARM STM32F4 microcontroller, with Power over Ethernet (PoE) supply [2]. UDP communication is available with dynamic IP addressing too. Client is capable to find server's IP address by monitoring responds on broadcast sent messages. Structure of the client's FLASH memory is shown in Figure 1.

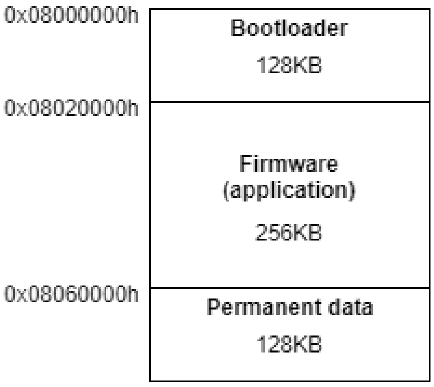


Figure 1 STM32F4 FLASH program memory partitions

Flash memory in STM32F4 has 512KB space for user program and begins at the address 0x08000000h. Firmware or application program partition has the most memory reserved and begin at address 0x0820000h.

Permanent data partition contains custom data for firmware or bootloader needs but could be also helpful for data exchanging between bootloader and firmware on a shared memory principle. The problem is that during the bootloader program is running, the firmware is out of stack pointer scope and vice versa. That means bootloader has no access to firmware program data and instructions. To solve the issue, we use a permanent data partition to store firmware version index number.

As long as the client-server communication is opened, we would like to know some information about the client. The examples of such information are the actual firmware version index, IP address, processor unique number that is an analogy to MAC address of network cards. Firmware index should be a real date and time of compilation or build. Standard C99 compilers offers _DATE_ and _TIME_ macros which do the thing at compile time. Processor unique number we got from STM32 memory where it is stored as a real serial number of chip. Table 1 shows example of 3 clients connected to server at the same time.

Main functionality of bootloader program control flow is based on general standards methods. Basic implementation is waiting for user request for specified short time after reset. If the request arrives in time, then bootloader can start to receive firmware packets and write them into the flash memory. After the timeout, running bootloader jumps to the application program partition, by modifying main stack pointer register content, in our case to value 0x0820000h.

2.2 Server implementation

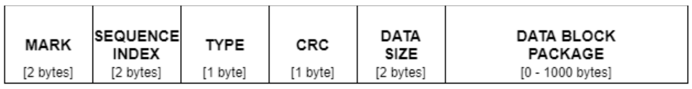
The purpose of the server is to serve as a centralized communication point, which unite and manage multiple clients. Backend of our server prototype has been implemented like desktop C console application. Working in local network, we need also an active DHCP server running. Several devices are trying to reach the server by broadcasting a socket message. The server is capable with technology of socket communication find out source IP address of sender device. Thanks to that server can assemble a table of active clients, and their properties (see Table 1).

2.3 Communication

The main purpose of the communication is a reliable firmware data transport from the server to the client. Take note that we need to transport very sensitive package of data, so the transmission must be error free. Every binary byte represents compiled program section, that means there is not space for mistakes like packet lose, packet sequence synchronization, bit swaps etc. Therefore, we must deal with the techniques that quadrant necessary protection, whether on the byte or packet layout, such as checksums, parities, Hamming codes, etc.

Communication utility can be described and analyzed on various layers and levels according to TCP/IP or OSI model [3]. Communication structure consist of these layers, and every layer has own data structure format called frame, and communication rules called protocol.

Since we have available in transport layer UDP protocol, we must implement superior security infrastructure in higher layer. We take an inspiration from the TCP transport protocol. Basic communication protocol functionalities required are the error detection and optionally an error correction. For the error detecting, we will use CRC – cyclic redundancy check method and parity [4]. Error correcting will be used indirectly by handshaking method and resending same packet multiple times, if the error is detected or after response timeout of packet acknowledge. It is a good practice to not send a large amount of data, in our case binary firmware, in a single data package – packet. Apart from that, embedded devices usually don't even have enough RAM memory to store large amount of incoming data. Therefore, we are going to use our custom protocol frame infrastructure (see Figure 2), for the safe data distribution.



MARK	SEQUENCE INDEX	TYPE	CRC	DATA SIZE	DATA BLOCK PACKAGE
[2 bytes]	[2 bytes]	[1 byte]	[1 byte]	[2 bytes]	[0 - 1000 bytes]

Figure 2 Communication protocol frame

The data block package inside packet frame structure in the most common case contains section of raw firmware binary file we want to transfer from server to client. But of course it is possible to use this format with another hierarchy of data. Literally the data block package may consist of another packet frame. It is useful when we are manipulating with structured data types.

Every packet contains its own sequence index number used for synchronization, but it is also necessary for a handshaking process. Handshaking concept in communication is that transmitter requires acknowledged response from the receiver. After the last packet has been delivered, we will check and compare CRC of the whole firmware.

2.4 Update management

Server has an ability for manual user management but can also run automatic update feature for multiple clients. After recognition the older client's firmware version, server begin with the transfer of the newer one.

Table 1 Example of clients list created by server

Client	ID	IP	Firmware version
0	2456122	192.168.1.15	13.9.2019 12:57:23
1	2456256	192.168.1.16	24.2.2020 08:12:03
2	2456319	192.168.1.17	26.2.2020 16:57:46

2.5 Software implementation

As we already know, client like embedded device needs very low level access and handling, due to its hardware close architecture, with minimal abstraction layers. Still valid, that C language with assembler elements is ideal choice for implementation. No other language offers tools for modifying main stack pointer register, what we actually needed for jumping in FLASH memory, among other important factors.

Figure 3 illustrates that an interface between the server and client is based on the socket communication. On the server side we have more freedom to choose an implementation platform. The most common backend implementations may be realized with C++, C#, python etc.

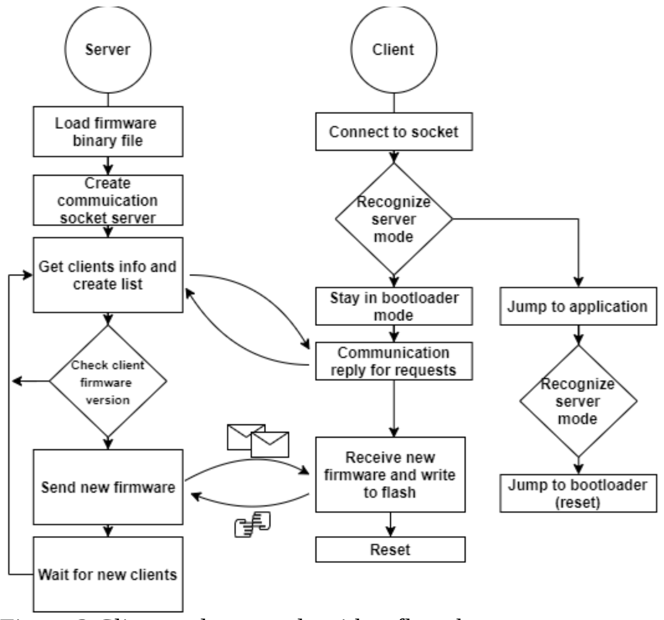

Server
Load firmware binary file
Create commuication socket server
Get clients info and create list
Check client firmware version
Send new firmware
Wait for new clients
Client
Connect to socket
Recognize server mode
Stay in bootloader mode
Communication reply for requests
Receive new firmware and write to flash
Reset
Jump to application
Recognize server mode
Jump to bootloader (reset)

Figure 3 Client and server algorithm flowchart

2.6 Possible future improvements

Let's imagine an IoT device application with access to global internet network. Whole management process could be handled by client, if actual firmware version can be downloaded from a cloud storage. That concept based on accessible firmware package from cloud is the main pillar for our process automatization. The cloud storage server may be accessible from local network as well as from global internet network. Internet communication bridge between LAN and WAN interface could be handled by NAT [5] and port forwarding techniques.

3 Results

This publication has been created as a summary of knowledge acquired during participating on a real commercial product in my part time job. Company, I worked in, was focused on automotive industry automation processes, especially quality measurement. In the Figure 4 is shown one of automation solution used in quality measurement of combustion engines heads.

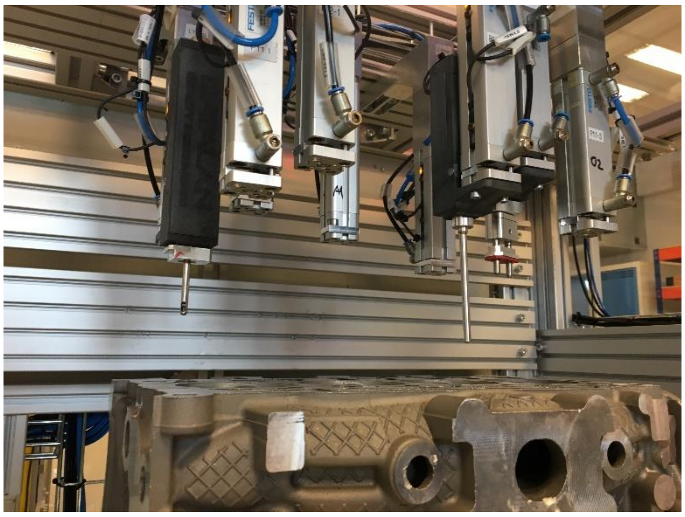


Figure 4 Clients - embedded devices mounted in machine for a combustion engine head quality measurement

Embedded clients, further called probes, were capable measure or read from analog and digital sensors and capture image from micro camera module with 20 FPS rate. The machine contains configuration of probes at specific positions to achieve measurement of required spots (see Figure 5).

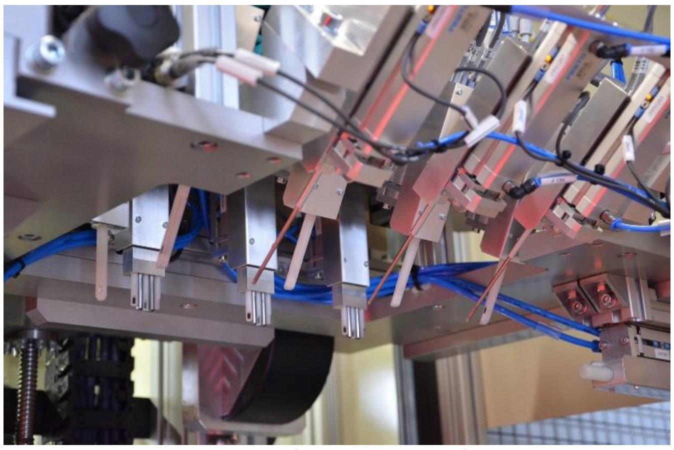


Figure 5 Collection of clients configured in quality measurement machine while operation

These probes had implemented remote firmware update management system described in this publication. During development process and operation, we have been able to upload new firmware version to all probes (5 – 20 depends on machine) in a short time and without mechanical manipulation.

4 Conclusion

In this publication we have described design and implementation of real and commercially used in industry remote firmware updates management system of multiple embedded or IoT devices. This system consist from one centralized server and a certain number of clients – embedded devices with functionalities to measuring analog or digital sensors, video streaming, actuator controlling. Firmware is control software of embedded device – client. During development or maintenance firmware version should be upgraded due to customer requirements or because bug fixing. In case the embedded device is in operation, mounted in complex machine miles away from development center, remote firmware management infrastructure is the only option.

Project is divided in 3 main steps, client implementation, server implementation and communication interface between each other. In our example case client represent STM32 ARM microcontroller, server is desktop backend console application and UDP socket technology has been used for communication. But analyzed concept is generally valid independently on hardware architecture.

Acknowledgments

I take this opportunity to express gratitude to my job colleague Ing. Lukáš Hrečko for management and expert advice in development process, I am also grateful to Ing. Richard Balogh, PhD for the expert consultation of this work.

References

[1] RICHA DHAM 2017. CYPRESS SEMICONDUCTOR *Over-the-air firmware upgrades for Internet of Things devices*. Available on internet: https://www.embedded-computing.com/embedded-computing-design/over-the-air-firmware-upgrades-for-internet-of-things-devices

[2] Roman Kleinerman, Daniel Feldman, 2011. *Power over Ethernet (PoE): An Energy-Efficient Alternative*. Marvell Available on internet: http://www.marvell.com/switching/assets/Marvell-PoE-An-Energy-Efficient-Alternative.pdf

[3] Vinton Cerf, Yogen Dalal, Carl Sunshine, 1974. *Specification of Internet Transmission Control Protocol*.

[4] Gupta, Vikas; Verma, Chanderkant, 2012. *Error Detection and Correction: An Introduction*. International Journal of Advanced Research in Computer Science and Software Engineering.

[5] François Audet; Cullen Jennings 2007. *Network Address Translation (NAT) Behavioral Requirements for Unicast UDP*.