Firmware and bootloader

Otilia Anton, Brice Gelineau and Jérémy Sauget

16 mars 2012

Contents

Introduction

The main purpose of this paper is to make a review on embedded bootloaders and firmware. Moreover, we intend to make the reader understand the use of the booting mechanism and to make him aware of the problems that are raised when modifying or installing firmware. We are focusing on safety, fault-tolerance and updating. We start by giving a brief definition of the main two concepts (bootloader and firmware) in Concepts section . The next section describes the main techniques in increasing reliability in embedded software. The Updates section describes an update design pattern and the actual implemented mechanisms for installing updates for firmware and bootloader. The last section presents an existing bootloader, encouraging the reader to hands-on experience with these platforms for a more thorough understanding.

1 Concepts

There are several definitions for firmware and bootloader. This paper focuses on embedded bootloader and firmware as the constraints in embedded applications (e.g. time-critical computing environment, low-power, fault tolerance or low-cost) differ from the traditional desktop systems. Each embedded system implements the approach that ensures its main function. For example, in house appliances there is a strong low-cost constraint whilst in aircraft systems timing and fault-tolerance are more important than costs.

1.1 Firmware

Firmware can be mainly referred to as being a fixed, rather small program that controls hardware in a system. Firmware is generally responsible for very basic low-level operations without which a device would be completely non-functional.

Usually firmware is stored in Flash, ROMs, PROMs, EPROMs. In most cases, firmware is not supposed to be changed by the user.

IEEE Standard Glossary of Software Engineering Terminology, Std 610.12-1990, defines firmware as follows: "The combination of a hardware device and computer instructions and data that reside as read-only software on that device. Notes: (1) This term is sometimes used to refer only to the hardware device or only to the computer instructions or data, but these meanings are deprecated. (2) The confusion surrounding this term has led some to suggest that it be avoided altogether."

1.2 Bootloader

The bootloader is the first code that is executed after a system reset. Its goal is to bring the system to a state in which it can perform its main function. This requires hardware initialization and choosing the correct image to load from flash. Because of its key role, the bootloader is usually placed in a part of the Flash that is protected from accidental erasure or corruption.

Hardware initialization may imply enabling access to RAM, setting up clocks and PLLs, and configuring other key peripherals. However, hardware initialization should be restricted to the essential at this level leaving the rest of the initializations for upper application code.

The steps following the hardware initialization in the booting process are further explained in the Updates section.

2 Reliability

A complex hand-written program nearly always contains errors, may it be a design fault or an incorrect implementation. Still, these bugs should not prevent the system in its whole to execute its mission. In order to attain that goal, several methods are commonly used.

2.1 Fault tolerance

Most of the methods presented in this section are implemented at the software side. Thus, they are only as good as the dedicated implementation. If the latter is faulty, the program will most likely behave incorrectly. Because of this, the code managing fault tolerance should be kept as simple and short as possible.

2.1.1 Acceptance tests

Acceptance tests consist in verifying the input or output data of a function. The test itself depends on the function it must work for. For example, in case of the computation of a square root, a simple test would be to square the new number. The comparison with the original number would give a natural test. In other circumstances, another possible test would be a range check. If the value is outside acceptable bounds, then it must be an error. This is especially useful to discard impossible or inconsistent values. In the previous example, a negative square root is probably incorrect. Finally, in case of two successive values which are supposed to be consistent, a possible test would be to verify that they are close to each other. This is often the case when dealing with sensors whose environment cannot change rapidly.

2.1.2 Wrappers

Wrappers allow to increase the robustness of a program. They work as interfaces between a module and the firmware. They verify both the inputs and the outputs of the wrapped software. It must filter any incorrect value that passes through it. So it prevents unmanaged or incorrect values to be given to a program and does acceptance tests to the output variables.

The quality of the wrapper relies heavily upon the quality of the tests, which is application dependant. It also relies on the availability of information from the wrapped components. It is for example impossible to do acceptance tests on a function managing sorted lists, if the lists themselves are not accessible. Finally, it relies on the extensive testing of the wrapped function. As the wrapper is supposed to filter the inputs and to reject values known to cause failures, if no test was done, these values cannot be discovered.

2.1.3 N-version programming

In addition to the previous solutions, another method heavily used in critical embedded systems is redundancy. The main goal of this solution is to avoid design, implementation and hardware defaults by relying on diversity. The system is made of several implementations with identical specifications that run in parallel. Each computation is done by each one of the implementations. Then, either a decider chooses the result or a consensus takes place. Either way, only one of the values is then used.

Because it relies on diversity, each implementation should be done by independent teams and run on different architectures. Still, it is possible to make each computation run sequentially, provided that each one is logically independent from the others. This method requires the function running in parallel to have precise specifications of what has to be implemented. Nevertheless, these must not be so specific as to require a specific design. If that was the case, it would defeat the goal of the method which is to keep sufficient diversity in the conception.

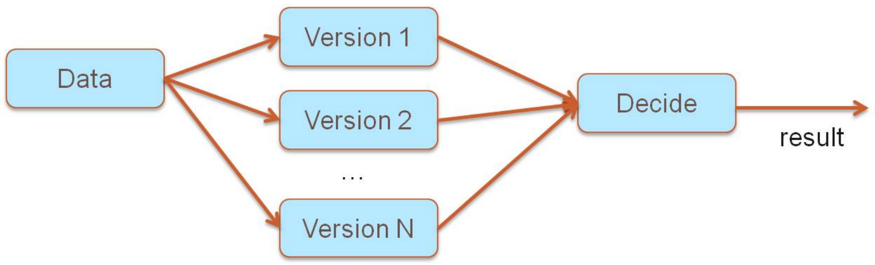

Figure 1: N-version programming

2.1.4 Backward recovery

Backward recovery relies on checkpointing and acceptance tests. If an error is detected at some point, the state of the system is taken back to the last save point. The task is then executed again, in order to avoid the occurrence of the error.

One of the best known backward recovery method is recovery blocks. In recovery blocks, the software is recursively divided in fault recoverable parts, each one with a well specified function. These blocks consist of at least two different implementations of the same functionality. When entering into a block, all the data needed by the function is saved and the first alternative is executed. At the end of the computation, the result passes an acceptance test. If it succeeds, the program goes on. Otherwise, the system rolls back to its previous state and tries the next alternative. If necessary, all the alternatives are sequentially tested. If none of them produces an error-free result, the block may signal an error to the calling function.

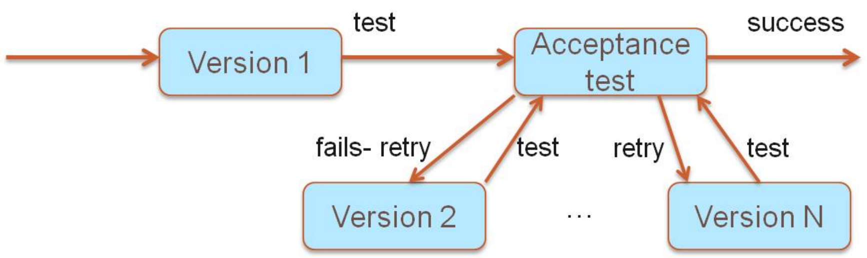

Figure 2: recovery blocks
idea from http://train-srv.manipalu.com/wpress/?p=130228

2.1.5 Forward recovery

Forward recovery is an alternative to the previous method. The software is still divided in fault recoverable parts but each of these parts has a single implementation and an acceptance test. At the end of the computation, the final value passes the test. If it succeeds, it is returned to the calling block. Otherwise, the system is put in a state that is considered as safe, depending on the error. This state can just return a default value but can also trigger some specific code. It is much faster than backward recovery but does not necessarily produce a value.

This method assumes that there are identified safe states depending on the detected errors. If the identification of the safe states is wrong, the error will not be corrected. Also, if the underlying fault is still there after treatment, the error can reoccur.

2.2 Virtualization

An aspect of the reliability which is more and more present nowadays is virtualization. This concept is based on spatial and temporal isolation. The supervisor, a special firmware, manages the system. It is able to resume and stop a program on the system and to make some verifications.

Thanks to isolation, if a guest operating system crashes, there is no impact on the entire system. The crash remains local and so all the system does not crash.

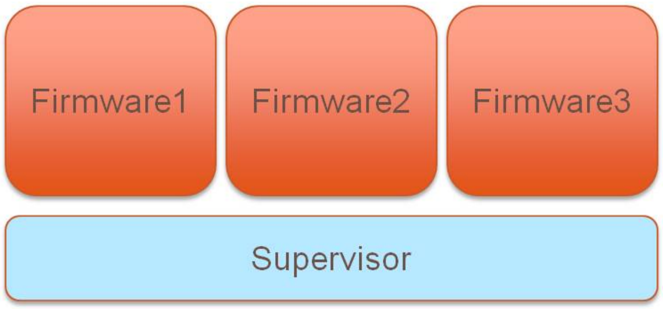

Figure 3: Virtualization

The supervisor can make lots of verifications. For example, it can check the right of a firmware image to access a memory area: this prevents a firmware from writing in another firmware memory area.

This method is used in aerospace for example. The difficulty consists in designing the supervisor for real time applications.

A team of computer scientists in the University of Salzburg on Austria (`http://cs.uni-salzburg.at/~hpayer/publications/iies09.pdf`) realized an example of real time supervisor on a Xscale processor (400 MHz, 64 Mo memory RAM).

2.3 Reboot

All the methods we have previously seen are to prevent from system failure. However, when the system is not responding, it is necessary to have some mechanism to reboot safely in a working state. For example, when a plane firmware crashes, it should not execute again all the instructions directed at the beginning of the fly. It has to memorize the situation continuously to be able to restart in a good situation.

The mechanism used to know when a system crash is called a watchdog. A watchdog is a timer which have to be periodically rearmed by the program. If the program fails to do it, it means that the system have crashed and a reset signal is sent. There are two different types of watchdog : internal watchdogs or external ones. The first one is a mechanism included in the processor. The second one is physically separated from the processor and so independent (Figure 4).

Some constructors design specific external watchdog timers. When a restart triggered by the watchdog happens, a specific information is given. This information generally consists in a register put at 1. In this case, the processor knows that it reboots due to a crash. It is able thus

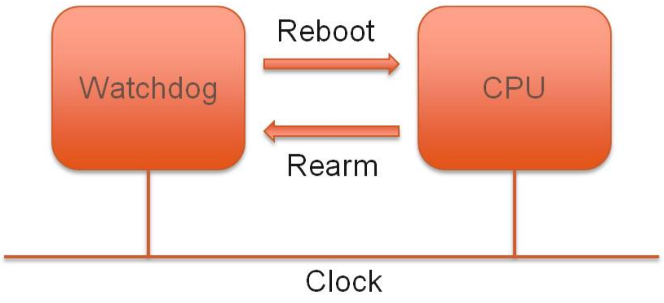

Figure 4: External watchdog

able to adapt its reboot process. Instead of restarting the system, the watchdog can just put a register to 1 or send an interruption. It gives the firmware the possibility to react to the problem without restarting and losing its informations.

When restarting, we have to know in which state the system must restart. In many applications, it is just impossible to restart the system at the initial state. The more common method used to prevent from this problem is saving the state of the firmware. At a fixed period, the state of the firmware is automatically saved. Hence, when the system crashes and then restarts thanks to the watchdog timer, the firmware can be in the last known state. But, saving state does not resolve all the problems. Indeed, all the operations made after the checkpoint are forgotten. It is not a problem if these are internal operations but, when these operations have consequences on the environment, it can be problematic. If the processor saves its state, begins configuring a peripheral and then crashes, when rebooting, it will have forgotten the beginning of the initialization and so, it will not give to the peripheral the right data to finish the initialization.

3 Updates

3.1 Updating a firmware : When ? Why ?

Even though firmware is not designed to be changed, updates in a bootloader or firmware are usually needed to correct bugs or to add new functionalities. Indeed, there are firmware that are not correctly designed and so contain bugs which can sometimes be critical. Sometimes, the firmware is just too old and does not comply with the client desires any more. In all these situations, a new firmware is needed.

However, all firmware cannot be updated. In critical systems like a plane control system, an update while flying is impossible.

There are plenty of methods for updating firmware. The more classical are using the UART or the USB port. But, if the system has an Ethernet controller, the update can be made on a network and on Internet too. For the mobile phones for example, the method used is called Update Over-The-Air: the new firmware is download thanks to wireless communication using a very specific protocol just made for the update operation. This show how important the update operation is on an embedded system.

So, updating a firmware is sometimes necessary. However, the update operation is critical because. If an unexpected problem occurs during the transmission and the updating process has not been designed accordingly, all the system can be blocked. This can be the case, for example, during the firmware update of a mobile, if connection is lost or if the processor crashes. The first embedded systems were totally blocked when this type of event occurred. Nowadays, there are some methods which have been imagined to realize safe updates on embedded systems. The most famous and logical one have been normalized in 2006 during the Europlop workshop. This method can be described by the diagram in Figure 5.

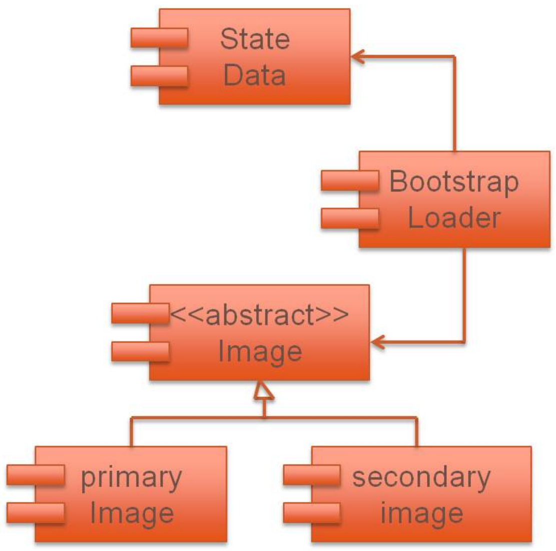

Figure 5: Firmware redundancy

The idea is to keep several firmware images in memory. So if one crashes when uploaded, the bootloader chooses another one.

This method is a general method. In the next paragraphs more information about applying this approach is given.

3.2 Problems

As mentioned, when performing an update on an embedded application several problems may appear. If ignored they lead to a non-functional device.

The main problems identified are:

- power failure during upgrade: the consequence of this is that new firmware is partially written.
- bad firmware: the consequence of this is that the device may stop functioning (e.g. bad firmware update for cameras)
- flash corruption: NAND flash have a certain ratio error which may prove to be significant while updates are written.
- communication errors: The consequence of this is partially written firmware.

3.3 Solutions

The most common solution is to always keep in Flash the firmware that was written before the device was sold or put in function. Whenever a new firmware is written ensure that the bootloader can switch to the initial firmware in case the new one fails. Figure 6 summarizes this. In case of failure the watchdog copies the values that indicate a reset because of the new firmware image. The values are checked by the bootloader at system reset and the bootloader chooses the firmware image that should be loaded.

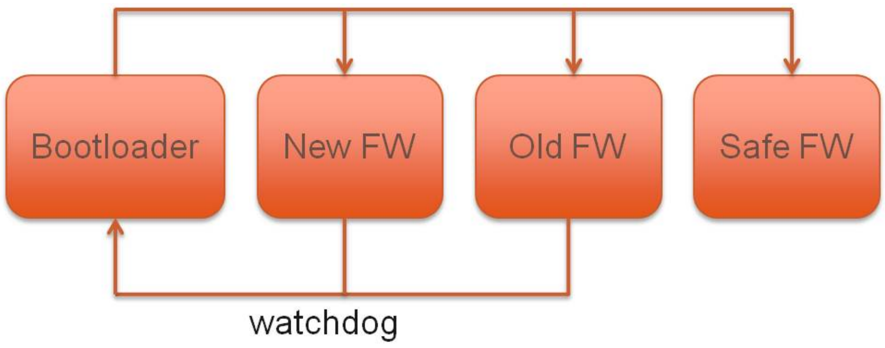

Figure 6: Safe update
idea from `http://mind.be/content/Presentation_Safe-Upgrade.pdf`

For Flash corruption errors, normally the Flash firmware (microcode) or error correcting code detect blocks with a higher ratio of failure and stops using them.

3.4 Booting steps

As explained in the Definitions section, after system reset the first step that is done is hardware initialization of the essential components (enable access to RAM, setting up clocks and PLLs).

Because of the possibility of Flash corruption or communication errors during writing, the next step is performing a checksum or a CRC(Cyclic Redundancy Check). If the computed checksum is not equal to the checksum found in the firmware, the system reboots and loads the safe firmware.

Another option at this step is to wait to receive a new flash ROM binary image through a suitable port on the device (through RS-232 port, USB port or Ethernet port). This means that the customer with a standard PC is able to run a program to send new ROM images to the device. Even though complex interfaces such as USB or Ethernet+TCP/IP are more tempting as they are more user-friendly it is better to use polled code and UDP instead of TCP. The lighter the solution the simpler to track problems afterwards.

If the checksum proved out to be correct, the bootloader should transfer control to the application by jumping to a fixed location in the application.

Except Flash ROMs with "read-while-write", the others cannot be read while being written. So if there is enough space in the RAM the updater can be copied to the RAM before being executed.

Also, if there is enough space in the RAM, the new image should be copied in the RAM. This allows storing the checksum of the new image before accessing the flash. Thus, it ensures a correct value to compare during checksum verification (it can also show if there were communication problems and the image was partially written).

3.5 Security

All the above is designed to cope with errors occurring while downloading the new firmware. However, it is advised to add a minimal security to the updating process.

The first thing to verify is that there were no transmission faults. This can be done by adding to the software its Cyclic redundancy check (CRC) value. The sender of the new program must compute the check value of the software and send it. Upon reception of the software, the device creates the CRC of what has been downloaded and compares it to the original CRC. If they are equal the program should be correct. Otherwise, it has been corrupted and it must be discarded.

CRC is commonly used and thus is broadly supported. Some micro controllers, including the STM32F1xx series, have an integrated unit to do it. Hardware support makes it a fast way to verify a software.

Another source of faulty software is a wrong compilation chain, especially if anybody can upload a new firmware. Adding a custom header at the beginning of the firmware provides protection against this. If the header is missing, the software has not been compiled by a dedicated chain of compilation. Thus it may have been targeted to a wrong architecture and be faulty.

Finally, one may want to protect its device against foreign programs. In this case, the new programs must be signed. The basic idea is that the programmer uses a private key to sign a hash of the program. The updater retrieves the hash and compares it to the hash of the new software. Again, if they differ, the program must not be updated. Obviously, this also provides protection against transmission faults. Even so, it doesn't mean that CRC can be used as a hash function, collisions being easy to make.

To be able to verify the signature, the device must know the public key of the constructor. This can be a problem if this key is stolen, as anyone could then send a new program. In order to prevent this, the firmware can use certificates. The list of supported keys can thus be easily updated.

3.6 FOTA example

Different approaches exist besides the one presented because of different memory sizes. However, the main idea is the same one: keep in memory several images of the firmware.

Upgrades to mobile phones, PDAs, and Tablet Computers use FOTA (Firmware Over the Air). In FOTA updates, the amount of memory in which firmware can be copied is huge: for example the platform Waspmote for Libelium has a 2GB SD card in which it stores several images. Only the firmware that is run is stored in flash memory. Figure 7 shows this architecture. The main idea is that when the programmer (normally the gateway) sends a new program, it is stored in the SD card. A second command, star_new_firmware for the Waspmote, is needed in order to use the new firmware. Then, the nodes copy the program from the SD card to the Flash memory and start the new program.

A great number of firmware images can be stored in the SD card. However, not all the SD cards can be used for OTA. The Waspmote has several memory storages:

- RAM (4KB): Volatile memory which saves the variables and instructions of the program being executed
- EEPROM (8KB): Memory used to store variables and certain flags used in the programs and which need to be kept after rebooting
- FLASH (128KB): Memory used to store the binary program which is currently running
- SD Card (2GB): High load memory system which uses FAT-16 file system. It allows to manage files which are controlled from an inode table. This is the place where all the new programs are stored.

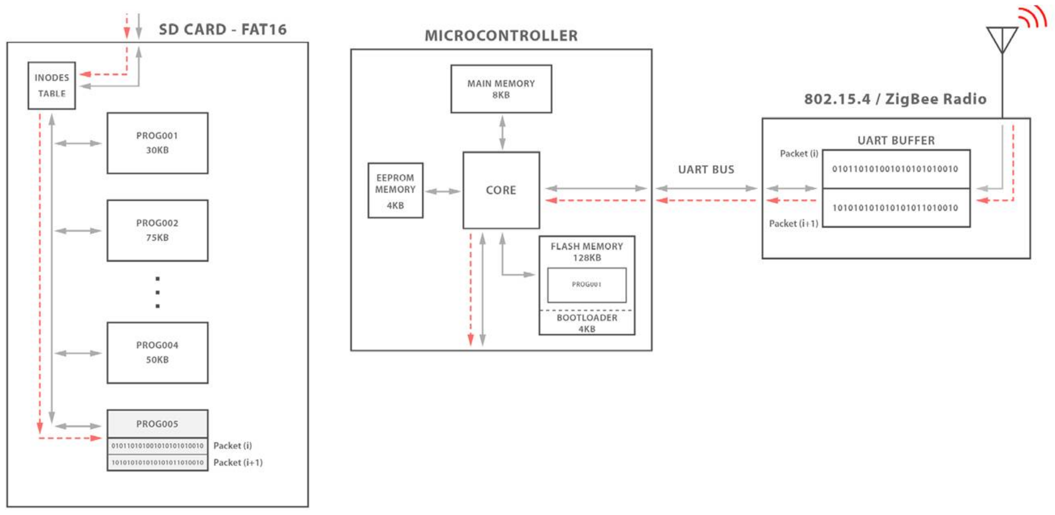

Figure 7: Waspmote memory and protocol for updates
image from
`http://www.libelium.com/documentation/waspmote/over_the_air_programming.pdf`

The Waspmote has a special file called "boot.txt" which contains references to the programs available on the device. Each time a new program is added a reference line is appended to the file.

Updates can be sent in unicast, multicast or broadcast and OTA has its own protocol for sending, selecting firmware version, scan nodes etc. However the steps for installing an update are consistent with the general approach presented at the beginning of this section. The main difference is the wireless communication.

- Locate the node to upgrade
- Check current software version
- Send the new program
- Reboot and start with the new program
- Restore the previous program if the process fails

4 Bootloaders

4.1 The Boot Sequence

The bootloader is the first program which is executed after a system reset. It must prepare the execution environment for the main program. Generally, it is the bootloader which allows us to update the firmware. All the tasks realized by the bootloader before calling the main program are part of the boot sequence. The following sequence is an example of classical boot sequence:

- configuration of the processor : it disables interruptions and configures the control registers
- initialization of the RAM : it initializes the RAM and allows the use of a programming language (stack pointer for programs in C for example)

- initialization of critical peripherals
- activation of the cache memory
- configuration of the MMU
- the interruptions can be enable

The interruptions must be disabled because it can stop the bootloader during the initialization. After hardware is configured, the bootloader starts the main program.

Contrary to the main program that can be executed either in RAM or in ROM, the bootloader has to be executed in ROM. When it is launched, RAM is not yet initialized. The only exception is debug mode, where the bootloader can be executed in RAM because another component initialized it. Obviously, only one part of the boot sequence has to be executed in ROM: after having initialized the RAM, the bootloader can copy itself on RAM and keep on executing.

It can also make a hardware discovery: in a computer for example, when you add RAM memory or PCI board, it is not necessary to specify it to the BIOS (the BIOS is one part of the complex bootloader of a PC). The BIOS will check the hardware before the initialization to know what it has to do. For PCI, the standard defines how to do hardware discovery.

4.1.1 Bootloader and security

The bootloader is a critical part of an embedded system because. If it is corrupted, there is no solution to start a program or to update a new bootloader. This is the reason why today, in all systems, there is a safe bootloader, stocked in ROM. The user cannot access this memory and so cannot modify it. In case of problem, the system can use this bootloader to start. Generally, it allows to download a new bootloader to fix the older one which has been corrupted. In the OMAP chips (designed by Texas Instruments), the safe bootloader is even written in the silicon.

4.1.2 Reboot mode

There are several different possibilities to configure the boot of a system: some static possibilities and some dynamic ones.

The first, and the easier, is the fixed reboot: this reboot can not be parametrized. It always does the same things. This mode is very easy to program but is more difficult to use because of its lack of flexibility : the boot sequence will always be the same.

The second is more flexible. It consists in watching the pin state when the system is booting to know what configuration to choose. For example, the user can decide to boot on flash memory or on RAM memory. In this case, the safe bootloader can be chosen by the user if necessary.

The third method is to command the boot thanks to an input. For example, use a SPI bus to communicate with a ROM, which can also stock the bootloader itself.

4.2 U-Boot

U-Boot is an open source bootloader. It is well-known in embedded systems because it can be used in a lot of hardware architectures. It is particularly adapted to boot an embedded Linux. U-Boot offers a lot of boot possibilities: users can access to a console command using UART, USB or Ethernet. This interactive program is started just before the first hardware initialization. U-Boot can update itself using the same mechanisms as the ones presented previously.

4.3 STM32F1xx

The STM32F1xx series provides an embedded bootloader programmed by ST which cannot be removed. It is in System memory, starting at position 0x800 0000 and can be used to reprogram the main flash memory if needed. It can be activated through various interfaces, depending on the device, but USART1 is always supported. In order to boot on this boot memory, BOOT1 must be kept low while BOOT0 is high.

References

[1] David Brenan. Embedded system update. `http://www.techrepublic.com/article/the-flash-rom-boot-loader-performs-critical-actions/5034893`, 2003.

[2] Silviu Craciunas, Christoph Kirsch, Hannes Payer, Harald Rock, and Ana Sokolova. Programmable temporal isolation in real-time and embedded execution environments. http://cs.uni-salzburg.at/ hpayer/publications/iies09.pdf.

[3] Chris Inacio. Sofware fault tolerance. `http://www.ece.cmu.edu/~koopman/des_s99/sw_fault_tolerance/`, 1998.

[4] Salecker Juergen. Embedded system update. `http://hillside.net/europlop/europlop2006/workshops/E2.pdf`, 2006.

[5] Embedded Software Division Mind. Safe upgrade of embedded systems. `http://mind.be/content/Presentation_Safe-Upgrade.pdf`, 2012.

[6] Jain Varun. Understanding embedded-system boot techniques. `http://www.edn.com/article/512530-Understanding_embedded_system_boot_techniques.php`, 2011.