The control loop applications in ARM boards

This document describes a particular application which is used as a skeleton for the EMS board: the control loop application. This application is built using the embOBJ framework and some HW timers provided by HAL. The HW timers offer a configurable but precise periodic control loop executed at maximum priority and the embOBJ framework offers more advanced services such as multitasking for lower priority tasks, UDP networking, error management etc.

The control loop executes in round robin three high priority tasks: one for reception, one for doing operations, and one for transmission. When the MPU is not involved in any of the above three tasks the scheduler of OSAL allows other tasks to execute. Typically those tasks are embedded in suited objects (the backdoor, the listener, the error handler, etc.).

Approval History

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Version | Author | | Date | Approved | | Date |
| 0.1 | Accame | iCub Unit |  |  |  |  |
|  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |

Revision History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Author | Comments |
| 0.1 |  | M. Accame | First edit of the document |
|  |  |  |  |
|  |  |  |  |

Table of Contents

[1 To be done 1](#_Toc376870202)

[1.1 To be done 1](#_Toc376870203)

[1.2 dxxwqcwe 1](#_Toc376870204)

[1.3 sdcewcxzc 1](#_Toc376870205)

[1.3.1 UDP protocol for EMS service 1](#_Toc376870206)

[1.3.2 UDP protocol for CAN gateway service 2](#_Toc376870207)

[1.4 cecwecwe 3](#_Toc376870208)

[1.5 Behaviour of a special eApplication: the eMaintainer 4](#_Toc376870209)

[2 Implementation details 5](#_Toc376870210)

[2.1 Embedded systems with multiple processes 5](#_Toc376870211)

[2.2 The eEsharedServices library 7](#_Toc376870212)

[2.3 The three processes in the EMS 8](#_Toc376870213)

[2.4 Example of a standard bootstrap 10](#_Toc376870214)

[2.5 How processes use the shared storage 11](#_Toc376870215)

[2.6 Detailed behaviour of the eLoader 12](#_Toc376870216)

[2.7 Detailed behaviour of the eUpdater 12](#_Toc376870217)

[3 Services offered by the eUpdater 14](#_Toc376870218)

[3.1 Queries and operations 14](#_Toc376870219)

[3.2 Firmware update in the EMS 14](#_Toc376870220)

[3.2.1 FW update of the eApplication 14](#_Toc376870221)

[3.2.2 FW update of the eLoader 14](#_Toc376870222)

[3.2.3 FW update of the eUpdater 15](#_Toc376870223)

[3.2.4 Joint FW update of the eLoader and eUpdater 15](#_Toc376870224)

[3.3 Firmware update of CAN devices attached to EMS 15](#_Toc376870225)

[4 Appendix 16](#_Toc376870226)

[4.1 Typical bootstrap sequence 16](#_Toc376870227)

# To be done

To be done.

## To be done

Bla bla bla.

BEHAVIOUR OF EMS AT POWER ON

eUpdater

WAIT MODE

Timeout of 5 seconds

SERVICE MODE

eApplication

EMS services

(info + FW update)

CAN gateway

(FW update of CAN dev)

Received a UDP packet on 3333

eLoader

**Figure 1**: The behaviour of the EMS at power on.

## dxxwqcwe

c wedcdcxswcxs.

## sdcewcxzc

b nytnty nt.

### UDP protocol for EMS service

dxewdwedw.

|  |  |  |
| --- | --- | --- |
| COMMAND | OPC | Description |
| CMD\_SCAN | 0xFF | The EthLoader sends it in broadcast to query existing devices.  PKT = {OPC}  The EMS sends back a reply of 14 bytes:  PKT = {OPC, D01, .. , D13}, where  D01: module->info.entity.version.major  D02: module->info.entity.version.minor  D03: BOARD\_TYPE\_EMS = 0x0A  D04-D07: IP net mask  D08-D13: MAC address |
| CMD\_CANGTW\_START | 0x20 | The EthLoader sends it to start the CAN gateway mode on EMS.  PKT = {OPC}  At reception, the EMS enters in CAN gateway mode. The EMS initialises the CAN1 and CAN2, sends twice the BOARD command over CAN to force the boards to enter in bootloader, and then it enables the communication CAN1 / CAN2 🡨🡪 UDP port 3334. The whole startup takes two seconds.  It sends back NOTHING. |

**Table 1** – UDP commands on service port 3333.

### UDP protocol for CAN gateway service

When the can cedcedcewcewqcweq. The UDP packets use the following protocol to exchange CAN frames between a host and the attached CAN boards.

UPD PACKET FOR CAN GATEWAY

UDP CAN GTW

HEADER

BODY

8 BYTES

16\* N BYTES

**Figure 2**: The UDP CAN GTW frame used for can gateway service.

HEADER OF UPD PACKET FOR CAN GATEWAY

HEADER

BODY

8 BYTES

SIGN

N

DUMMY

1 BYTE

1 BYTE

6 BYTES

Used to recognize a valid frame: 0x12

Number of CAN frames

**Figure 3**: The header of the UDP CAN GTW frame.

BODY OF UDP PACKET FOR CAN GATEWAY

HEADER

16\*N BYTES

CANFRAME

BODY

i = 1 .. HEADER.N

CANX

LEN

ID

DUMMY

DATA

1 BYTE

1 BYTE

2 BYTES

4 BYTES

8 BYTES

CAN1 (1) or CAN2 (2)

Of DATA field

CAN ID at 11 bits

CAN data

**Figure 4**: The body of the UDP CAN GTW.

## cecwecwe

Its behaviour can be completely defined by the user.

The only mandatory recommendations are in the construction of the project space. Then there are some advisable recommendations which ease the use of the system. See the following two tables.

|  |  |  |
| --- | --- | --- |
| Type | Importance | Description |
| Project space | Mandatory | 1. The code space must start at 72K and not go beyond 256K. 2. The data space must exclude RAM between 62K and 24K. 3. The code at address 72K must have 4 bytes with the address of top of stack and at address 72K+4 must have 4 bytes with the address of the reset handler.   What happens if condition is not satisfied:  Fatal behaviour.  For (1): below 72K there are the eLoader and the eUpdater.  For (2): the highest 2K bytes are used for inter-processing communication and using them can disturb the correct behaviour of the eLoader.  For (3): it is the convention for jumping operation done by the eLoader. |

**Table 2** – Behavioural mandatory recommendations for eApplication.

|  |  |  |
| --- | --- | --- |
| Type | Importance | Description |
| UDP restart mode | Advisable | The application should respond to command CMD\_RESET received on port 3333.  What happens if condition is not satisfied:  The eUpdater can be executed only by setting power off and on or with a method non-compatible with the EthLoader protocol. |
| Application info | Advisable | At code address 72K+512 there should be placed a variable *applicationInfo* of type const struct eEmoduleInfo\_t (of size 64 bytes) which contains information about the application (name, build date, version, code space, resources, etc.).  What happens if condition is not satisfied:  The eUpdater shall give incoherent information about the eApplication upon reception of command CMD\_PROCS |
| Other 3333 commands | Advisable | The application should respond to commands CMD\_SCAN, CMD\_PROCS, and CMD\_UPD\_ONCE received on port 3333.  What happens if condition is not satisfied:  The EMS is not seen by EthLoader and/or does not send it the required information. |

**Table 3** – Behavioural advised recommendations for eApplication.

## Behaviour of a special eApplication: the eMaintainer

The eMaintainer behaves quite exactly as the eUpdater except for the CAN gateway feature.

It is used to perform FW update of the eUpdater.

Implementation details

They are in the following.

## Embedded systems with multiple processes

We refer to a process as a normal program with a given code and memory space which starts execution from its reset handler. The reset handler typically has some start-up code which initialises the reserved RAM and then it calls the main() function.

At power on, but in general at the exit of a reset state, the CPU executes the reset handler of the process positioned at the default address[[1]](#footnote-2). Most common embedded systems only have a single process which at some point enters in a forever loop and executes some actions. The single process, also called application, typically can use the whole FLASH and RAM.

SINGLE PROCESS SYSTEM

CODE SPACE (FLASH)

0x08000000

startup();

reset\_handler()

ADRofTopOfStack

main();

main()

{

<do something>

<do something else>

for(;;);

}

RAM

// inits the entire RAM

**Figure 5**: The normal embedded system. There is a single process which is executed starting from its reset handler located at default address.

Other systems use two processes: (a) the bootloader which is executed at exit of reset state, (b) the application which is launched by the bootloader with a simple jump to the relevant reset handler[[2]](#footnote-3).

The bootloader and application must have disjoint code spaces located in separate partitions of the FLASH but can use the same RAM as that is cleared and initialised by the start-up code. However, if they exclude from initialisation a small portion of RAM they can use it for inter-processing communication. The application writes some data in the shared RAM and force a reset. At this point the bootloader finds in shared RAM the data written by the application and behaves accordingly. The mechanism works also in the other direction: the bootloader can issue some orders to the application by means of the shared RAM.

TWO PROCESSES SYSTEM

CODE SPACE (FLASH)

bootloader

0x08000000

startup();

reset\_handler()

application

main();

main()

{

<do something>

<read jump address>

jump();

}

reset\_handler()

main();

main()

{

<do something>

<do something else>

for(;;);

}

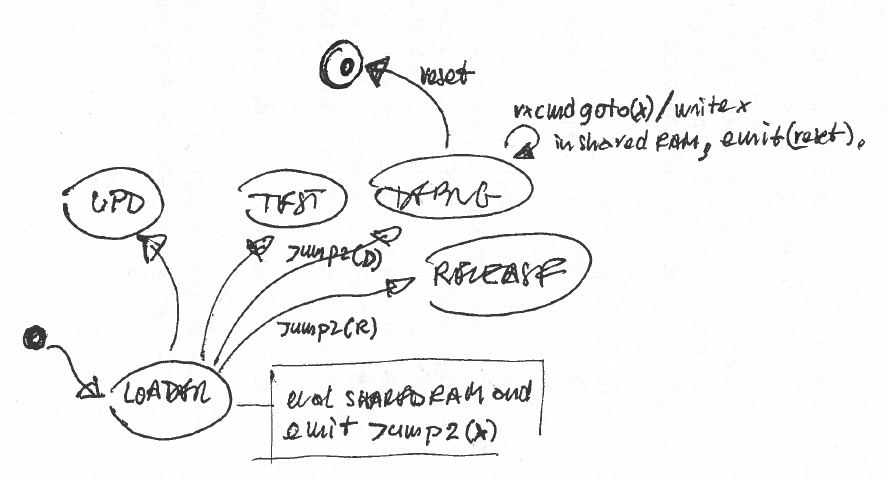
startup();

RAM

ZI (initialized by startup() of running process)

NZI (shared)

**Figure 6**: The embedded system with bootloader. The bootloader is executed at default and has responsibility to perform some actions and then to load the final application. The application can force a new execution of the bootloader by asserting a reset. A small portion of RAM is not zero-initialised and can be used for inter-processing communication between the two processes.



SCHEME OF A MULTI- PROCESS SYSTEM

**Figure 7**: An embedded system with 5 processes. The loader executes one of four processes: updater, test application, debug application, release application. It is possible to switch from one process to another at the reception of a network command (shown only for debug process).

In general, it is possible to have N of such processes and force the execution of one from another. From the booting process (loader) it is possible to force execution of any another process with a simple jump to the new reset handler. The execution switch from any other process to another can be safely done passing through the loader: (a) the starting process writes in shared memory the address of the final process, (b) it forces a reset, and finally (c) the loader jumps to the reset handler of the target process whose address has read from shared memory.

With this mechanism it is possible to implement complex yet flexible systems. For instance a process may perform FW update, one implements test of the device, one is the debug application and another one the release application. In presence of a communication link, e.g., IP or CAN, one can decide with a simple message which process must be executed.

## The eEsharedServices library

The EMS has data structures which are shared amongst all the processes. This data structure is manipulated using the eEsharedServices library which offers access to the following memory regions.

* *boardInfo*: contains RO permanent information about the HW board, such as version number, release date, unique identifier, etc.;
* *deviceInfo*: contains RW permanent information related to how the board is used as a device, such as MAC address, IP address, CAN features, user-defined data, etc.;
* *partitionTable*: contains RW permanent information about all the processes running on the board (such as name, version, release date, memory map, etc.). It also contains the process loaded at power on by the eLoader (STARTUP), the process which eventually must run[[3]](#footnote-4) (DEF2RUN)
* *IPCdata*: contains RW volatile information used for inter processing communication, such as orders for the loader about which process it must launch (ORDER).

The above memory regions are mapped on the EMS as in the following figure.

The eEsharedServices library is the only way to manipulate the shared memory. To insure the same behaviour, each process must be compiled with the same C code to generate code which is replicated in the code space of each process.

MAPPING IN EMS OF SHARED MEMORY

CODE SPACE (FLASH)

EEPROM

Shared storage

RAM

Shared RAM

IPCdata

partitionTable

boardInfo

deviceInfo

process

eEsharedServices

RAM for the process

EEPROM available to processes

<some code>

**Figure 8**: The volatile shared memory is mapped in RAM which is NOT zero initialised at start up and cannot be directly used by any process. The permanent shared memory is mapped in an EEPROM region not directly used by any process. The eEsharedServices library is used as the only way to manipulate the shared memory. It is executed in the same code space as the calling process.

An alternative is using the so called shared library solution, for which it is used a unique code partition for eEsharedServices. A call to eEsharedServices from within any process generates a jump to this code partition. This solution does not duplicate code space and avoids that an application accidentally uses a wrong C code version. However, it is more difficult to manage a change of the eEsharedServices because we need to update also all the processes which use it.

ALTENATIVE MAPPING IN EMS OF SHARED MEMORY (NOT USED)

CODE SPACE (FLASH)

EEPROM

Shared storage

RAM

Shared RAM

IPCdata

partitionTable

boardInfo

deviceInfo

process

eEsharedServices

RAM for the process

EEPROM available to processes

<some code>

**Figure 9**: The volatile shared memory is mapped in RAM which is NOT zero initialised at start up and cannot be directly used by any process. The permanent shared memory is mapped in an EEPROM region not directly used by any process. The eEsharedServices library is used as the only way to manipulate the shared memory. It resides in a dedicated unique code space which is shared by all the processes.

## The three processes in the EMS

The EMS operability is performed by three processes: the eLoader, the eUpdater, and the eApplication.

The eLoader is executed at exit of reset state and its main role is to run next process, (the eUpdater or the eApplication) and to enter in error mode if no valid process is found in the EMS.

The eUpdater is loaded by the eLoader either at bootstrap or by request of the eApplication. Its main role is to perform updating of its firmware and of the firmware of the attached CAN devices.

The eApplication is launched by the eLoader and is the process which gives standard operability of the EMS.

The way the processes are mapped onto the EMS is the following.

MAPPING IN EMS OF MAIN PROCESSES

CODE SPACE (FLASH)

eApplication

eLoader

eUpdater

EEPROM

shared storage

RAM

shared not ZI RAM

Used by the running process

Used by eApplication only

eEsharedServices

eEsharedServices

eEsharedServices

MEMORY-MAPPED HW PERIPHERALS

Used by the running process

**Figure 10**: When a process runs it uses its reserved code space in FLASH. All the processes share a small portion of RAM and a small portion of EEPROM for services offered by a shared-memory-manager library. They are for instance communication to the eLoader to jump to a given process, or the storage of the partition table of the EMS. Apart from the shared RAM, the running process uses the entire remaining RAM. And the eApplication has exclusive use of all the remaining EEPROM.

MAPPING IN EMS OF MAIN PROCESSES (NOT USED)

CODE SPACE (FLASH)

eApplication

eLoader

eUpdater

EEPROM

shared storage

RAM

shared not ZI RAM

Used by the running process

Used by eApplication only

eEsharedServices

MEMORY-MAPPED HW PERIPHERALS

Used by the running process

**Figure 11**: When a process runs it uses its reserved code space in FLASH. All the processes share a small portion of RAM and a small portion of EEPROM for services offered by a shared-memory-manager library. They are for instance communication to the eLoader to jump to a given process, or the storage of the partition table of the EMS. Apart from the shared RAM, the running process uses the entire remaining RAM. And the eApplication has exclusive use of all the remaining EEPROM.

## Example of a standard bootstrap

The following example shows what happens in a standard bootstrap in which the eUpdater is executed for 5 seconds and then it is executed the eApplication.

STANDARD BOOTSTRAP: 1. FROM ELOADER TO EUPDATER

CODE SPACE (FLASH)

eLoader

eApplication

0x08000000

startup();

reset\_handler()

ADRofTopOfStack

eUpdater

0x08004000

0x08018000

main();

main()

{

<do something>

<read jump address>

jump();

}

reset\_handler()

reset\_handler()

**Figure 12**: The boot phase. At reset exit, it is executed the process at default address: the eLoader. The eLoader reads the address of the STARTUP process and just jump to it. In the example the STARTUP process is the eUpdater.

STANDARD BOOTSTRAP: 2. FROM UPDATER TO APPLICATION

CODE SPACE (FLASH)

eLoader

eApplication

0x08000000

startup();

reset\_handler()

ADRofTopOfStack

eUpdater

0x08004000

0x08018000

main();

main()

{

<do something>

<set jump address as

0x08018000 >

reset();

}

reset\_handler()

reset\_handler()

jump()

<some code>

**FROM PREVIOUS FIGURE**

**Figure 13**: The execution of the application. After some operations, the eUpdater wants to force execution of the DEF2RUN process (the eApplication in our case), thus it (a) sets the jumping address for the eLoader equal to the address of eApplication, and (b) force execution of the eLoader by issuing a reset.

## How processes use the shared storage

The eLoader verifies validity of the shared storage and if not coherent it resets it. That may happens because it is its first execution and EEPROM is blank or because someone accidentally erased EEPROM or even because there has been a change of the binary format of data.

If a reset is required, the eLoader performs the following operations.

* It prepares the partitionTable for a maximum of 8 processes (eLoader, eUpdater, eApplication, and other five, e.g., test or debug). It inserts inside the partitionTable an entry for itself, sets the eUpdater as the STARTUP process and also as the DEF2RUN process.
* It prepares the boardInfo with information which is specific to that board and also assigns a unique ID using the feature embedded in the STM32F107 MPU.
* It prepares the deviceInfo, sets the MAC address using the IIT OUI[[4]](#footnote-5) and the unique ID. It sets the IP address to 10.0.1.99 and the IP mask to 255.255.255.0.

The eLoader uses the volatile memory IPCdata section to retrieve orders from any process. If the eLoader finds an order to jump to a given process (ORDER), so it does.

If there are no jump orders the eLoader retrieves the startup process from partitionTable and jumps to it. By default the startup process is the eUpdater.

Any other process, when it runs, verifies the validity of the shared storage and if it is not coherent then it resets it[[5]](#footnote-6). Then it performs the following.

* It verifies that inside the partitionTable there is an entry for itself, and if not it adds it.
* It retrieves the MAC and IP address and IP mask and use them to activate its IP stack.

In addition to standard use, The eUpdater, when in service mode, can perform operations such as read the partitionTable to tell the host what is loaded, change it (add/change eApplication), read and write deviceInfo (MAC and IP addresses, IP mask). The eUpdater also uses the volatile memory IPCdata section to order the eLoader to jump to the next process before forcing a reset.

In addition to standard use, the eApplication can perform operations such as read the partitionTable to tell the host what is loaded, and use the volatile memory IPCdata section to order the eLoader to jump to a given process (typically the eUpdater) before forcing a reset.

## Detailed behaviour of the eLoader

The eLoader behaves as in the following figure.

**Figure 14**: Behaviour of the eLoader. The eLoader is executed only for a small fraction of time unless it enters in error mode, where it blinks LEDs at 10 Hz.

## Detailed behaviour of the eUpdater

The eUpdater behaves as in the following figures.

**Figure 15**: Behaviour of the eUpdater. The eUpdater shows its wait mode with LEDs blinking at 1 Hz and its service mode with LEDs blinking at 0.5 Hz.

**Figure 16**: Behaviour of the eUpdater when it enters in service mode. The thread which processes UDP messages from port 3333 is always active. The thread which performs CAN gateway on port 3334 is active only after the former thread activates it upon reception of a specific activation UDP message on port 3333.

Services offered by the eUpdater

Services are offered when the eUpdater is in service mode.

It is possible to force execution of eUpdater in service mode in several ways: (a) by sending a command CMD\_UPD\_ONCE to the EMS which runs any process[[6]](#footnote-7), (b) by sending a command CMD\_RESET to the EMS which runs any process and sending any command to its port 3333 by 5 seconds, (c) by switching the EMS power off and then on and sending any command to its port 3333 by 5 seconds.

The application EthLoader uses method (a).

## Queries and operations

When the EMS is in service mode, it is possible to query about the loaded processes and to change some parameters (IP address, etc.). The application EthLoader is able to do so.

## Firmware update in the EMS

FW update can be done when the EMS executes the eUpdater in service mode and there is a remote host which can communicate with the proper protocol. The application EthLoader is able to do so.

There are several cases.

### FW update of the eApplication

The FW update of the eApplication when no change is done on major number of eEsharedServices can be safely done by forcing execution of the eUpdater in service mode and using commands of UDP port 3333.

If instead the new eApplication uses a new version of the eEsharedServices with a change in major number, then it is required to update the eLoader and the eUpdater first.

When in service mode, it is possible to program the code space of the eApplication with commands CMD\_START, CMD\_DATA, and CMD\_END.

The program operation shall force the STARTUP to eUpdater so that a successive reboot shall execute the safe process eUpdater. It is advisable entering the service mode and use command CMD\_PROCS to verify the presence of the new eApplication.

Then, in order to test correct execution of the eApplication, it is necessary to insure that DEF2RUN is eApplication with command CMD\_BOOT with argument eApplication. And send a CMD\_RESET.

### FW update of the eLoader

The update of the eLoader is a risky operation because a not working eLoader which is not able to launch the eUpdater shall isolate the EMS from the network and only a JTAG reprogramming will solve.

A FW update of the eLoader where there has been no change in major number of eEsharedServices can be done using the eUpdater.

If instead the new eLoader uses a new version of the eEsharedServices with a change in major number, then it is advised to update the eLoader and the eUpdater together (see later).

### FW update of the eUpdater

A FW update of the eUpdater where there has been no change in major number of eEsharedServices can be done using a special application called eMaintainer, which offers the same services as the eUpdater (except for the CAN gateway) but which can update also the memory space of the eUpdater.

The eMaintainer must be loaded as a normal eApplication. When it runs, it is possible to program the code space of eUpdater with commands CMD\_START, CMD\_DATA, and CMD\_END.

The program operation shall force the STARTUP to eApplication so that a successive reboot shall execute the safe process eMaintainer.

### Joint FW update of the eLoader and eUpdater

A FW update of both eLoader and eUpdater is required when the eEsharedServices changes its major number. The operation can be done using the eMaintainer.

The presence of processes which manage incompatible eEsharedServices storage is not a problem because every process shall adjust the storage to its liking if it finds it incompatible. In this case, however, everything shall be reset thus some strange effects can happen such as the IP address to be found at value 10.0.1.99.

In short here are the suggested operations: (1) load the new eMaintainer and force a reset; (2) the loader shall execute eUpdater and then eMaintainer which reset EEPROM and use address 10.0.1.99 and some strange MAC; (3) load the new eLoader and eUpdater; (4) verify with the PROC command; (5) impose a reset: now it starts the eLoader which finds a good EEPROM and then launches the eMaintainer with a good MAC and IP 10.0.1.99; (6) change the IP address and then jump to eUpdater and load the new eApplication.

## Firmware update of CAN devices attached to EMS

FW upgrade of attached CAN devices can be done when the EMS executes the eUpdater in service mode and enables the CAN gateway with command CMD\_CANGTW\_START. At this point a remote host which can communicate with the proper protocol over port 3334 can send and receive CAN frames.

The application CanLoader is able to do so.

Appendix

In here it is contained extra information.

## Typical bootstrap sequence

At power switch on, the eLoader loads the eUpdater. The eUpdater enters in wait state for a maximum of 5 seconds.

If no message arrives, then after the 5 seconds the eUpdater writes in a shared memory part the order for the eLoader to load the eApplication and forces a reset. At this point, when the eLoader executes it finds the order to load the eApplication. If this process exists in its reserved memory address, then it loads it. Otherwise it loads the eUpdater with the order to drop the wait state and enter directly the service state.

Instead, if any UDP message arrives, the eUpdater enters a service state and offers firmware upgrade and information about loaded processes.

The eUpdater shows its wait state with LEDs blinking at 1 Hz and its service state with LEDs blinking at 0.5 Hz.

NORMAL BOOTSTRAP SEQUENCE WHICH LOADS APPLICATION

eApplication

eLoader

(1) first reset after power on

(2) it jumps to STARTUP process (typically the eUpdater)

(4) the eUpdater reads from partition table in EEPROM which is the DEF2RUN process (typically the eApplication), writes in shared RAM the order for the eLoader to jump to it, and then forces a reset

(6) it retrieves order to jump to eApplication, verifies the eApplication exists, it jumps to it

(5) second reset after power on

(3) the eUpdater stays in wait state for 5 seconds

(7) we stay forever in eApplication

eUpdater

wait

load

**Figure 17**: In the normal bootstrap case, the eUpdater is executed for 5 seconds, and then the eApplication starts. Both processes are loaded by the eLoader.

NORMAL BOOTSTRAP SEQUENCE WHICH ENTERS SERVICE MODE

eUpdater

(1) first reset after power on

(2) jumps to eUpdater

(3) the eUpdater stays in wait state for up to 5 seconds

(4) a UDP packet arrives and the eUpdater switches to service mode

wait

service

(5) it stays in service mode forever or until a specific UDP commands orders to bootstrap

eLoader

load

**Figure 18**: In the normal bootstrap case, if the eUpdater receives a UDP command within 5 seconds, then it enters the service mode.

1. The default address on EMS is 0x08000000 and is also remapped on 0x00000000. [↑](#footnote-ref-2)
2. Actually some other tricks are used, such as set the new position of stack in the relevant registers of the MPU. [↑](#footnote-ref-3)
3. It is the one which the eUpdater orders the eLoader to run after the 5 seconds have passed. See later on. [↑](#footnote-ref-4)
4. Actually, now it is 0x000002 [↑](#footnote-ref-5)
5. The process may find the storage inconsistent only if the major number of the release version of its linked eEsharedServices library is different from the one of the process which run earlier which is the eLoader. [↑](#footnote-ref-6)
6. If the process is the eApplication, then it must satisfy the advised recommendation to behave correctly to the command CMD\_UPD\_ONCE received on port 3333. [↑](#footnote-ref-7)