New Bootloader for the Imote2 platform
Technical Report 373/09

Markus Engel, Marc Krämer
{m_engel,kraemer}@informatik.uni-kl.de

University of Kaiserslautern, Germany

Abstract. Bootloaders are widely used to startup one or more operating systems. On embedded systems like the Imote2, bootloaders are used to write the executable to the nonvolatile memory and, if a correct image is found, run the executable (operating system). The hardware initialization needed to get an image or write the image, in contrary to a PC architecture, is done by the bootloader. This report describes a bootloader which is fail save and its codeloader is platform independently written in Java.

1 Introduction

Boot-strap-loaders (named bootloaders) are used to start an operating system (OS) on a hardware platform. In theory it is possible to place the operating system directly at the start point and boot the operating system. On a PC platform the master boot record (MBR), which is holds the startup code is only 512 bytes wide, and only 440 bytes are usable. The first and essential parts to boot the operating system have to fit in these 440 bytes. On the other hand it is sometimes needed to boot an alternative OS or change some startup parameters like the commonly known GRand Unified Bootloader [1]. On a embedded hardware (sensor nodes) like the Imote2 [2], there is no such restriction with the MBR. On PCs the OS is separated form the application and can be exchanged independently. On sensor platforms it is common to embed the operating system in the application. Due to this fact, the exchange of the application can't be done by using the operating system. It has to be done by the use of special hardware or at boot-time by the bootloader.

2 Naming conventions

In Fig. 1, you can see the principle parts to load an application to the Imote2. First you start on the PC side by compiling the application and the operating system code with a cross compiler. A cross compiler is a compiler which generates code for the target hardware on a different host system, the code is not runnable on the host but on the target hardware. In our case we use a gcc for the arm [3] platform. The result is an arm-executable in the executable and linking format (elf). After stripping the elf-file to bare assembler instructions, we call this file the image. For the transfer (upload) of this image to the Imote2, on PC side and on Imote2 side an application is running to perform the transfer. The application on PC, we call the Codeloader, which reads the file from disk and controls the transfer to the Imote2. On the Imote2 the reception of the image is done at boottime by the bootloader.

The whole transfer and the erasure of the old application is controlled by the codeloader. The bootloader uses a simple state machine and performs the received commands. After the successful upload of the image, a reboot is performed and the image is executed.

3 Comparision with the preinstalled Bootloader

On the Imote2, there already exists a preinstalles bootloader and a codeloader [4]. So, why is there is neeed for a new developed boot- and codeloader and not the reuse of existing ones?

3.1 Original Code-/Bootloader

For the codeloader, the answer is the old codeloader was developed to run in cygwin [5] under MS Windows [6], so it is not a windows program nor a linux/unix program. Our goal was a program running on Windows, Linux,

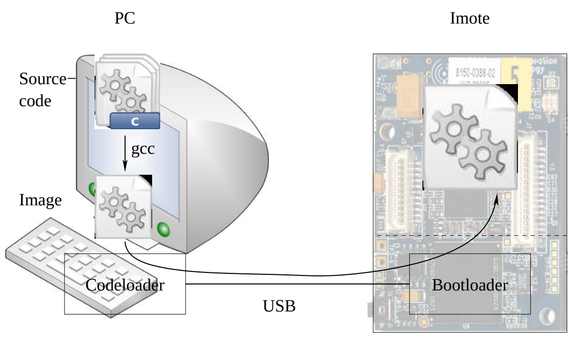


Fig. 1: Codeloader vs. Bootloader

and if possible on Mac [7].
The bootloader has more disadvantages: It was developed for the use of tinyos, and since tinyos is using only C and not C++, there was no need for that.
In Table 1 the memory layout of the original, preinstalled bootloader is shown. The original bootloader resides at address of 2 MiB in flash and copies the image at the location in between 0 and 2 MiB – so the maximum size is limited to 2 MiB. Another fact is the double copy concept. A new image received by the bootloader and written to a primary or a secondary flash location. The locations alternate if the last written image was successfully transferred. After a successful transfer the image is copied to the start address `0x20`. A major design error is the size of the second image location, which is only 1 MiB large, followed by essential boot information at `0x01E00000`. By successfully uploading an image larger than 1 MiB twice, you can assure to make your Imote2 unusable.
During the intensive use of the Imote2, we detected a very low programming speed of less than 20kb/s. Even this is not a problem for small programs, it is for larger ones. Additionally the upload process sometimes stops, having only flashed half of the image. The time waste for both were immense during development and testing sessions.
At least, the documentation given to the original bootloader is incorrect and not up-to-date. It represents an early development stage. Some of the effects encountered on the platform couldn't be explained with that documentation. The memory layout shown in Table 1 is only one example, which was detected by reverse engineering the source code and memory content of an Imote2.

Table 1: old memory layout

Memory Location	Size	Description
`0x00000000 - 0x0000001F`	32 Byte	Boot Vectors
`0x00000020 - 0x001FFFFF`	2 MiB	Programm
`0x00200000 - 0x002FFFFF`	1 MiB	Bootloader
`0x00300000 - 0x01BFFFFF`	26 MiB	Unused
`0x01C00000 - 0x01CFFFFF`	1 MiB	Primary bootlocation
`0x01D00000 - 0x01DFFFFF`	1 MiB	Secondary bootlocation
`0x01E00000 - 0x01EFFFFF`	1 MiB	Adress-/Attributetable
`0x01F00000 - 0x02000000`	1 MiB	Unused

3.2 New Codeloader

The goal for the new Codeloader is a platform independent base, which is executable at least running MS Windows, Linux and if possible Mac OS X. For the application, Java[8] is platform independent and runs on all these platforms. The direct hardware access to the usb interface from Java is realized using libusb [9], which is also available for the aforementioned operating systems.

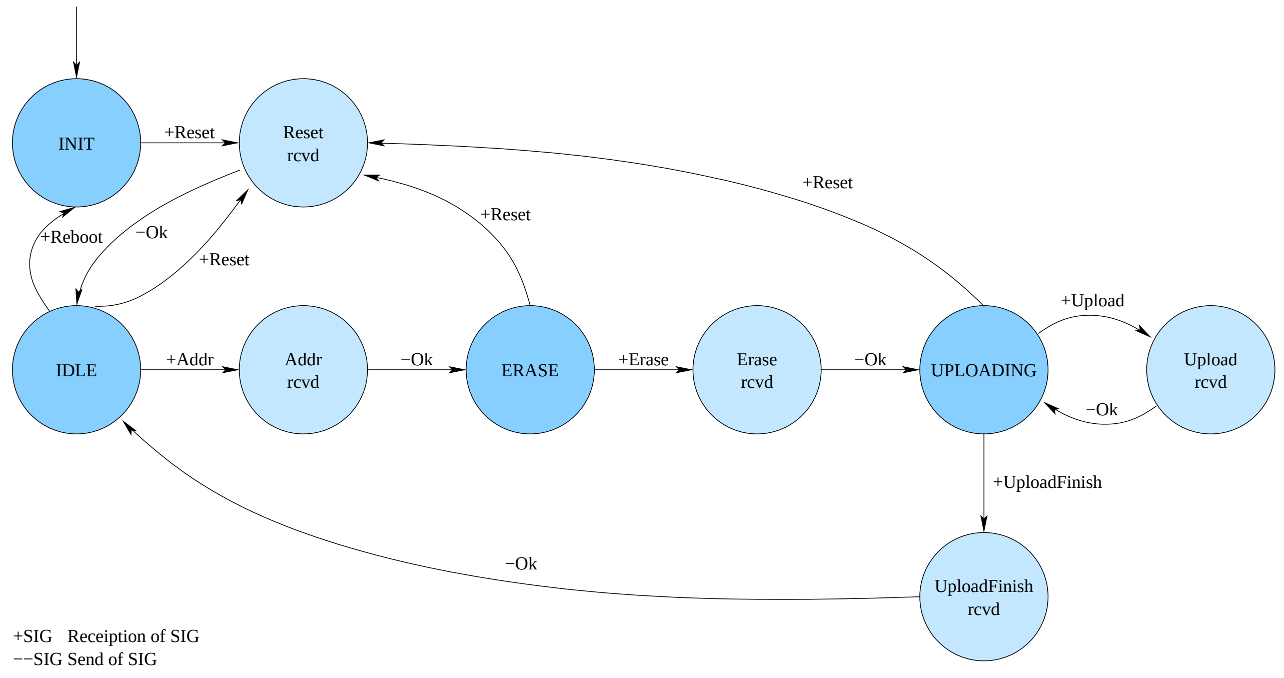


Fig. 2: communication protocol bootloader-side

Figure 2 shows the state machine protocol of the bootloader which is addressed by the codeloader. The Bootloader starts in `INIT` and waits for a `Reset` signal of the codeloader acknowledged by a `Ok` and a change to the state `IDLE`. The codeloader sends the address for the transfer and its size which leads to state `ERASE`. After issuing `Erase`, the space for the data is erased and the state machine ends in state `UPLOADING`. Now the codeloader transfers the whole image to the Imote2 and sends `UploadFinish`, which brings the Imote2 back in state `IDLE`. Now a new transfer can start or the node can be reset to boot the new image. Note that codeloader is able to write at almost any address of the Imote2, the address check is only done in the codeloader! By this you are able to write all data on adresses $\geq$ `0x00040000`, which corresponds to the new memory layout as seen in Table 2. The codeloader is available for download[10].

For a larger amount of nodes it is useful to ensure, that a specific image is programmed on only one specific node. Our new codeloader has built-in support to check if a specific Imote2 is connected to the PC, and writes the data only to this node. If your changing more than one node at a time, this helps you not getting confused.

One advance of implementing the codeloader in Java was the ease of porting it to a client/server architecture, by using RMI (Remote Method Invocation). In terminal server environments, a server is running on each client, waiting for an image to upload, while the client is started on the terminal server, providing the image. On Linux the display variable can be used directly to connect to the client and start the upload.

3.3 New Bootloader

The implementation of the new bootloader was straight forward from the point of the communication state machine, as described in the last section. On the platform a new, more flexible memory layout was choosen (see Table 2). For the bootloader a size of 256 kiB is reserved (currently the bootloader needs only 100 kiB). We added $1\frac{3}{4}$ MiB for user configuration data. The intention of this section is to have parameters for a node at a specific address. The configuration section can be changed while the image is kept unchanged. This helps in deploying the same image with only different configurations. The last, and biggest section with 28 MiB is the image location. If an image is smaller than this size, the remaining flash memory can also be used for

configuration or static data, which did not fit in the configuration section.
Note, on the Imote2 platform there are to limits on programming and erase which is NOT taken into account by the codeloader and must be done by the programmer itself: Addresses for writing are only programmable if the alignment is `0x40` (64). Erasing can be performed only in whole 128 kiB blocks. In practice you can only write to adresses which divide by `0x20000` (128 kiB) and the whole next 128 kiB will be erased, even if you only write a few bytes.

Table 2: new memory layout

Memory Location	Size	Description
`0x00000000 - 0x0003FFF7`	256 kiB	bootloader location (fixed)
`0x0003FFF8 - 0x0003FFFB`	4 Byte	Node ID, should be lower than 255 (fixed)
`0x0003FFFC - 0x0003FFFF`	4 Byte	serial number - used for device identification (fixed)
`0x00040000 - 0x003FFFFF`	1.744 MiB	node specific data (user programmable)
`0x00400000 - 0x02000000`	28 MiB	image location and individual data (user programmable)

For the new bootloader in contrary to the original bootloader, we changed the behavior, when a program can be transfered to the Imote2. When the Imote2 starts up, and a USB cable is connected it waits for 3 seconds for a new image. If the PC does not provide a new image within this time, the image on the Imote is executed. While the image is executed, no new image is accepted until reboot.
Only one exception exists to this rule: before an image is executed it is checked, that the image is complete, which is done by comparing a special marker value at the end of the image. If the marker is not present or invalid, the bootloader waits until a new image is provided.

After the successful check of the marker the image is loaded and started. In our case loading means, all static data members (section bss) are initialized. In case of C, the main program starts. In case of C++, all static initializers have to run prior execution of the main program. The behavior is adapted by a modification of the linker script for the Imote2, which now supports running C++ code.

Writing large code for the Imote2 raises the point of failures, which makes it necessary to have good debugging options. For example is simple `printf` statement in the source code is compiled without complains but stops execution immediately, because there is no implementation for this function. If you use the old bootloader, it simply stops and you don't know why. We implemented a better error handler, which shows this error by alternate switch on and off all leds. Additionally, pressing the button causes the error handler to print a stack-trace to standard UART port. We implemented handlers for segmentation fault (the most common bug) and illegal instruction.

The slow upload process of the old bootloader was improved to a programming speed of more than 100kb/s. For most images the overall programming and erasure time was reduced to less than one minute, which makes application testing faster.

3.4 Planned Features

After the successful implementation of the basic features for the code- and bootloader, more whishes arise. We have already shown that application development is improved by a higher programming speed. For the future we want to improve this speed by not flashing the image to the Imote2, but simply copying the data to the dynamic RAM and start the code right away. The gain would be a higher data transfer to the node and no erase of the flash. On the contrary, the data is not written permanent, so any loss of power causes loss of the image, which is acceptable for fast application development.

In future it should be easier to replace the old TinyOS bootloader by our new one. Currently, external hardware is needed for the update via the JTAG connection. The same problem applies to an update of our bootloader, it needs to be flashed via external hardware device.

4 Usage

Programming a sensor platform is a very critical task. Depending on the caused error the node might be unuseable afterwards. Anyway it is considerable bad practice not to give all possible useful information to the user. In cases of an failure the user can determine if the problem is solvable by him. Both codeloader and bootloader indicate each step of the process by certain outputs.

4.1 Status

An image is uploaded by the codeloader, by issuing the command `java -jar codeloader.jar -file image.out`. In this case the image is named `image.out`. The codeloader prints `Waiting for device.`, adding dots until an Imote2 is attached to the USB port and started. The Imote2 shows a blue led on (Linux) or flashing (Windows) which indicates it has received a USB configuration and is waiting for the codeloader for 3 seconds. The codeloader confirms the detection of the Imote by printing `detected`.
The next state is the erase state where codeloader prints ”Erasing” and the Imote2 has all 3 (red, green, and blue) led on. Additionally a fourth led is fading from bright to dark, which indicates the state of erasure.
The upload process is shown in more detail on the PC. It prints the percentage of the already transferred data to the whole image size. The Imote2 indicates the upload by a flashing green led, every time a USB packet is received and a fading external led which corresponds to the percentage on the PC. After a successful upload the external led is switched of, the codeloader prints `Reseting device`, all leds on the Imote2 switched off and the new image is executed.
If on boot the red led is active, this indicates that the bootloader cannot load the image, because it is invalid or corrupted.
If all leds are amber on boot of the Imote2 and stay in this state, there is a serious hardware problem on the Imote2, which is most likely a defect RAM module. Consider this Imote2 to be broken.

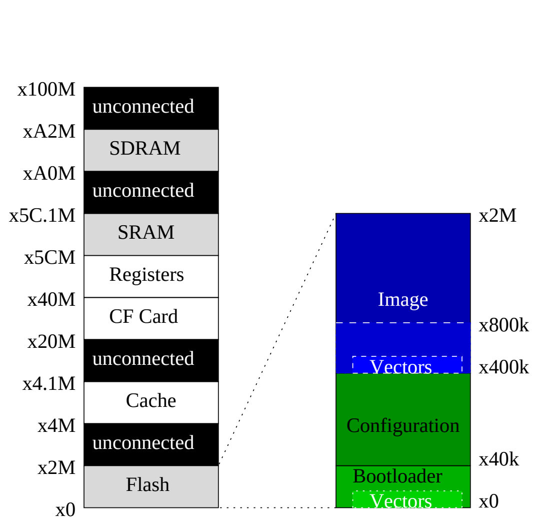
4.2 Command line of the codeloader

The command line for the codeloader is very simple and easy to understand. A new upload is initiated by writing `java -jar codeloader.jar -file image.out`, which will upload the image `image.out` to the Imote2. All command line options are shown below. For more details on the usage, refer to [10].

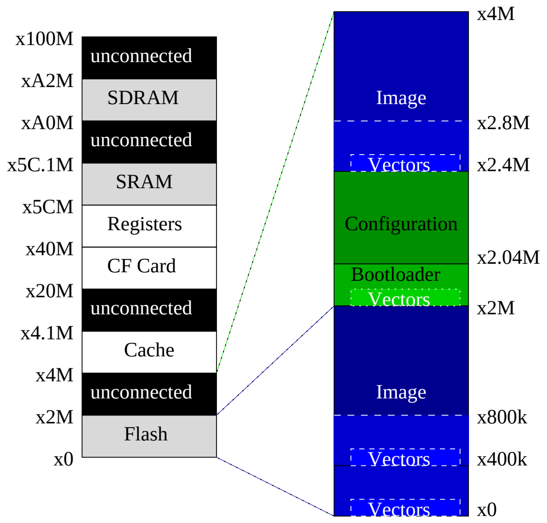
```
usage: codeloader [-connect <connect>] [-connectdisplay] [-dryrun] [-file
       <file:target>] [-help] [-invalidate] [-listen] [-noreboot] [-pid
       <product-id>] [-snr <serialnumber>] [-vid <vendor-id>] [-write
       <write>]
 -connect <connect>      Connect to a daemon specified by its network
                         address
 -connectdisplay         Connect to a daemon specified by the DISPLAY
                         environment variable
 -dryrun                 Reads and Writes Config File(s) and exits then
 -file <file:target>     Adds a file to upload with its optional target
                         address
 -help                   print this help message
 -invalidate             Invalidate default image Location
 -listen                 Run in Listening mode
 -noreboot               Does not reboot the device afterwards
 -pid <product-id>       Product ID of the Imote. Defaults to 0x1337
 -snr <serialnumber>     Serial Number of the Imote. Type '*' for any.
 -vid <vendor-id>        Vendor ID of the Imote. Defaults to 0x042b
 -write <write>          writes a new config file
```

5 Memory

The Imote2 platform has a virtual memory of 4 GByte. This virtual memory is divided into several regions which represent either memory, or hardware functions, or left unused. The memory management unit (MMU) can be used to map regions to others. On PC platforms this feature helps the loader to remap the base address of the program.



(a) Memory layout at execution of bootloader, without MMU



(b) Memory layout at execution of image, using MMU

Fig. 3: Memory layout at bootloader (a) and image (b) stage.

5.1 Physical Memory Layout

In Fig. 3(a), the virtual memory of 4 GByte is shown. The first 32 MB of the memory are used for the static program in the flash memory. The same space is left unconnected after this section. The next section is used for cache control and resides at 64 MB. At 512 MB the virtual region for compact flash, memory or pc card is set. At 1 GB, all registers, controlling the Imote2 are mapped. At 1,472 GB the fast SRAM with 256 kB, which is used as RAM, stack and heap resides. And finally at 2,560 GB, 32 MB of slower SDRAM are present.

5.2 Memory layout without MMU

At execution of the bootloader, the whole memory is left without remapping. On the right side of Fig. 3(a), the layout of the flash during this phase is shown. The bootloader has a maximum size of 256 kB and starts at `0x0` with the vectors mapping to the error handlers (e.g. interrupt handler, illegal instruction or segmentation fault). These vectors are only used during the execution of the bootloader. Next a configuration section of $3\frac{3}{4}$ MB is set and followed at 4 MB by the image. The image itself has a vector section pointing to error handlers inside the image. The rest of the image is just the code generated by the compiler and linker. Since the Imote2 platform searches for the vectors at `0x0`, one possibility is to have the vectors of the bootloader make the remapping for the image. There are good reasons not to do so: first the bootloader is not active anymore, second, the image should act (and be as fast), as if there has never been a bootloader, so why having it handle the interrupts or faults?

5.3 Memory layout using MMU

Figure 3(b) shows the memory layout used, when executing the image. As in Fig. 3(a), the image still resides at 4 MB, additionally the vectors of the image are mapped to `0x0`, so the bootloader and its vectors are invisible at this place. Since the MMU has a granularity of 1 MB, parts of the configuration section is invisible, so we decided to have the whole segment of 4-8 MB mapped to itself, and to `0x0`. Since the memory region after 32 MB is unused, we decided to map the real flash contents to this region, so the image can access its configuration, by accessing the the memory location plus 32 MB.

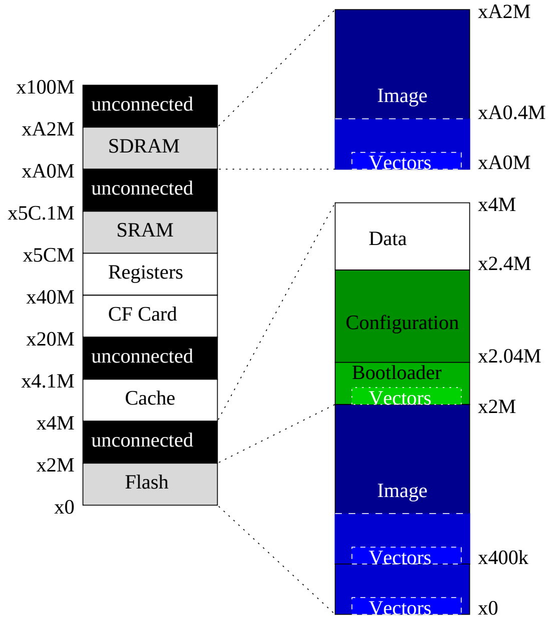


Fig. 4: Memory layout at execution of image in SDRAM

5.4 Memory while executing the image from SDRAM

In Fig. 4, the memory layout for images executed directly from SDRAM is shown. Our goal was to have images directly execute from SDRAM without writing it to the flash memory. Without using the MMU, a new linker script and some changes in the object files have to be made. But by the use of the MMU, there is no need to do so, only the bootloader and the codeloader need to know this and change the MMU accordingly. The image resides in SDRAM at address `0xA0000000` with its vectors. To keep this compatible, as mentioned, the region of the next 4 MB is mapped to `0x0`, which hold the vectors of the image. The full image is then mapped to the region of the flash memory, which is now hidden. And for all images using the configurations, the real flash memory is mapped to the region above 32 MB. This memory layout is easy to use, and no changes have to be made during compilation. Only the upload parameter determines, whether the image is written to flash, or directly executed in RAM.

6 Conclusion

In this report we presented you a new bootloader for the Imote2 platform, which avoids the known problems of the original bootloader. In practice it has already shown its robustness and flexibility. In application development the support for C++ and the support for error backtracking are the outstanding features of this new bootloader. In daily use features of the codeloader, e.g. `listen` and `connect` which adds support for terminal server environments and `snr`, which programs only the specified device become more and more helpful.

The communication protocol for the Imote2 was designed platform independent and simple, which makes it easy to adapt new functions to the codeloader without changing the bootloader. Even if no implementation is available for over-the-air-programming, the current protocol is designed to support this feature as well.

References

[1] Free Software Foundation, Inc.: Grand Unified Bootloader. `http://www.gnu.org/software/grub/` (2010)
[2] Memsic: Imote 2 datasheet. `http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=134%3Aimote2` (2010)
[3] ARM Ltd.: ARM Architecture. `http://www.arm.com/` (2010)

[4] Shahabdeen, J.A.: Boot Loader Architecture. Technical report, University of California (2005)
[5] Red Hat, Inc.: cygwin homepage. `http://cygwin.com/` (2010)
[6] Microsoft: Microsoft Windows. http://www.microsoft.com/Windows/ (2010)
[7] Apple Inc.: Mac OS X. http://www.apple.com/macosx/ (2010)
[8] Oracle Corporation: Java. `http://java.sun.com` (2010)
[9] N.N.: libusb. http://sourceforge.net/projects/libusb/develop (2010)
[10] Krämer, M., Engel, M.: Bootloader for Imote2. `http://vs.cs.uni-kl.de/activities/boot/` (2010)