IN3405 Bachelor thesis

Network-based Bootloader For Distributed Embedded Systems Applications

Chiel de Roest
4036832

Harmjan Treep
4011724

in Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science

at Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
August 14, 2012

Committee:
dr. S.O. Dulman (supervisor, client)
dr. phil. H.-G. Gross (bachelor thesis director)

Preface

This document describes the design and process of building a network-based bootloader for the LPCXpresso boards. It is our bachelor's thesis for the bachelor Computer Science at the TU Delft. Our assignment was executed as part of the "Snowdrop" research group in the Embedded systems department.

We were both interested in embedded programming close to the hardware but struggled to find an bachelor's thesis assignment in that area. We arrived at the research group via Maurice Bos who was already doing a project there.

The "Snowdrop" research group was at the time in the middle of implementing the *proto*SPACE floor, which is a floor where each tile has a RGB LED, a pressure sensor and can talk to its four direct neighbors. A central CAN bus connected to all the tiles was designed in to be able to debug and work with the network. After some meetings with Stefan Dulman we arrived at the assignment of writing support tools for the *proto*SPACE floor. We ended up implementing a CAN bootloader, allowing a programmer to deploy code on the nodes, or a subset of the nodes, in the floor without having to re-flash them.

We learned a lot about embedded software in this project. Both of us want to continue in the embedded software field and this project gave us an insight in what embedded software is about.

We want to thank the Snowdrop group for adopting us in their group and allowing us to work with their resources. Stefan Dulman and Andrei Pruteanu for supervising the project and feedback on our reports. Also Steffan Karger and Agostino Di Figlia for answering a lot of our questions.

Summary

In our Bachelor's project, we joined the Embedded Systems department of the Faculty of Electrical Engineering Mathematics and Computer Science. A select group of the Embedded Systems department, called "Snowdrop" is developing applications concerning large scale adaptive systems. As a playing field for their applications, the group uses the *proto*SPACE floor, consisting of 189 tiles with each containing a microprocessor.

The product to develop is a network-based bootloader that can program multiple microprocessors simultaneously. This would decrease the amount of testing time for developers of the floor, as currently each microprocessor has to be programmed individually.

We have had some setbacks during the development of the bootloader. This was largely due to the debugging time of hardware related bugs, which was not taken into consideration before the start of the project.

During the final weeks, the bootloader software has been tested on a part the *proto*SPACE floor, as the floor itself is also in development. As these tests are not yet finished, the only preliminary result is that 16 successful node flashing was performed by the bootloader software.

We have not found a bootloader as general as this one. It only needs the LPC1769 microprocessor and an external crystal oscillator and it doesn't require any change to the user application. This makes the bootloader a good candidate for other projects that use the LPC1769 microprocessor.

All in all, we are happy that we have chosen this project as it is a level-wise daring one. Our learning experience was way more precious than the time spent on the project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

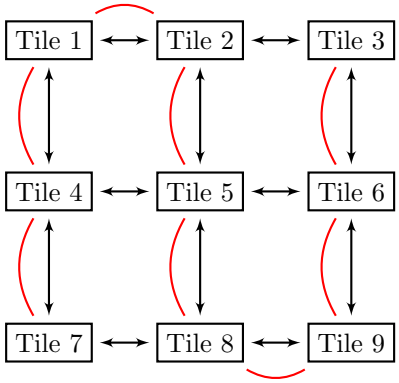


Figure 1.1: Overview of the *proto*SPACE system, the arrows are the UART connections and the red line is the CAN bus.

In this chapter we want to introduce the assignment briefly and discuss the role of our work in the research group. The content will overlap with our project proposal which is included in Appendix B.

1.1 *proto*SPACE

*proto*SPACE is a joint venture between the Hyperbody research group from the Faculty of Architecture and the Snowdrop research group from the Faculty of Electrical Engineering, Mathematics and Computer Science. Hyperbody is interested in interactive architecture and Snowdrop in locally-interacting agents that generate complexity in large-scale systems. These interests come together in the *proto*SPACE floor, seen in Figure 1.2.

The *proto*SPACE floor consists of tiles which each have a pressure sensor to sense if someone is standing on the tile and a RGB LED. Each tile can communicate with its four direct neighbors via UART and a central CAN communication bus is connected to each tile for debugging, as seen in Figure 1.1. This is of interest to the Hyperbody group because it is an interactive architecture, via the pressure sensors and the LEDs the floor can interact with a person or a group of people. The *proto*SPACE floor is of interest to the Snowdrop group because every node only has knowledge about its direct neighbors. This makes it a locally-interacting agent. With a relatively simple program on each node a network can perform a complex task.

The long term goal for the *proto*SPACE project is to have buildings that interact with the user and do that with agents. The advantage of that implementation is that in large and complex systems a central governing agent that makes all decisions is not feasible and puts limits on the size of the system.

1.2 Problem statement

The main assignment we worked on was developing debug tools for developing on the *proto*SPACE floor. Debugging and deploying applications on the floor is really hard.

Deploying applications involves manually flashing all the tiles you are going to test on, so walking with your flashing tool from tile to tile updating the code. This in practice meant that tests were only done on small portions of the floor and that no real application has been deployed on all 189 tiles in the network. Developing an application to flash all or a subset of all tiles with a piece of software will make deploying code on the floor easier.

Another part of our assignment, for which we did not have enough time to implement, was making tools for debugging applications on the floor. The problem here is that debugging deployed applications is hard. The tiles all run a virtual machine such as eLua or

Figure 1.2: The *proto*SPACE 3.0 room.

Proto. Ideally you would want to be able to inspect the internal state of a tile via the CAN bus.

Chapter 2

Orientation

Before starting the implementation we off course did a study on what we needed to do and how we we're going to do it. We discuss our orientation in this chapter, which will overlap with our orientation report in Appendix C.

We have already discussed the goals of the *proto*SPACE project, now we want to look into more technical details of the project. The hardware that will go into the floor will be discussed, we will discuss the microprocessor we are going to have to program for and the tools at our disposal to develop and debug with.

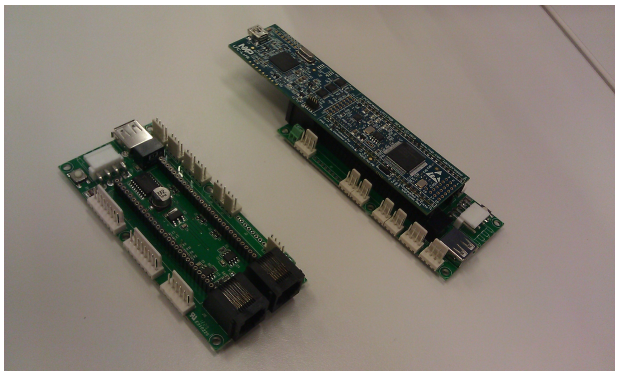


Figure 2.1: The LPCXpresso LPC1769 board and the LPCXpresso shield V3 assembly.

2.1 Hardware

Our code will be deployed on the second generation hardware for the floor. The first generation hardware did not function according to the requirements and had to be redesigned. The design behind the second generation hardware is discussed in [5] and will only be briefly touched upon here.

The hardware for each tile consist of two parts, the LPCXpresso board and the LPCXpresso shield. The assembly is pictured in Figure 2.1. The LPCXpresso board is the blue board on the top and the shield is the green bottom board.

2.1.1 CAN

The bootloader we develop and all other development tools for the *proto*SPACE floor, such as debugging tools, use the CAN bus to be able to communicate with all nodes easily. We will explain how the CAN bus works. The limitations imposed by the CAN bus through the implementation in the floor are discussed in subsection 2.1.3.

A CAN bus is a communication protocol implementing the first 2 layers of the OSI layer model, the physical and the data link layer. In simple terms the CAN standard defines the physical side, such as the wires and the transceiver, and the frame format, i.e. how a CAN frame looks like and how other nodes should respond to a frame. The entire CAN standard is discussed in more detail in [2].

CAN Physical layer

We discuss the physical layer of a CAN network and what constraints the network has to conform to.

A CAN network consists of at least two nodes connected via 2 signals, CAN Low and CAN High. This is a twisted pair signal, which means that the actual signal is the difference in voltage between CAN Low and CAN High. A twisted pair connection has a lot of good electrical properties such as a good resistance to noise and interference. This makes CAN a standard that is also used a lot in the automotive industry.

A CAN bus has a line topology, as seen in Figure 2.2. This means that the wires connect through the nodes in a line, with a clear start and end node. As seen in Figure 1.1 is this done in the *proto*SPACE

Table 2.1: CAN arbitration table.

	Dominant	**Recessive**
Dominant	Dominant	Dominant
Recessive	Dominant	Recessive

floor by running the CAN bus up and down the floor.

Every node has a CAN transceiver which does the translation from the electrical signals at the physical layer to just a serial bit format, as seen in Figure 2.2. The transceiver just provides an interface from the CAN High and CAN Low signal to the microprocessor. The microprocessor has the data link part of the CAN standard built in.

The start and end node of the CAN bus should be terminated, which means that CAN Low and CAN High should be connected with an 120 Ohm resistor. This is to prevent reflections on the bus.

The speed of the bus is not fixed in the CAN standard. The speed of the bus is limited by the length of the cables, the number of nodes and the number of connectors. All these factors add parasitic capacitance to the bus, this is further described in [2].

A node at the physical level puts a bit on the bus and other nodes receive that bit. You have the dominant bit, the 0, and the recessive bit, the 1. In Table 2.1 can you see the arbitration table. Nodes that are trying to send a recessive bit but read back a dominant bit know that they have lost arbitration and back off from the bus. This means that CAN has non-destructive arbitration, if two or more nodes on the bus collide a node wins and continues so no time is lost because of arbitration.

CAN Data link layer

The CAN standard also defines two types of data frames that can be used. We will only discuss the standard frame. The extended frame that was later added to the standard is discussed in [3].

Every CAN frame consist of an 11 bit ID and between 0 and 8 bytes of data. The CAN frame with all fields is drawn in Figure 2.3.

All frames are broad-casted onto the entire bus, so every node receives every frame. All frames need to be acknowledged to be considered sent by a node, and every node acknowledges all frames. So if on a bus of 10 nodes 9 receive the frame correctly, then the sender will only see that the frame got acknowledged and consider the frame sent.

Every message has a CRC field, as seen in Figure 2.3. This means that if a message is received by a node, it is already checked that that message was received correctly.

In Figure 2.3 it is shown that the identifier of a CAN message is the first part sent in a frame (after the start of frame bit). This is done so that the identifier of the frame also determines the priority of the message. Messages with low identifiers have more 0 bits early in the message and the 0 bit is dominant on the CAN bus. If a node sees that it has lost arbitration in the identifier field it tries again after the frame has been sent. If however a node sees that it has lost arbitration outside of the identifier field it sees this as an bus error. After a small amount of bus errors the CAN node shuts itself down to prevent itself from breaking the bus. This can happen for example if the nodes operate at different speeds or the length of the CAN cable is too long for the speed the bus is used on.

2.1.2 LPCXpresso

LPCXpresso is a low-cost development board. There are different LPCXpressos for different processors. We use the LPC1769 as the main processor, the LPC1769 as a micro-processor will be discussed in section 2.2.

The LPCXpresso is designed as a complete development toolchain. For the hardware this means that half of the board is populated with a flash tool. Normally, to flash a processor a JTAG or SWD flash device is needed. On the LPCXpresso this is integrated. Together with the LPCXpresso IDE, further discussed in subsection 2.3.2, you can prototype code for the LPC1769 very rapidly. The LPCXpresso is cheap because NXP, the manufacturer of the LPC1769, sponsors the platform to allow people to get experience with their processors and subsequently design products with them.

The LPCXpresso for the LPC1769 comes with a debug LED and I^2C EEPROM flash memory, which can be used for permanent storage.

2.1.3 Shield

The LPCXpresso shield is custom developed for the *proto*SPACE floor, version 3 is shown in Figure 2.1. It adds all hardware not directly available on the LPCXpresso board, such as the CAN transceiver and the power regulators.

The CAN bus goes through the Ethernet connectors on the shield. The Ethernet cables were chosen

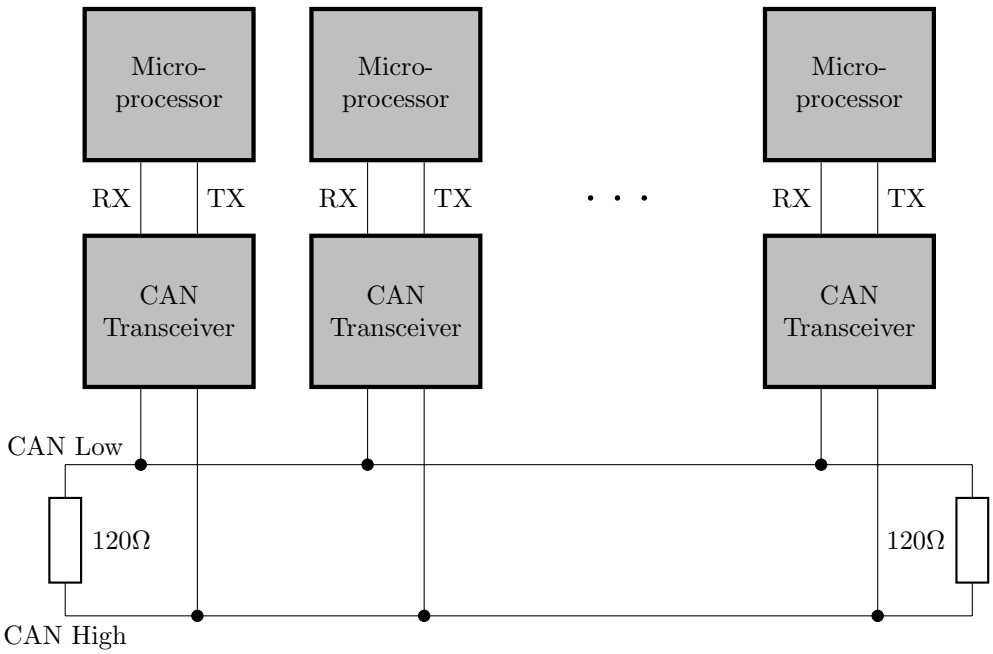


Figure 2.2: A CAN network.

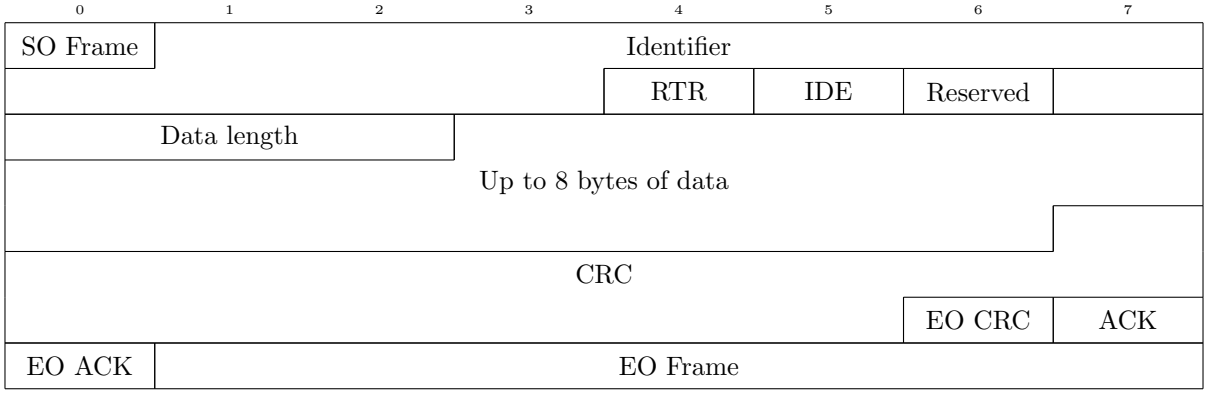


Figure 2.3: The CAN standard data frame format.

because Ethernet also uses a twisted pair in the cable. Ethernet cables are however more widespread than CAN cables and are cheaper. There were special termination Ethernet stumps made to serve as CAN bus terminators.

The other connectors on the shield are for the UART connections to the neighbor nodes, the wireless transceiver and the pressure sensor. All of these peripherals are described in Steffan Karger's paper [5].

2.2 Micro-processor

The part of the hardware we were going to do the actual work on is the micro-processor. The LPC1769 was used in the design, in this section we will explain how the LPC1769 works in general. The complete working of the LPC1769 is described in the *LPC17xx User manual* [7].

2.2.1 Peripherals

A micro-processor is not fast compared to a desktop processor. The LPC1769 can run at top speed at 120MHz. To be able to execute complex tasks and communicate with other chips efficiently, manufacturers add peripherals to a micro-processor. For example a CAN peripheral. The CAN peripheral on the LPC1769 has the CAN protocol build in. A program can just tell the peripheral to send a message over the bus, and the peripheral will take care of the timing, arbitration and all other details. Peripherals are implemented in hardware so the sending of the CAN message can go in parallel with the program code.

In the LPC1769 the program communicates with the peripherals via setting values at specific places in memory, which are called registers. The program can also change processor settings such as the clock source to use or what pins of the processor are used for what function.

A micro-processor can use interrupts. Interrupts are small pieces of code that are called at an event. Examples of such events are the passing of some time or the receiving of a CAN message. Interrupts need to be enabled by setting a register. Also, a function pointer to the interrupt handler needs to be put in a specific place in memory, in the interrupt vector table.

2.2.2 Memory

The LPC1769 has a specific layout of its memory which we will discuss in this section. The memory layout defines where in the address space of the LPC1769 certain items are located, such as flash, RAM and peripherals. An abbreviated overview of the LPC1769 memory is visible in Figure 2.4.

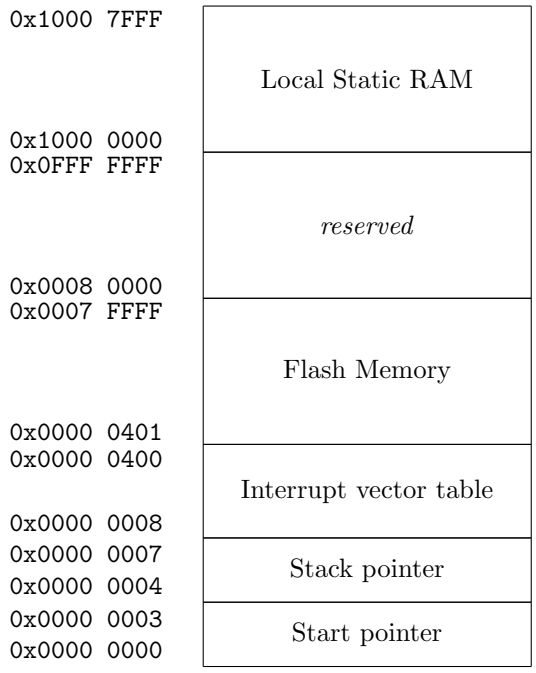


Figure 2.4: Relevant parts of the LPC1769 memory.

At the very start of the flash are the start and stack pointer. These two pointers determine how the processor is initialized when switched to user code. Directly above these variables is the interrupt vector table with pointers to all the interrupt handlers.

In the flash memory is the application code. The bootloader and the bootloaded program both have to be in this region.

In the reserved space are all the peripherals and a lot of unused space. RAM is at the top of the address space.

As a programmer you can put whatever you want in the flash memory. From `0x0000 0000` to `0x0007 FFFF` you can put whatever you want during flashing. In the area above you can only set things during runtime of the processor (only in registers and in RAM) and it will be gone when the processor reboots.

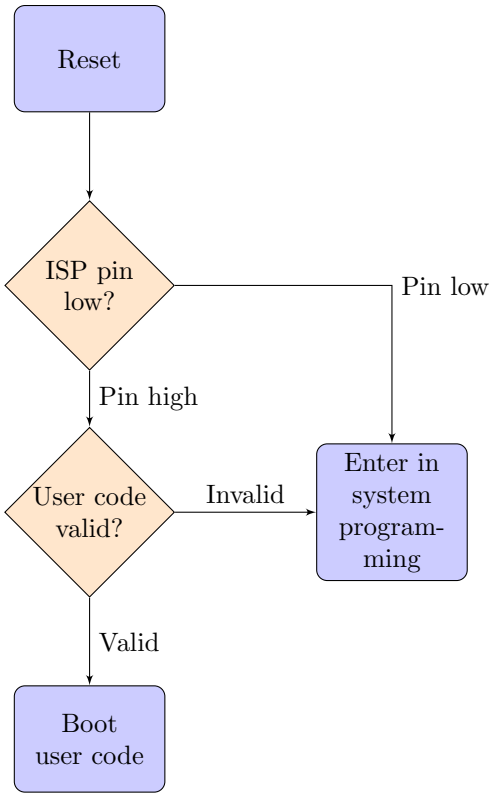


Figure 2.5: Relevant parts of processor boot sequence.

2.2.3 Boot sequence

The LPC1769 has a specific boot sequence, which is shown in Figure 2.5. There is already an UART bootloader on the LPC1769. This is called In System Programming, or ISP. To enter this mode when booting, a specific pin has to be pulled to ground.

The LPC1769 also verifies whether the code in the processor is valid. This is verified by looking at a certain word in the interrupt vector table, which should be the hash of the previous entries in the interrupt vector table. This way, the LPC1769 determines if there is a valid program in the flash. You can have a corrupt program when flashing went wrong or when the processor was just shipped from the factory.

2.3 Development tools

In this section we will discuss the tools available to us for completion of the project.

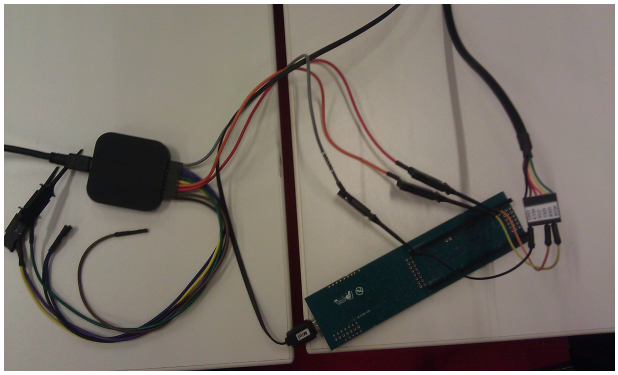


Figure 2.6: A debug session with the logic analyzer.

2.3.1 Workplace

We worked in the lab on the 9th floor of the Faculty of EEMCS. The lab was intended for master students of the Embedded Systems group. This had the advantage that we were sitting in the same room as the master students that also worked on the *proto*SPACE floor.

The *proto*SPACE room with the floor itself is actively being used in architecture.

2.3.2 LPCXpresso IDE

The LPCXpresso IDE is an Eclipse-based development environment. It works well with the LPCXpresso boards to allow fast prototyping. There is however a limit on the free version for code size when downloading and debugging via the LPCXpresso IDE. This implicates that you cannot use the upper half of flash memory when using the LPCXpresso IDE. A solution was found for this problem which is bootloading the code via the onboard UART bootloader and an USB to UART interface cable. You can't however start a debugging session over the UART cable.

2.3.3 Logic analyzer

We had a logic analyzer available to us to debug our system with. With a logic analyzer a developer can visually inspect a logical as a function of time. We had a Salea Logic. That logic analyzer can also debug protocols, so it is also a good way to see what is really happening on the CAN bus. The analyzer was used for both the UART and CAN bus, as can be seen in Figure 2.6.

Chapter 3

Design

In this chapter we discuss the design of the project. After the orientation phase, described in chapter 2, we started designing the bootloader. We started with gathering requirements to describe the functionality of the bootloader and the constraints on the bootloader. Afterwards we generated concepts and compared them to each other with their conformance to the requirements.

3.1 Terminology

In the course of this report some jargon is used. To avoid confusion over the meaning of e.g. names and abbreviations, the following list with descriptions is made.

EEPROM Electrically Erasable Programmable Read-Only Memory.

Flashing Writing data to the flash memory of a node.

Programming Flashing a node with an application.

Bootloader Software to load applications after startup of a node.

I^2C Computer bus to drive the EEPROM chip.

IDE Integrated Development Environment.

Node A LPCXpresso board with a shield, which is part of the *proto*SPACE network.

Programmer The node that connects the host to the CAN bus.

Host The entity on the PC side.

User application The applications that are programmed on the nodes.

XTAL Crystal oscillator.

IRC LPC1769 internal clock.

3.2 Requirements

Before the start and during the first week of the project a couple of meetings took place with the researchers working on the *proto*SPACE floor. During these meetings the researchers described what problem they wanted solved and what hardware was already there. Furthermore, we investigated the environment in which the CAN bootloader has to operate, this is explained in chapter 2. Out of these meetings a list of requirements was composed.

3.2.1 Requirements

We identified the following requirements.

1. The bootloader must be able to bootload a program from a PC onto a node;
2. The bootloader must be able to bootload multiple nodes at the same time;
3. The bootloader should be able to only program a select subset of all the nodes in the network;
4. The bootloader must be able to start a bootloaded program;
5. The bootloader must work on the hardware that goes into the floor, which is the LPC1769 on the LPCXpresso in the in-house developed LPCXpresso shield v3;
6. The bootloader should be able to program the entire floor, which are 189 nodes;

7. The bootloader should be able to program the entire floor in 5 minutes with a binary with the eLua VM in it and a simple eLua program;
8. The bootloader should be robust;
9. The bootloader should put as little constraints on the user program as possible;
10. The bootloader should be kept as generic as possible.

We want to elaborate on the last three requirements.

With robust we meant that you should not be able to break the bootloader. You cannot easily re-flash the entire floor if you break the bootloader, which is a lot of work. So the bootloader needs to deal with unexpected and wrong input without breaking itself. Breaking the user application is more acceptable because you can still re-flash the user application if the bootloader still works.

What we meant with putting as little constraints as possible on the bootloaded program is that we found implementations that required the bootloaded program to be compiled differently of not use some functionality of the processor. We want to build a bootloader that can bootload any program for the floor. This way a developer for the floor can try some things out on his desk and then move to the floor without having to think about the differences. You don't want a program to work differently on the floor because of the bootloader implementation.

The requirement that the bootloader should be kept as generic as possible came from us. We want to try to develop a generic LPC1769 bootloader that we can open source and which can be useful to more people than just us. Off-course we should not sacrifice functionality for generality.

3.3 Booting the bootloader

The goal of the bootloader is to be able to reprogram the nodes whenever you want to. But the user application can also be running, and you do not want to influence that application. We identified 2 concepts to be able to start the bootloader.

3.3.1 Separate bootloader

In this concept the bootloader is completely separate from the user application. This way the user application cannot break the bootloader when something goes wrong. But it also means that the bootloader cannot be started halfway through the user application without changes in the user application.

As later will be discussed in subsection 5.8.2, the flash memory of the LPC1769 is divided into several sectors of either 4 or 32 kilobytes. The bootloader has to reserve a sector for itself in this concept, which is bad for transparency because the user application could use that sector and since the bootloader uses less than 4kB does this mean flash space of the processor gets wasted.

3.3.2 Linking bootloader in user application

This concept links the bootloader code inside the user application. When the node resets, the user application is called and the user application subsequently has to call the bootloader software. When the bootloader software is finished the user application can run its actual code. This concept scores bad on the little constraints as possible requirement. The bootloader has to be called from the user application, so the user application has to be changed before bootloading. On the robust requirements this concept also scores bad, because if you accidentally flash the wrong binary or if something goes wrong with flashing the bootloader is broken and you'd have to re-flash the floor.

Also is this really complex. Because the bootloader is among the code to be flashed, the sector the bootloader is in will get blanked and then flashed again at some point of the flashing process. To solve this you would have to specially link the bootloader for operating from RAM, then copy the bootloader code to RAM and then start flashing.

3.3.3 Chosen concept

We choose the separate bootloader concept because this concept was more robust and easier to implement. The lost space because of the reserved sector is not as important as the robustness of the system.

3.4 Bootloader location

The most influential design decision is the location of the bootloader code in the flash memory of the LPC1769. It will influence the transparency of the bootloader software to the user application. Two con-

cepts are considered: placing the bootloader code at the top and at the bottom of the flash memory.

3.4.1 Top of flash

In this approach the bootloader code is placed at a high address in the flash memory, as graphically depicted in Figure 3.1. To put the bootloader code at the top of the flash, the bootloader code start at the start of the highest sector. Unfortunately, the highest sector is of size 32kB, meaning that the much smaller bootloader in this fashion takes all 32kB. As there is plenty of flash memory left (512kB − 32kB), this is not a big issue. The eLua VM with a eLua program is approximately 180kB large, so there is still enough space left for bigger and more complex programs.

Placing the bootloader at the top of the flash memory has its problems, e.g. it asks for a different way of starting up a node. When reprogramming occurs, the first sector can be overwritten. As seen in subsection 2.2.2 the start, stack and interrupt vector table reside at the bottom of the flash. So if the user application resides here all interrupt service routines pointers are overwritten with the handlers of the user application. This also means that the user application now gets booted instead of the bootloader, which we do not want. Since the bootloader software should be executed first, this creates a problem.

This problem can be solved by copying both the reset handler and the stack pointer of the bootloader. When placing these values, and not the user application's values at the reset handler's and stack pointer's locations, the node will after reset start executing the bootloader software. Then, the bootloader software can subsequently call the user application.

This however introduces a new problem: how does the bootloader after the node resets know where in the flash memory the user application starts? This is solved by saving two pointers needed to start the user application: a pointer to the user application's reset handler and the user application's stack pointer. Both values are received during the transmission of the user application to the node. All that is needed is save the values somewhere where they are not overwritten. As the bootloader knows the storage locations of these values, the bootloader can retrieve them as such.

Another problem that might occur is that the user application is of such a large size or linked in such a way that it wants to overwrite the bootloader code. This would brick the node as the node still expects bootloader code it wants to execute after the node is reset. This problem is fixed by only programming a user application when it does not want to overwrite the bootloader code. To deal with this problem, positioning the bootloader code at the top of the flash memory is common practice.

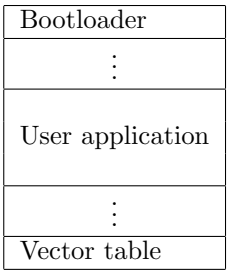


Figure 3.1: LPC1769 flash memory showing locations of bootloader and user application code when placing the bootloader at the top of flash memory. Dots indicate possible unused memory.

An advantage of this approach is that the bootloader is totally transparant to the user application, as required by requirement 9. Now, any user application that does not violate the space of the bootloader code can be programmed onto the node without modification.

3.4.2 Bottom of flash

In this approach the bootloader code is placed at a low address in the flash memory, as depicted in Figure 3.2. This implicates that the linker script of the user program has to be changed to not use the piece of flash memory that the bootloader resides in. As the linker script has to be changed for the user application to be able to be used alongside the bootloader, the bootloader is not transparent to the user application. As transparency is described by requirement 9, this method is not chosen to be used.

3.4.3 Chosen concept

As the both the bottom of flash and the linking in of the bootloader methods do not meet the transparency requirement 9, these methods are not chosen. For the top of flash method the boootloader code only has to be linked to the top of the flash once, which does not interfere with requirement 9. The top of flash method does meet this requirement and is hence the chosen concept.

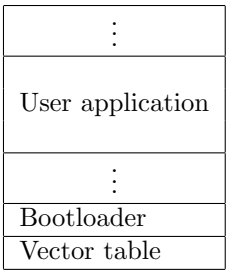


Figure 3.2: LPC1769 flash memory showing locations of bootloader and user application code when placing the bootloader at the bottom of flash memory. Dots indicate possible unused memory.

3.5 Interrupts

We had to decide if we were going to use interrupts or implement everything in a polling fashion.

3.5.1 Use interrupts

The main advantage of interrupts is that you can use the standard NXP library.

A disadvantage is that it is more complex because in the separate bootloader concept you have to replace the interrupt table with stubs that call the user application interrupts or move the interrupt table (you can specify an offset in a register).

3.5.2 Polling

Using polling to implement all functionality means we keep checking in a loop for an event to take place. The disadvantage of this is that you cannot do things in parallel. In other embedded systems applications that would be a problem but it isn't for the bootloader.

Another disadvantage is that we would need to implement the drivers ourselves. We could still use the provided NXP library as an example though.

3.5.3 Chosen concept

We choose to implement the drivers ourself in a polling fashion. This way we did not have the complexity of trying to rewire the interrupts.

3.6 Modules

We split our software in modules to organize it and decouple our system. The modules we designed for are shown in Figure 3.3. The lowest modules are drivers that directly set registers in the LPC1769.

Every module has an *init()* and *deinit()* function. The *init* function is to setup the module, by initializing needed peripherals and variables. The *deinit* function is to tear down the module, by deinitializing peripherals and variables. We need to deinitialize the peripherals to make sure the processor is in the reset state when booting the user program. If, for example, the clock or the CAN peripheral is already configured the user program may behave wrongly. Every module has an initialize function so the implementation behind the module does not matter. For example, whether you solve storage with I^2C (which needs to initialize peripherals) or with storing them in the image (which does not need initialized peripherals) should not matter.

This decoupling is like the *facade pattern* from object oriented languages. The storage module gives an easy interface to the EEPROM chip via the I^2C module. This means that the main function does not need to worry where and how the pointers are stored. In the development process we actually redesigned and rewrote the storage module to store the pointers in the processor flash instead of the external EEPROM. We only had to rewrite the storage module and throw away the I^2C module to make that change.

3.7 Computer network communication

We need to be able to program the network via a computer. We looked at a few concepts for this problem.

3.7.1 USB to CAN tool

We looked at a few USB to CAN tools such as PCAN from PEAK systems. Most of these tools have an API to program them on the computer. All of these tools are very expensive though, the PCAN costing about 200 euros.

We still needed to develop programs and tools that worked with the bought-in parts. We also set a generality requirement: we want to develop a general bootloader that works on LPC1769, a requirement which this solution scores bad on because you need to buy a USB to CAN tool.

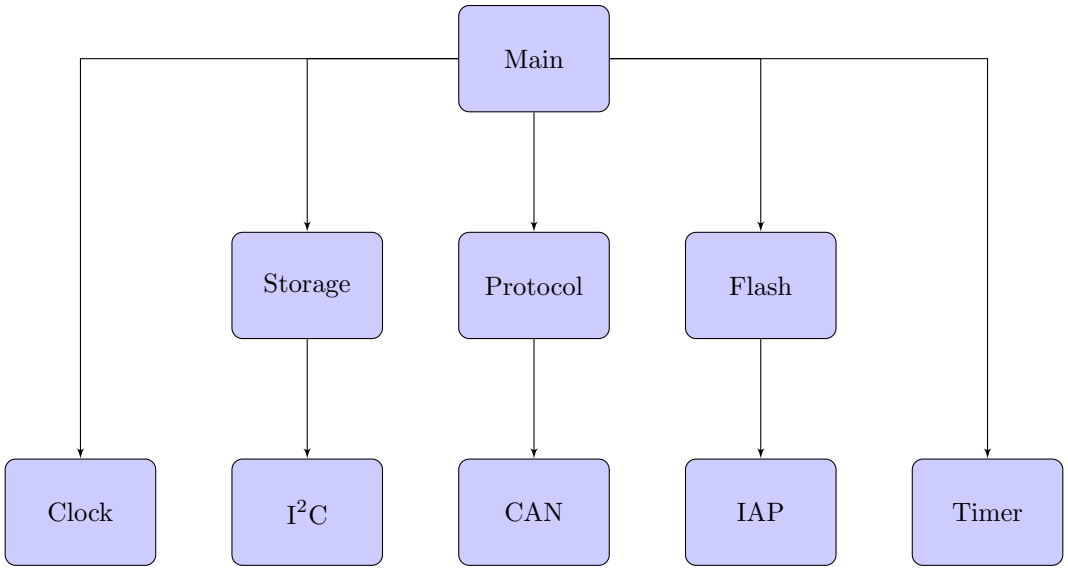


Figure 3.3: The modules we designed and their dependencies.

3.7.2 Link binary in Programmer node

This concept uses a special node, which we will call the *programmer* from now on, which connects the host via a UART bridge to the CAN bus. The topology of such a network is graphically depicted in Figure 3.4.

This concept was actually implemented while developing the programmer side of the protocol. The idea is to compile the binary of the user application into the programmer node. With the *ld* utility we can link in a binary file in the programmer program that we then send over the CAN bus.

The flashing of the programmer binary can be done via the TTL-232R-3V3 cable from FTDI. This cable is an USB to UART interface. The *lpc21isp* tool works with this cable and can flash the full size of the LPC1769. This cable was already used by the developers for the floor to circumvent the 128kB memory size limit from LPCXpresso.

A disadvantage of this technique is that there is no real communication back to the host computer from the programmer node. So you cannot get a status back to the host if the tool failed.

Developing an application with which developers can easily flash programs is possible but the actual flashing will take longer because the *ld* utility also has to run.

3.7.3 UART communication with Programmer node

The communication can also be implemented over a new UART protocol, connecting the host with the programmer. Both the host and the programmer can talk by this protocol.

The advantage of this method is that it is very extensible. If in the future more actions need to be added, just the protocol needs to be adapted. The drawback of this method is that designing and implementing such a protocol is much work, much more compared to the first method.

3.7.4 Chosen concept

We choose the last concept, that implements a programmer node that communicates with the host via UART. The main reason is because of its extensibility.

Moreover, the client noted that he wants to have debugging tools for the whole floor. The communication to facilitate such tools can very well make use of an adapted version of the protocol. This would not have been possible with the first method.

Note that the aforementioned extensibility is not a requirement for this project. Going for the first method would make it much easier for this project and meet the requirements. However, it would make it much harder for the client to create new tools.

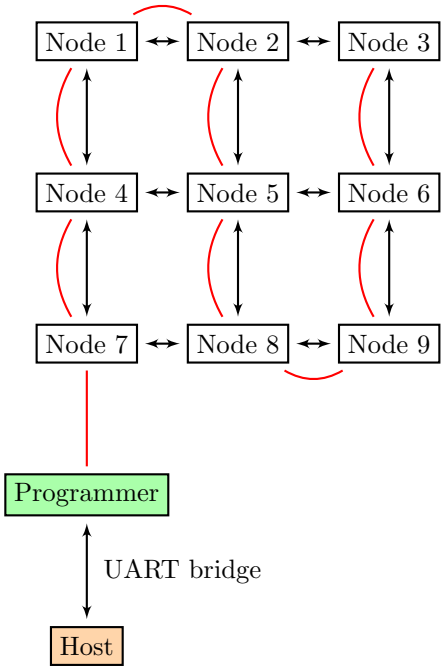


Figure 3.4: Network topology of the CAN bus (red lines) and UART connections (black arrows). The programmer (green) connects the host (orange) via a UART bridge to the CAN bus.

3.8 User interface

Requirement 1 indicates that a host application has to be supplied for users to program nodes. Hence, a command line tool acts as the front end to the user to perform actions on the CAN bus. As the user group of the programming tool will consist only of students and employees of the Embedded Systems department, there is no need for a graphical user interface. This was also underlined by the client.

Requirement 3 indicates that a user should be able to select certain nodes to program. To facilitate such functionality, one method is to supply a list of node configurations to the command line tool. The command line tool can garnish the necessary information out of the list to select certain nodes. The advantage of this approach is that the list only has to be supplied once. A disadvantage is that the data in the list may be outdated, e.g. some nodes in the list may not be on the CAN bus anymore but are still in the list. Depending on the implementation of the CAN protocol this can lead to faulty behavior. Another disadvantage is that managing a file for a network of many nodes can become difficult.

Another method is to scan all the nodes on the CAN bus, i.e. all nodes that are connected to the CAN bus must send their IDs on the CAN bus. The command line tool will query the user to either select or not to select every node that has send their ID. The advantage of this method is that the information is not outdated, unless a node is removed before it is selected by the user. The disadvantage is that when a user wants to program the nodes every node has to be manually selected. This is not efficient and is not doable in large networks.

The decision was made to go for the first option. To cope with outdated information newly found nodes are appended to the list. Moreover, nodes that are no longer on the CAN bus are disabled for programming in the list. The combination of this method these solution creates a command line tool that is simple to operate, even for large networks.

Chapter 4

Software development process

In this chapter the software development process is discussed. It discusses the software development method used and how it gives shape to this project. It also discusses how source code is managed, documented and tested. Finally, the feedback of the Software Improvement Group (SIG) on the software is discussed.

4.1 SCRUM

Software development process method SCRUM is used during the course of this project.

For meetings, daily scrum meetings are used to discuss the progress of the last day and what is to be done during the day. Once every week, a meeting is held with the Snowdrop group, which is the *sprint planning meeting*. Here, the development team and the clients sit together and discuss the progress of the software development and a planning is made for the following week. These meetings are also useful as it gives information of other members of the Snowdrop group. For example, one member wanted to use the bootloader and could make a planning for himself by knowing when the bootloader would roughly be functional.

As there are only two persons working on the project not all of SCRUM's tools are used. For example, the role of *scrum master* is never assigned as it becomes more or less redundant. Instead, both developers are just part of the *development team*. Stefan Dulman (the main client) and Andrei Pruteanu serve as *product owners*.

A detailed backlog is not used. As there is a daily talk between the two developers the necessity is not there to use an actual backlog. However, a list with things to do is used. This list only serves the purpose as a reminder for and prioritizing what needs to be done. It does not, in contrast to a backlog, serve as a real planning document.

Though it may not look as a strict use of SCRUM, the techniques that are being used serve their purpose well. In the end, no changes to these techniques were needed.

4.2 Version control

Source code management system *Git* is used to manage all the written code. Git was chosen as it is much more efficient and can handle branches better than other source code management systems such as SVN. Also, Git is, in contrary to many other systems, fully distributed. This means that all information may be stored on multiple locations, not one central one. As such, Dropbox is also used as a git repository, which served as a back-up during the project.

The necessity of good branching performances is needed as the project contains a numerous subsystems. Source code management works a lot better when using branches for each individual subsystem.

Proper structuring the Git repository saves a lot of time during the software development process. Therefore, the Git repository is structured in a manner that is suggested by Vincent Driessen's *A successful Git branching model* [4].

The project's three subsystems *Bootloader*, *Programmer* and *Host* have their own branch in which patches can be committed for only a specific subsystem. As part of the source code is shared among the subsystems a *library* branch is used. Whenever needed, a subsystem can merge this branch to use new functionality. A separate branch *reports* is used to store LaTeX files for the writing of the reports.

At the point that testing is to be conducted, an *octo merge* is performed. By doing this, the latest patches of all subsystems and the library are merged into one branch, the *develop* branch. Now, the develop branch

has includes all the latest patches. This makes it easier to switch from inspecting the bootloader code to the programmer code, as switching between branches is no longer needed. An octo merge is graphically depicted in Figure 4.1, which displays the `gitg` application which is a graphical front-end for Git.

4.3 Documentation

The code of embedded systems applications is generally low-level, which makes the code relatively hard to comprehend. Also, embedded systems code generally consist of register reading and setting code, which makes it even harder to comprehend. As well documented source code improves its readability and understandability, a large amount of time is spent to properly comment the code inside the source code files. These comments both explain what happens when a function is called and why it is called.

Besides the documentation in the source code files, another often used manner of source code documentation is use: *Doxygen*. Doxygen is a documentation generator that is able to generates a large set of both textual as graphical documentation formats of the source code. In this project, Doxygen is used to create several diagrams and a set of HTML pages that denotes the API of all subsystems.

4.4 Testing

Automated testing of embedded systems applications can be practically not doable, also for this application. For example, automated testing of hardware specific bus drivers. If one wants to test such a driver, the easiest way would be to use a simulator to mimic bus traffic. As this would take such a large amount of time and effort to set up it is not used, as the effort to make them outweighs the benefits of automated tests. Instead, several non-automated testing approach are used, such as visual inspection using a logic analyzer as discussed in subsection 2.3.3. As this testing approach is not automated and it does not guarantee a well functioning product, it gives the developer some much wanted feedback.

Another way of testing is the use of LPCXpresso's on-board LED. By programming a node with an application that turns on the LED, visually inspecting the LED gives feedback on the result of the programming action. The LPCXpresso also facilitates the

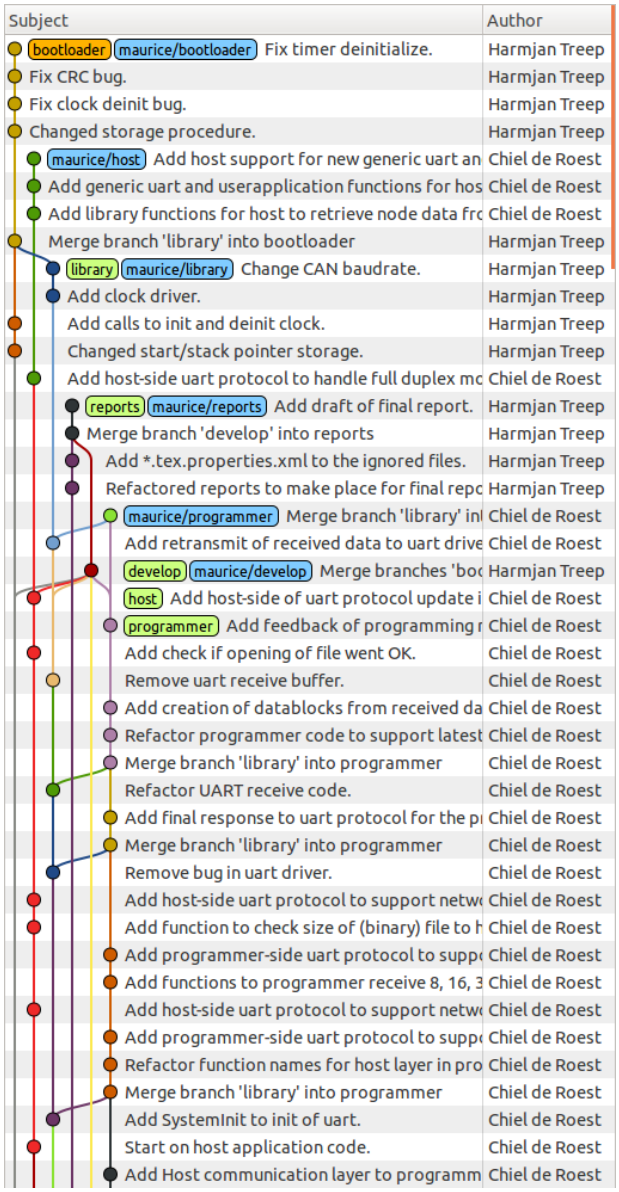


Figure 4.1: A snapshot during development of the Git commit tree. Multiple local and remote branches can be seen. Also, an octo merge can be seen, which merges the bootloader, host, library and programmer branch into the develop branch.

reading of the microcontroller's memory and peripherals.

SIG also observed that there was no test code in the first code upload. SIG suggested to add at least some test code so that the client can make changes to the code and can automatically test the original code. As mentioned before, the client does not need test code as such, so it is also not included in the final code upload.

4.5 SIG

The first code upload to SIG was rated very well, receiving four out of five points. The reason the code did not get a perfect rating was due to the size and complexity of some functions.

On some degree this analysis is correct. For example, there was a function that was unnecessarily on the larger and could be very well split up. On this part, the advice was taken and code was adjusted accordingly.

However, a larger function may be advantageous. For example, driver functions generally have larger functions as usually a large quantity of settings need to be performed. Usually these settings are noted in the manufacturer's manual which is also read by other developers. It is easier for other developers to see a one-to-one correspondence of the manual's setting sequence and the code's. Breaking up the code does not improve this.

Also, performing these settings are usually bound to a certain sequence, breaking up the code in small functions may lead other developers in failing this sequence. As the examples above, having a somewhat larger function and better readability is chosen above function length.

As mentioned in section 4.4 SIG pointed out to add test code. As discussed, this advice is not followed.

Chapter 5

Implementation

5.1 Startup

The purpose of this network-based bootloader is to listen for a while to the network for new applications and programs these or otherwise starts existing applications. The workings of this bootloader is graphically depicted in Figure 5.1.

After resetting, the bootloader is the first application that is executed. The bootloader software first initializes all components, such as network drivers, clock settings and timers.

Secondly, the bootloader software checks whether there is already a user application stored on the flash memory. If yes, the bootloader will forever wait for new applications to be available. If not, the bootloader will wait for a fixed amount of time for new applications to be available on the network.

When a user application is available, the bootloader programs this application into the node's flash memory. Finally, the bootloader components are deinitialized to make sure the user application is executed with the 'standard' settings. After deinitilizing, the bootloader code starts the user application.

5.2 CAN protocol

A CAN protocol is implemented to transmit data from the programmer to the nodes on the CAN bus and backwards. The current protocol supports the scanning, the programming and the resetting of nodes. The CAN protocol implemented to program nodes is graphically depicted in Figure 5.2.

Firstly, the programmer puts a `0x100` message on the CAN bus which makes just rebooted nodes go into bootloading mode. The programmer does this for two seconds, a time long enough for every node to receive it. Nodes in bootloading mode will listen to subsequent messages from the programmer, nodes not in bootloading mode will eventually load the existing user application.

Secondly, the programmer broadcasts a `0x101` message asking all nodes in bootload mode to send their node ID. The nodes respond by sending a `0x102` message with their node ID as payload.

Thirdly, the programmer selects the nodes to program individually by sending one `0x103` message per node with payload the ID of the node. Selected nodes will listen to data sent by the programmer containing the new user application.

Fourthly, the programmer starts sending the new user application to be programmed on the previously selected nodes. The programmer receives a user application in *blocks* of 4kB for easier flashing, will be discussed in subsection 5.8.3.

A block is described by a *block sector number* and its 4kB *block data*. The programmer first broadcasts the block sector number in a `0x104` message, so the nodes know in which sector to place the data to be received.

The programmer subsequently sends the block data in `0x105` messages, eight bytes of data per message. This is the maximum amount of data bytes per CAN frame, as discussed in subsection 2.1.1.

When all the block's data is sent, the programmer sends a `0x106` message with the computed hash of the block as payload. Every node also computes the hash of the received block, compares it and sends a message with the result back to the programmer. This is done by sending a `0x107` message with payload the ID of the node, the result of the hash and the result of the flashing process. The hashing model used is discussed in subsection 5.4.2, flashing is discussed in subsection 5.8.2.

When all blocks are sent, the programmer sends a `0x108` message which will tell all nodes to reset. This will make the nodes run the just programmed user

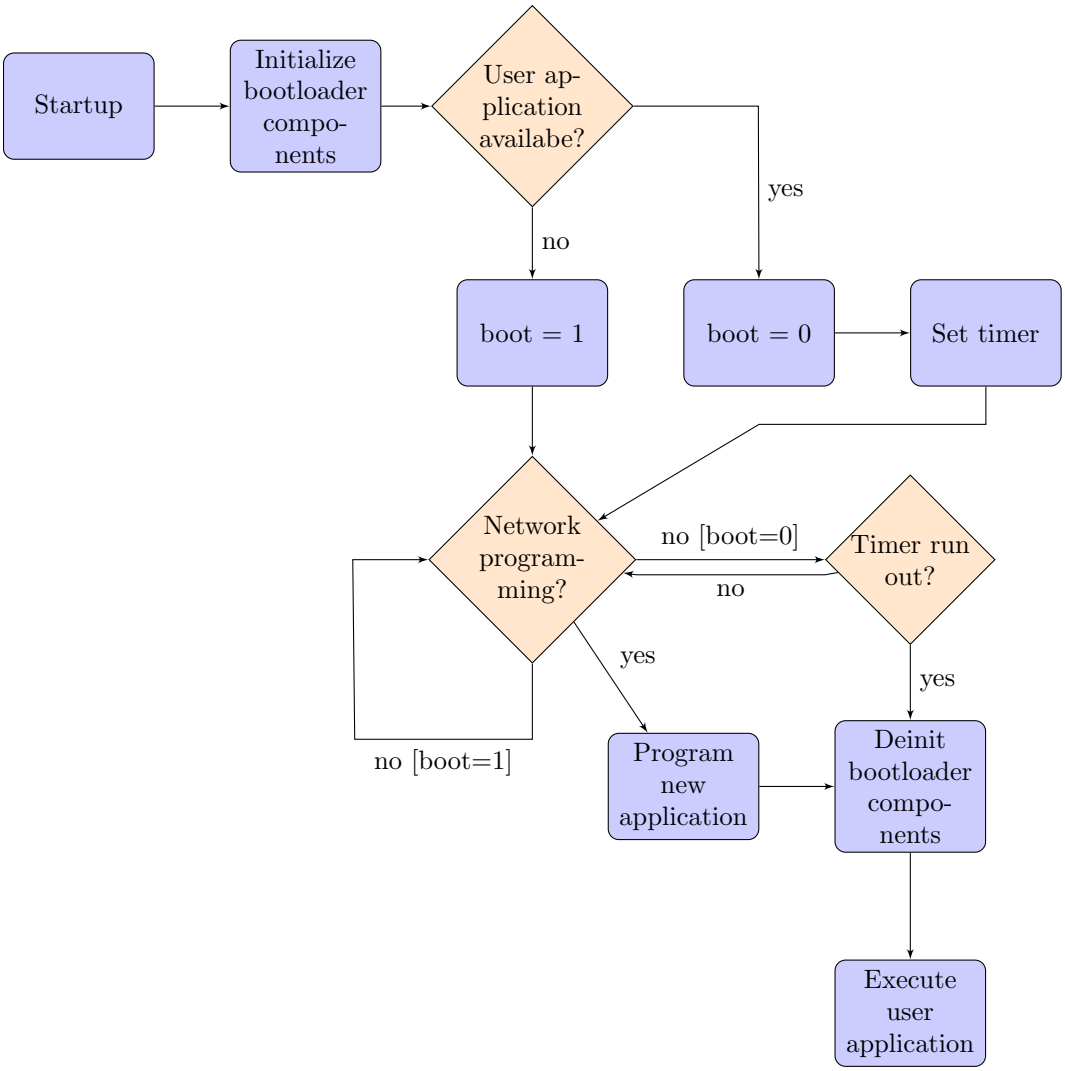


Figure 5.1: Flow chart of bootloading process. “Boot” is a flag indicating whether the node is in bootloader mode.

application.

5.3 CAN node identification

Every LPC1769 micro-controller has a 128-bit unique identifier, meaning that every node on the CAN bus is guaranteed to be unique. However, as discussed in subsection 2.1.1 a CAN frame has a 11-bit identifier, implicating that uniquely addressing a node cannot take place by using the CAN frame identifier.

Discussions by other users of the chip on NXP's discussion website indicate that using only the first four bytes of the unique identifier results in a decently unique identifier [8]. This approach is used as the network is small enough to use the smaller identifiers.

As the identifier is still too large to fit in the CAN identifier field, the identifier is placed in the data field. This implicates that only a maximum of four bytes can be used for other data in a CAN frame. As for the CAN protocol discussed in section 5.2, not much data is needed to be send along with the identification bytes. As such, this approach of node identification is used.

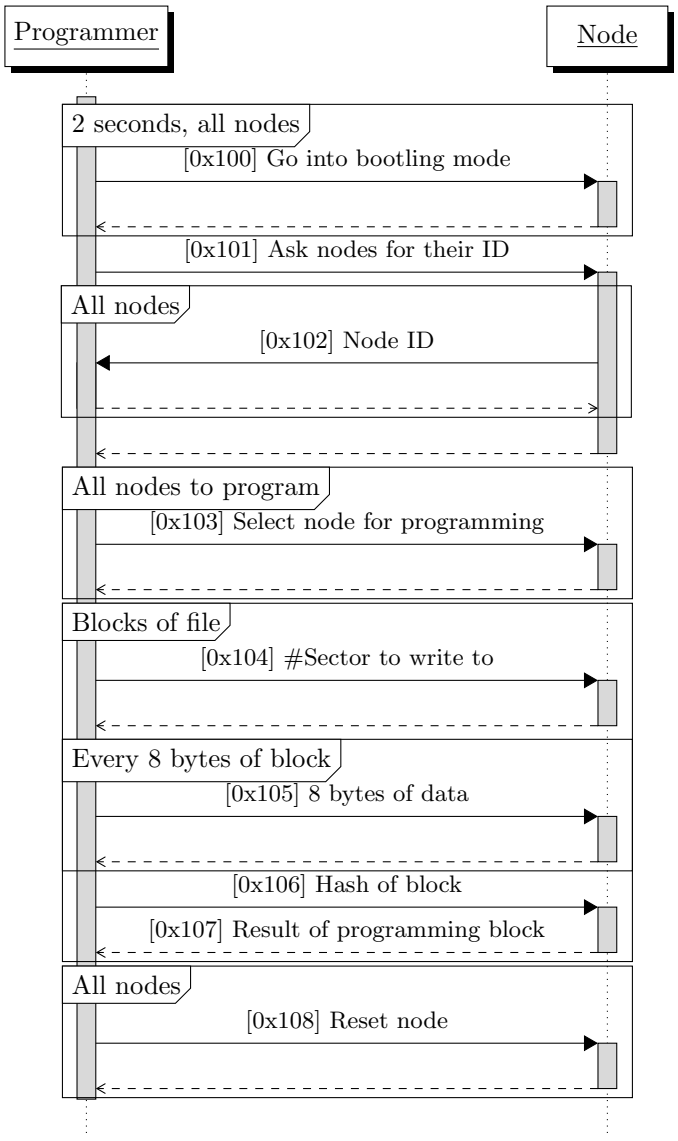


Figure 5.2: Diagram of CAN protocol used to program nodes.

5.4 CAN sequence check

The CAN bus protocol has a checksum field, as described in subsection 2.1.1. Using this checksum field the CAN receiver can verify the data of a CAN frame. What is not included in the CAN protocol is a way of checking whether the sequence of CAN frames received by the receiver is the same sequence as the CAN frames were sent by the transmitter.

As the sequence of CAN frames is vital for the successful programming of a binary file to a node, the sequence of CAN frames needs to be guaranteed. This can be either done by appending a sequence number to every CAN frame or to create a hash of the 4kB block.

Going for the sequence number approach means that two bytes are needed to indicate the number of a CAN frame. This is due to the fact that a 4kB block constitutes 512 CAN frames, as 8 data bytes go into one CAN frame. To send 4kB with sequence numbers $\frac{4096}{6} = 683$ CAN frames are needed. This results in a percentage increase of 33.4% of traffic on the CAN bus.

Another approach is the use of a hash function to create a hash of the 4kB data block. At the end of a 4kB block the transmitter and the receiver calculate

their hashes. The transmitter sends its hash to the nodes and the nodes respond with the result of the comparison of the hash. In this fashion, only one CAN frame is sent by the transmitter (the hash) and n are sent by the n nodes. As the n CAN frames sent by the nodes are also used for other result also used in the aforementioned approach, eventually only a single CAN frame is needed for sequence checking.

There are two types of hashes used during implementation. Firstly, a CRC32 algorithm was used. Secondly, a new iterative hash function was created and used.

5.4.1 CRC32

The first sequence checking implementation used an existing CRC32 library [1]. However, the generation of the hash took a large portion of time. The logic analyzer showed that the time to calculate the hash of the 4kB block takes roughly the same time as it takes transmit the 4kB. Checking the sequence of frames now doubled the time to download a binary file via CAN. This was not acceptable and as other implementations did not have significant speedups, another hashing scheme had to be found.

5.4.2 Iterative hashing

As the CRC32 implementation showed, starting the generation of a hash just after a block has been sent takes a significant amount of time. What would be more suitable is to have a scheme that generates a partial hash after each transmission of a CAN frame. This keeps the CAN bus busy, as the CAN frame is being sent while the hash is newly updated. When the last CAN frame of the block is sent, the total hash is directly sent. This generates little extra time for the transmission of a block on the CAN bus, see Figure 5.3.

Unfortunately, such an iterative hashing scheme was not found and thus had to be designed and implemented. A fast algorithm is listed in 1. This algorithm consists of four 32 bit hashes that are updated after each received CAN frame. Each hash represents two bytes of the data field on the CAN frame, e.g. hash^0 is based on the first two bytes of the data field. The algorithm solely uses addition and binary operations, making it very fast.

As a single CAN frame can consist of eight data bits not all hashes can be send, as the aim is to send only one CAN frame containing the hash. A recombination function is listed in 2, which creates two hashes out of the four hashes that remained after 512 times the calling the hash update function. The only thing this function does it performing the `XOR` operation on the first and the fourth hash and the second and the third hash.

The preceding algorithms are not scientifically proven, i.e. the strength of the hash function is not proven or determined. However, during the testing of a large set of data blocks, its corresponding hashes seemed to be unique. As this is not in any sense a scientifically determined property, it should not be used as a checksum function. However, because of the test results and because of its speed, the hash function seems a suitable function for the sequence checking problem.

5.5 Clock

A big part of the functionality and capabilities of the processor is defined by the clock the processor uses. There are a lot of different clocks in the processor, which all come from the same source and then get divided and multiplied to create a new clock.

5.5.1 Clock source

The first thing you have to decide is what clock source to use. There are 3 possible clock sources, the internal oscillator, the external oscillator and the real-time clock oscillator. The internal oscillator is a fixed 4MHz clock The external oscillator frequency is determined by an external component, on the LPCXpresso it is 120MHz. The real-time clock oscillator is a slow (32kHz) internal oscillator used for generating a 1Hz clock to keep time with.

The only peripheral on the bootloader that was influenced by the clock is the CAN peripheral. On several places in the user manual it says that you should not use the IRC oscillator if you want to use CAN speeds above 100kbit/s. Our upper speed limit for the CAN bus was 100kbit/s so we tried getting it to work with the IRC. After setting everything up our CAN communication didn't work perfectly, we had a lot of missed messages and error frames. With the logic we figured out that we were not getting a true 100kbit/s signal, it was about 103kbit/s and each node had a different deviation from the 100kbit/s speed. This was because the IRC is not precise enough, even though the user manual said it should be.

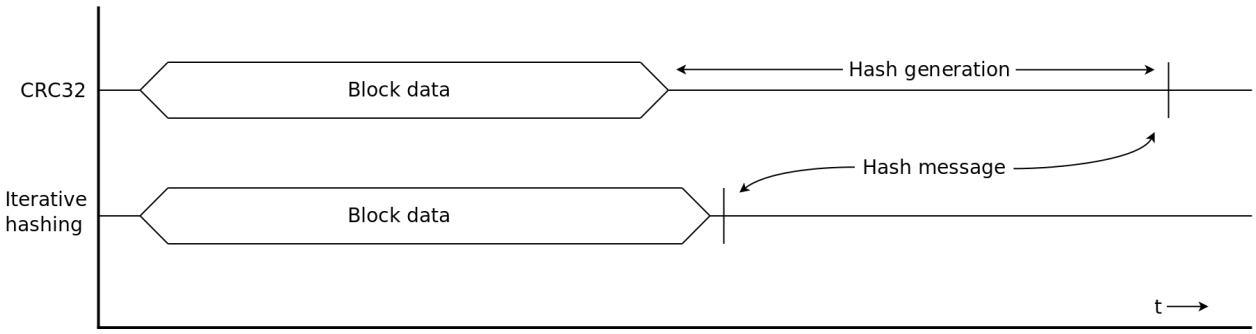


Figure 5.3: Difference in computation time between CRC32 implementation and iterative hashing function when sending 4kB of data over the CAN bus.

Algorithm 1 Updating set of 4 hashes after receiving CAN frame number q with *frame data*.

function UPDATEHASHES($(\text{hash}_{i-1}^0, \ldots, \text{hash}_{i-1}^3)$, frameData)
  for $i = 0 \rightarrow 3$ **do**
    fstNib $\leftarrow$ frameData$[2i]$
    sndNib $\leftarrow$ SHIFTLEFT(frameData$[2i+1], 8)$
    $\text{hash}_q^i \leftarrow \text{hash}_{q-1}^i + (q+1) \cdot$ (BITWISEOR(fstNib, sndNib)
    $i \leftarrow i + 1$
  $q \leftarrow q + 1$
return $(\text{hash}_i^0, \ldots, \text{hash}_i^3)$

Algorithm 2 Combining 4 hashes into 2 hashes.

function COMBINEHASHES($(\text{hash}^0, \ldots, \text{hash}^4)$)
  for $i = 0 \rightarrow 1$ **do**
    $\text{hash}^i \leftarrow$ BITWISEXOR$(\text{hash}^i, \text{hash}^{4-i-1})$
    $i \leftarrow i + 1$
  return $(\text{hash}^0, \text{hash}^1)$

After the IRC we tried the XTAL as the clock source, with which we got it working.

5.5.2 Dividers and multipliers

A lot of manipulations are done on the clock signal to get all the clocks the processor requires. The entire path is shown in Figure 5.4.

There are still a lot of constraints on what values you can set in what register and what speed the clock has to be at a certain point. For example, a constraint on f_{OCC} is that

$$275MHz \leq f_{OCC} \leq 550MHz$$

Our way of approaching this problem was to determine a wanted f_{CAN} and trying to work to that. The f_{CAN} is determined by the desired speed of the CAN bus,

$$t_{nominalCANbit} = (TSEG1 + TSEG2 + 3) \cdot \frac{1}{f_{CAN}}$$

$TSEG1$ and $TSEG2$ are time segments in 1 bit in the CAN standard. They determine at what point the bit is sampled. We will not elaborate on setting $TSEG1$ and $TSEG2$.

After determining f_{CAN} we can solve the following equation

$$f_{CAN} = \frac{\frac{\frac{f_{XTAL} \cdot MSEL \cdot 2}{NSEL}}{CPUCLKDIV}}{PCLKSEL0}{BRP}$$

and get our clock settings. NXP has provided a spread sheet which we used when solving this equation at `ics.nxp.com/support/documents/microcontrollers/xls/lpc17xx.pll.calculator.xls`. We ended with a CPU clock of 96MHz.

5.6 UART

As discussed in section 3.7 the host is connected to the programmer via a UART bridge. The UART protocol discussed in this section is able to send binary files from the host to the programmer.

5.6.1 Protocol

The UART protocol has to be able to send binary files to the programmer applications. The protocol also has to be able to either select specific nodes or select all active nodes for programming.

If the user application supplies a list of nodes to select, the host has to send the IDs of these nodes to the programmer. After sending the IDs to the programmer, the host starts sending data of the file to programmer. The file is split in blocks of 4kB and is sent one by one to the programmer. This is graphically depicted in Figure 5.5.

Besides supporting the programming of nodes, the protocol supports the scanning of nodes. This makes it possible to retrieve all the IDs of the active nodes. Having the IDs of such nodes makes it easier to selectively program nodes as explained above. The protocol to enable network scanning is graphically depicted in Figure 5.6.

The UART protocol is very extendible. The protocol uses a set of identifiers. These identifiers are the first messages of the protocol send by the host to the programmer. For example, for the programming of nodes identifier `0x01` is used, scanning uses `0x02`. Adding more identifiers is fairly easy and can be later be used to support debugging tools.

5.6.2 Transmission verification

The UART hardware on the nodes is able operate in full duplex mode, meaning that both sending and receiving can be done simultaneously. This functionality is used for verification of data transmissions from host to programmer. When the programmer receives data, it immediately retransmits the received data back to the host. If the host receives the same data it has sent, the data transmission was successful. The advantage of verifying data transmissions this way is that it does almost no extra time.

The preceding use of full duplex only verifies data transmission from host to programmer, not from the programmer to the host. Retransmitting data sent by the programmer to the host is not possible. This would create a cycle and the retransmission scheme would never end. Hence, data verification using full duplex is only possible in one direction.

As most data is sent from host to programmer, this direction is chosen to use the full duplex verification scheme. Also, the host sends applications to the programmer. Having an error in such applications is worse than having an error programmer to host communication, e.g. the ID of a node.

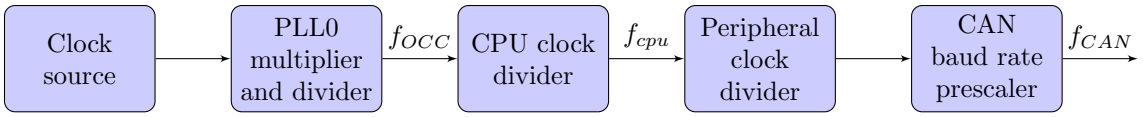


Figure 5.4: The clock signal path.

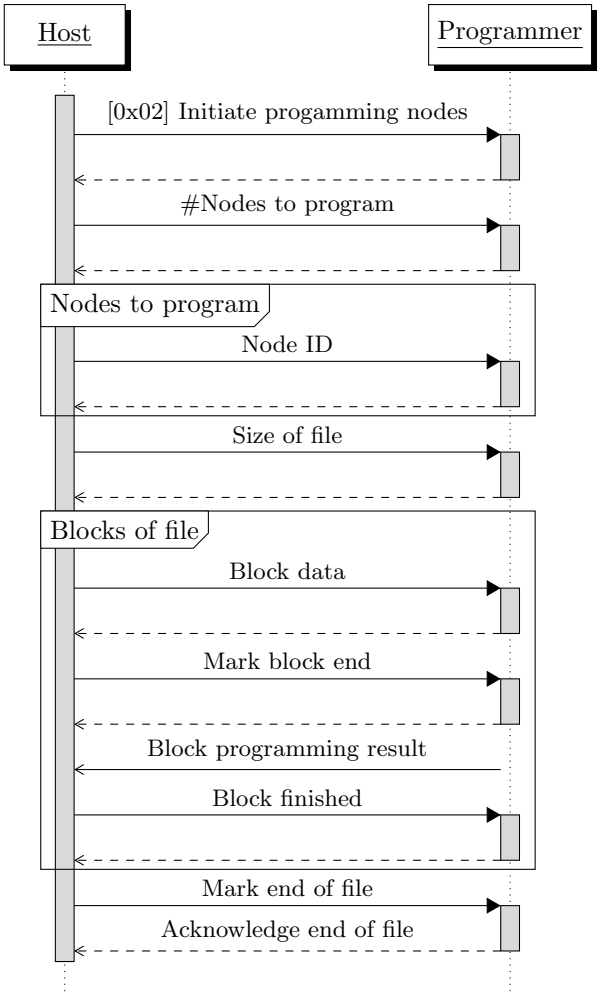


Figure 5.5: UART protocol for selective node programming.

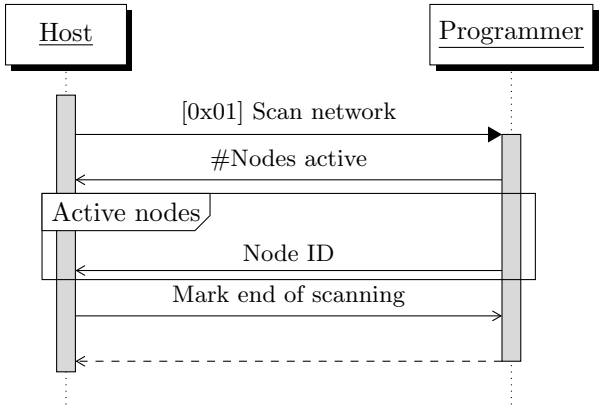


Figure 5.6: UART protocol for scanning of nodes.

5.7 Storage

As discussed in section 3.4, part of the design is to save two pointers at every node that are needed to start the user application on a node. As they need to used after rebooting, these pointers need to be stored in non-volatile memory. Two locations for the pointers were considered: the on-board EEPROM chip and the interrupt vector.

5.7.1 I^{2}C EEPROM

As discussed in section 2.1, this project uses the LPCXpresso LPC1769 development board. This board comes with an on-board EEPROM chip: the 24LC64, manufactured by Microchip. This chip has a storage capacity of 64kB and is connected to the LPC1769 microcontroller by a I^{2}C bus. The chip can be seen on the left side of Figure 5.7.

To be able to read from and write data to the EEPROM chip a driver has to be written. As interrupts are not enabled the driver needs to use a polling implementation.

The implementation of the I^{2}C driver worked fine, though it resulted in a significant larger codebase. This was not really satisfactional as a large portion of the codebase had the sole task to read and write just 8 bytes of data. Also, the use of this EEPROM chip made the bootloader less general, as additional

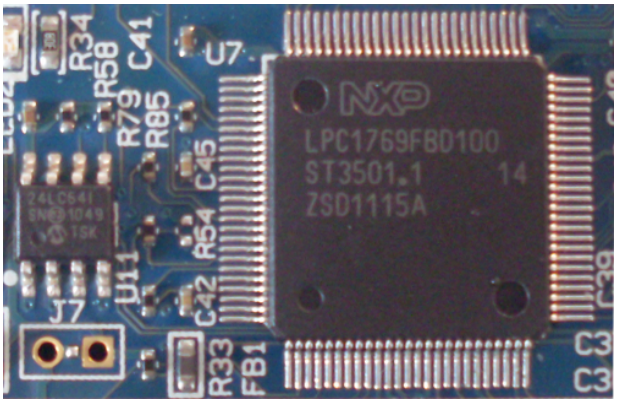


Figure 5.7: Location of the 24LC64 I^{2}C EEPROM chip (left black) and the LPC1769 microcontroller (right) on the LPCXpresso LPC1769 development board.

	⋮	
0x0024	*Reserved*	⟸ Start pointer
0x0020	*Reserved*	⟸ Stack pointer
0x001c	*Reserved*	⟸ Hash
0x0018	Usage fault handler	
0x0014	Bus fault handler	
0x0010	MPU fault handler	
0x000c	Hard fault handler	
0x0008	NMI handler	
0x0004	Reset handler	
0x0000	Top of stack	

Figure 5.8: Segment of LPC1769's vector table. Left column are the addresses of the pointers to various handlers described in middle column. Denoted are the new storage locations for the stack pointer and start pointer.

hardware is needed. Another approach was needed, without the use of the EEPROM chip.

5.7.2 Reserved interrupt vectors

After the implementation of the I^{2}C driver and the usage of the EEPROM chip, another non-volatile storage location was found. This was found in the LPC1769 chip itself.

In subsection 2.2.2 the memory model of the LPC1769 is discussed. Part of the memory model, the vector table is shown in Figure 5.8. It shows that certain parts of the vector table are reserved. Some of these parts are used by the microprocessor itself, for example the four bytes from location `0x001c`. These four bytes later turned out to be used to store a hash to verify whether a valid user application is stored on the microprocessor. Other parts of the vector table however seem to be not always used. The four bytes at `0x0020` and `0x0024` seem to be not used by any application tested. This was determined by looking at the values stored in these eight bytes for a set of applications. These values turned out to be `0x0000` for each application tested, thus unused. This indicated that the eight bytes are fit to serve as storage locations for the two pointers.

Due to the fact that the bootloader software is written in a modular fashion it was very easy to switch from the EEPROM implementation to the reserved interrupt vectors as storage location. During this implementation, the reserved location starting at location `0x001c` was first tried as storage location. This however failed. As these reserved locations have an undocumented function to the microprocessor, finding out why it failed to work was not easy. Ultimately it was found only what these four bytes serve for, and turned out to be vital in the flashing of the bootloader.

Now, as the bootloader does not need the EEPROM chip anymore, the bootloader removed an external hardware dependency. This is a huge advantage, also as other programmer that are programming for *proto*SPACE want to use the EEPROM chip for storing the eLua script so they can do viral programming.

As the current storage location has undocumented behavior, it can turn out to produce faulty behavior. Nevertheless, because of the modular fashion of the bootloader switching back to the EEPROM implementation is fairly easy and does affect little existing code.

5.8 Flashing

Once data from the programmer is received via the CAN bus, the nodes have to write the received data to the on-chip flash memory. This is done using IAP commands. Before IAP commands can be used, several steps have to be executed, such as the saving of the start and stack pointers and sector remapping.

5.8.1 Saving of pointers

When a node receives the first block of a new user application, several steps have to be performed for

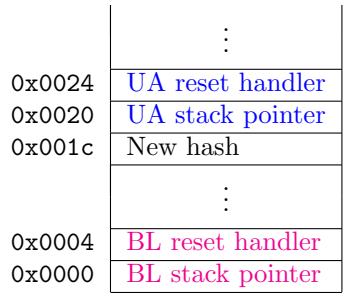


Figure 5.9: Segment of LPC1769's vector table showing user application's (UA) stack and rest handler pointer and the bootloader's (BL) stack and reset handler pointer.

the node to be able to start-up the user application later on, which is discussed in section 3.4. The first block contains both the pointer to the reset handler and the stack pointer of the new user application. As discussed in section 5.7, these pointers are stored in reserved locations of the vector table.

When a node resets, the reset interrupt handler of the bootloader should be executed. The address that the microprocessor looks for is at location `0x04`, so at this location the pointer to the bootloader's reset handler should be stored. Also, after reset the stack pointer should be set to bootloader's stack pointer, which should be located at `0x00`.

The user application's stack pointer and reset handler pointer are located in the first and the second four bytes, respectively, of the first block received. These values are read from the block and subsequently the values are stored in the vector table as shown in Figure 5.9.

Lastly, the microprocessor requires a hash of the first seven values of the vector table to be placed in the eight word in the vector table. As the bootloader changes some of the values in the vector table, the hash should also be updated accordingly.

5.8.2 In-Application programming

As described in section 5.2 the bootloader software receives binary files in a number of 4kB blocks. To flash this data on the on-chip flash memory, In-Application (IAP) programming is to be used. IAP consists of a set of predefined functions that user software can call to perform actions with the on-chip flash, e.g. writing to flash. The bootloader software uses these functions to write data received over the CAN bus to its flash memory. A subset of these IAP functions is actually needed to perform a write operation. This subset is listed in Figure 5.10.

Several steps are involved in a successful write operation of a 4kB block in RAM to the on-chip flash. Firstly, the sector to write to has to be *prepared*. Secondly, the sector has to be *blanked*. If the blanking procedure is not performed, arbitrary data that was not written to can be seen as instructions and can be executed as such. The sector has to be prepared before it is blanked, as blanking a sector is merely a writing operation of `0xFF` to the sector. Thirdly, a *blank check* is performed to determine whether the previous operation was successful. Fourthly, the sector has to be *prepared* again for writing to take place. Fifthly, the actual *write* operation takes place and writes 4kB to the on-chip flash. Lastly, the data stored in the sector is *compared* with the original data. This determines whether the flashing of the data block is performed successfully.

As mentioned before, not blanking a sector can result in the execution of arbitrary instructions. However, data is always written to the on-chip flash in blocks of 4kB. If all sectors are of this size blanking would be redundant, as the 4kB would be completely overwritten. This is not the case: the on-chip flash is divided into 16 sectors of 4kB and 14 sectors of 32kB. This means that the above sequence of functions calling has to be modified to cope with 32kB sectors. This is handled by *sector remapping*, discussed in subsection 5.8.3.

5.8.3 Sector remapping

Binary files are transmitted over the CAN bus in blocks of 4kB. Each of those blocks carries a sector number to denote the current 4kB block that is being transmitted. This sector number is defined as n_{vir}: the virtual sector number. To be able to flash 4kB blocks in both 4kB sectors as in 32kB sectors the virtual sector numbers need to be adapted.

This is done by a bijective function f that takes a virtual sector number as input and transforms it in a physical sector number and an offset in that physical sector number:

$$f : n_{\text{vir}} \rightarrow (n_{\text{phy}} \times s) \qquad \text{where}$$

$$n_{\text{phy}}(n_{\text{vir}}) = \begin{cases} (\frac{n_{\text{vir}}-16}{8}) + 16 & \text{if } n_{\text{vir}} \geq 16 \\ n_{\text{vir}} & \text{otherwise} \end{cases}$$

$$p(n_{\text{vir}}) = \begin{cases} n_{\text{vir}} \bmod 8 & \text{if } n_{\text{vir}} \geq 16 \\ 0 & \text{otherwise} \end{cases}$$

Function	Action
Prepare sector(s)	Required for write operation to specified sector(s).
Copy RAM to Flash	Copies data from RAM to on-chip flash.
Erase sector(s)	Fills sector(s) with string `0xFF`.
Blank check sector(s)	Checks if erase sector(s) succeeded.
Compare	Compares original and flashed data.

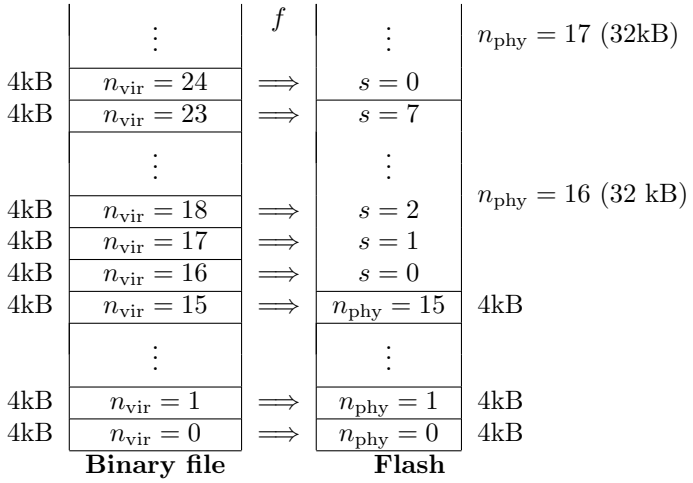
Figure 5.10: Subset of IAP commands necessary to perform write operations from RAM to on-chip flash.

Function f maps every 32kB sector to 8 sectors of 4kB. Here, the offset s is used to determine where in the 32kB sector the 4kB block is to be placed. If s is zero, it means that the block is to be put on the beginning of a 32kB block. The working of this remapping is graphically depicted in Figure 5.12.

The aforementioned sector remapping scheme also influences the order in which write operations are to be performed. For example, if there is a 4kB block and is remapped to a 32kB sector. This sector has to be cleared depending on the position of the block in that sector. If the block is mapped to the beginning ($s = 0$), then the whole 32kB sector can be cleared. Otherwise, if the block is mapped somewhere else in the 32kB sector ($s \neq 0$) then the block shouldn't be cleared as it would destroy all the previously written data in that 32kB sector.

Because of the added remapping scheme the sequence of steps to perform a write operation discussed in subsection 5.8.2 needs to be altered. Now, a sector only has to be cleared if $s = 0$. The altered version that incorporates the above remapping scheme is graphically depicted in Figure 5.11.

Figure 5.12: Sector remapping scheme. Binary file (left) containing virtual sector numbers (n_{vir}) are mapped to physical sector numbers (n_{phy}) of the flash (right). This makes it possible to receive blocks of fixed size (4kB) and flash them into sectors of variable size (4kB or 32kB).



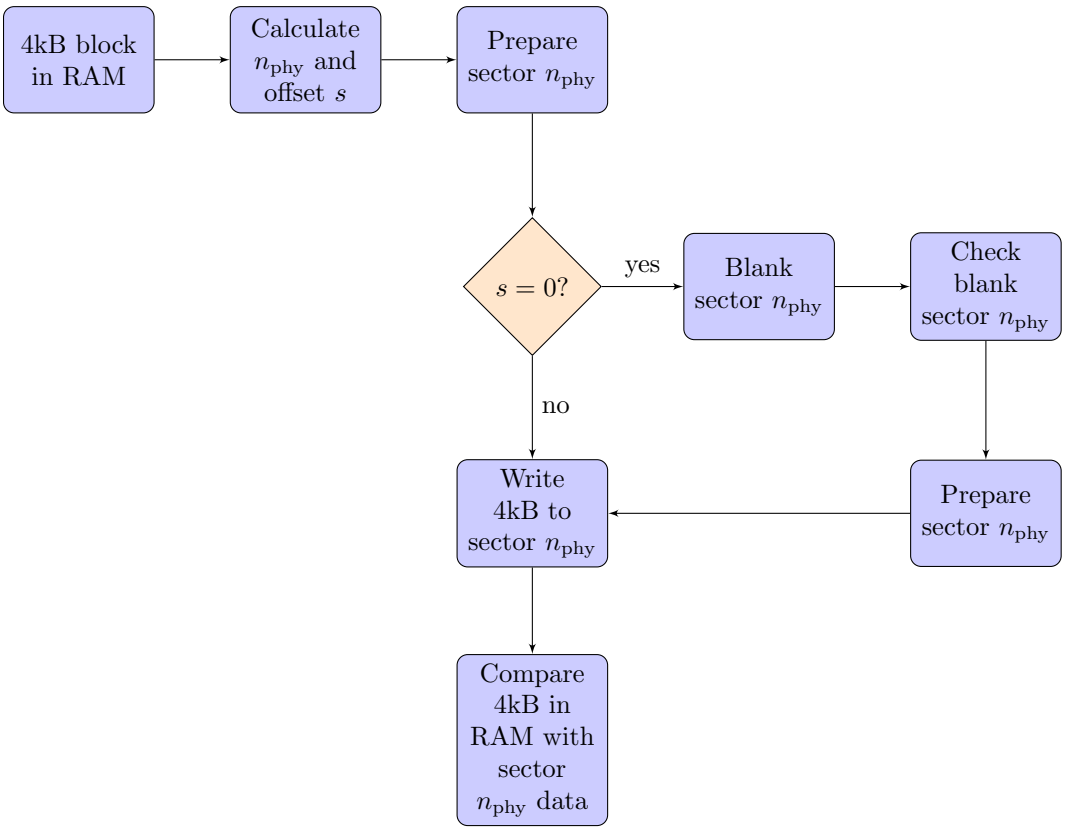


Figure 5.11: Flow chart of steps to perform a write operation from RAM to on-chip flash using IAP programming. Here, n_{phy} is the physical sector number, the number corresponding with the flash sector of the on-chip flash. Offset s is the offset between the virtual sector number and the physical sector number, as explained in subsection 5.8.3.

Chapter 6

Conclusion

The problem statement of this project was very interesting for the both of us. As it was a more advanced project, we knew that we would learn a lot about embedded systems. This was the main reason why we had chosen this project.

We have underestimated the difficulty of and the time needed for the project. Before the orientation phase, we thought there might be some time left to create other debugging tools. The most time consuming part of the project was debugging of mostly hardware related problems.

Also, some very specific knowledge was required to get thing working. For example, creating a suitable clock to run the CAN peripheral on the bootloader.

Although the project was a lot harder than expected, we have managed to create a well functioning network-based bootloader. The bootloader meets all the requirements set in subsection 3.2.1. During tests with a 16-node mesh, the eLua VM (170kB) was flashed in one and a half minutes on all nodes. Flashing the nodes one-by-one takes at least half a minute, so it's beneficial to use the bootloader for networks larger than three nodes when flashing the eLua VM. Also is the bootloader a lot less labor intensive.

The bootloader is really generic. The only hardware needed is the LPC1769 and an external crystal to have a stable CAN bus. Generality makes it possible to deploy the bootloader on all kinds of devices. For example, the code works on the DUTRacing car DUT11, seen in Figure 6.1, because the same processor and crystal is used. We think this shows that our work can be useful in more projects than just the *proto*SPACE floor.

One of the advantages of this bootloader is that the user applications do not have to be changed. We found no other bootloader for the LPC1769 for which you do not have to change the user application.

The source code of this project will be published on GitHub so others can also use the bootloader.

Though it took a long time to finish this project, we are very happy with the results and all the things we learned. We knew we took a risk with this project, but it really paid off.

6.1 Future work

The bootloader created in this project is just the scratching of the surface of what is possible with this hardware. With the CAN bus a whole debugging system can be created, to debug every node individually.

One of the things that would have been nice to be able to do during the project is updating the bootloader via bootloading a bootloader updater. This is just an application that links in the code of the new bootloader. The application contains IAP commands to write the new bootloader code over the old bootloader. By programming this application with the bootloader, new versions of the bootloader can be programmed using our bootloader.

Another, less important and wanted application is a graphical user interface to execute the command line tool of the host. This was needed for the current users of the bootloader, but might be handy for other users.

Figure 6.1: The DUT11 during a test-run

Bibliography

[1] Michael Barr. Crc implementation code in c. http://www.barrgroup.com/Embedded-Systems/How-To/CRC-Calculation-C-Code, accessed 25-july-2012.

[2] Steve Corrigan. Controller area network physical layer requirements. http://www.ti.com/lit/an/slla270/slla270.pdf, accessed 26-july-2012.

[3] Steve Corrigan. Introduction to the controller area network (can). http://www.ti.com/lit/an/sloa101a/sloa101a.pdf, accessed 26-july-2012.

[4] Vincent Driessen. A successful git branching model. http://nvie.com/posts/a-successful-git-branching-model/, accessed 3-may-2012.

[5] Steffan Karger. An embedded spatial computing platform for interactive environments. Master's thesis, TU Delft, 2012.

[6] Steve Franks Martin Maurer. lpc21isp. http://lpc21isp.sourceforge.net/, accessed 25-july-2012.

[7] NXP. *UM10360 LPC17xx User manual*, August 2010. Rev. 2.

[8] LPCXpresso users. Read serial number. http://knowledgebase.nxp.com/showthread.php?t=774, accessed 25-july-2012.

Appendix A

Flashing the bootloader manual

This appendix describes how to flash the bootloader software on the LPC1769 microcontroller with the *lpc21isp* tool. This is needed when adding more nodes to the network or replacing existing nodes.

As Code Red's LPCXpresso IDE limits the upload capacity of code to the LPC1769 to 128kB, files exceeding this limit cannot be uploaded. This barrier can be removed by buying a license from Code Red. As this is expensive and only adds more restrictions to the bootloader software, a more universal upload method is the use of the UART bootloader.

To use the UART bootloader, the open-source flashing tool *lpc21isp* is used [6]. This tool is used equip newly added nodes with the bootloader software, which is due to linker settings around 500kB in size.

To use the UART bootloader and the lpc21isp tool, execute the following steps:

1. Power off the LPC1769;
2. Connect the `ISP` pin with a ground pin. On the LPCXpresso LPC1769, the `ISP` pin is named as `P2.10`;
3. Connect a UART device to the LPC1769:
   - Connect `Tx` of the UART device to `Rx` of the LPC1769. On the LPCXpresso LPC1769, `Rx` is named as `P0.3`;
   - Connect `Rx` of the UART device to `Tx` of the LPC1769. On the LPCXpresso LPC1769, `Rx` is named as `P0.2`;
   - Connect the UART device's group to a ground pin of the LPC1769.
4. Execute the *lpc21isp*, which will poll the LPC1769 UART0 channel for a response some time. Use the following parameters: `lpc21isp -bin Bootloader.bin [UART device] 115200 14746`;
5. Power the LPC1769.
6. *lpc21isp* will now upload the bootloader code to the LPC1769.
7. The LPC1769 is now equipped with the bootloader software and can now be added to a network.

Appendix B

Project proposal

Problem

The hyperbody research group (`http://www.hyperbody.bk.tudelft.nl/`) in faculty of Architecture has the goal to

> explore techniques and methods for designing and building of non-standard, virtual and interactive architectures.

A room in the faculty of Architecture has been equipped with a floor with nodes in it. Each of these nodes has a RGB LED and a pressure sensor. This room is called protoSPACE 3.0 as seen in Figure B.1.

Every tile in the floor has its own processor. Every processor can only talk to its direct neighbours. Advantages of this is that you can just add new tiles with the code deployed on them and they will seamlessly integrate into the floor. Traditionally you would do this with a central server who controlls all the LEDs. From a certain size however one central server cannot handle the load. This is why sensor node technology is a solution. Even if you could implement the protoSPACE 3.0 floor without sensor nodes, with sensor nodes you can extend the floor infinitly without problems. Since the floor is a prototype there is one communication bus where all nodes are connected to, the CAN bus. The system is displayed in Figure B.2. The CAN bus is the red line and every node can talk to its neighbours via the black arrows.

The role of the Embedded Systems group in this effort is to research effective ways of implementing a node network which is easy to extend and program. The protoSPACE 3.0 room is already designed and produced, but still no effective way of programming and debugging is in place making writing code for the room a very tedious task.

We want to do our bachelor thesis with the Embedded Systems group and work on the toolchain features such as programming and debugging a node network.

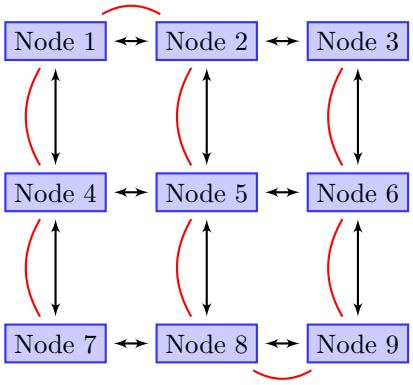


Figure B.2: A sensor node network with a CAN bus

Proposal

What we want to focus on for our bachelor project is programming individual nodes and the entire network over the CAN bus. Right now if you want to flash the entire network you have to go from node to node with your programmer. We want to develop a tool to flash a node. This could be done with a virtual machine in which we load code over the CAN bus, the department has experience with proto and eLua. We could also try to implement a CAN bootloader on the hardware. This a point of research in our assignment.

An extension of our assigment is distributed programming, so you give your code to a node which then updates all neighbours untill every node has the new code. The CAN bus is handy because you can address a single node in the network, for debugging purposes, but in real life in large networks you just want to flash an entire network without having a CAN bus run through all nodes.

To make this more clear we tried to compile a set

Figure B.1: The protoSPACE 3.0 room

of tasks we would have to accomplish.

- Familiarize ourselves in protoSPACE 3.0 hardware, which is:
  - The LPCXpresso 1769
  - Extra board the LPCXpresso 1769 is tagged on
  - The UART protocol for communication between nodes
  - The CAN protocol for monitoring and programming nodes
- Design the software to program the nodes
- Implement the program
- Couple our program with a GUI
- Showcase our findings with an application

We think this is a challenging assignment. With our current knowledge is it hard to estimate how long we would need to implement such a system. We do however think we should be able to implement programming the nodes over the CAN bus, and if we do finish that before the end of the project we can focus on an application in protoSPACE 3.0.

We will be supervised by Stefan Dulman (`http://www.st.ewi.tudelft.nl/~dulman/`). We can also get support from the master students in his research group.

Appendix C

Orientation report

Our main assignment is to implement tools to develop with on the *proto*SPACE floor. The first subproject was to build a CAN Bootloader for the LPC1769 and in this document will we discuss how we approached this problem and what design considerations we took into account.

Orientation

To orient ourselves on the assignment of implementing a CAN bootloader we did an example project. We implemented a CAN driver for the LPC1769 in the CMSIS library from NXP with the LPCXpresso shield as hardware platform, this is exactly the hardware platform that is planned to be installed in the floor of the *proto*SPACE room. After some struggling we managed to implement a new project which could send and receive messages on the CAN bus.

Platform

The hardware platform for our project is the LPCXpresso LPC1769. This board is plugged into a custom developed board called the LPCXpresso shield. We worked on the first version of the hardware board, in the new version only minor bugs are fixed and the board is physically smaller overall.

We worked with the assembly as seen in Figure C.1. An assembly very much alike this one will be implemented in the *proto*SPACE floor.

The boards are connected with their neighbours via UART and all boards are also connected to each other via a single CAN bus, as seen in Figure C.2.

CAN is a communication protocol implementing the lowest two layers in the OSI network layer model. CAN works by setting messages with an 11 bit ID and up to 8 bytes of data on the bus. Every node on the bus receives every message.

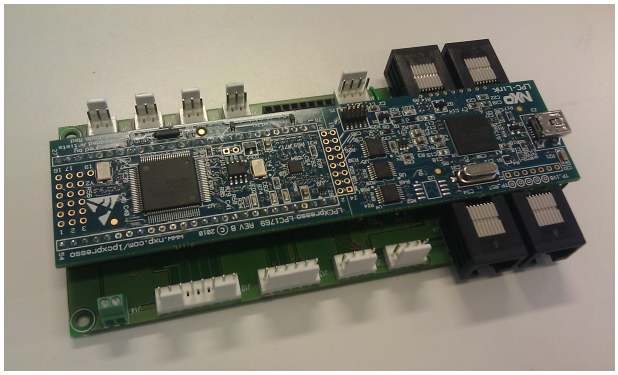


Figure C.1: The LPCXpresso LPC1769 and the LPCXpresso shield V1 assembly

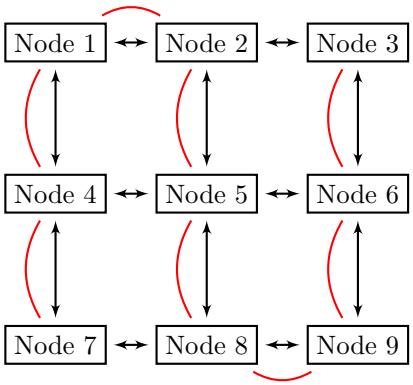


Figure C.2: A sensor node network with a CAN bus

Users of *proto*SPACE want to be able to load applications onto the nodes in the *proto*SPACE floor without having to lift any tiles. That is why a CAN bootloader is used.

Tools

The LPCXpresso platform comes with an IDE. This IDE is especially good for debugging code for the LPCXpresso: you can set breakpoints and step through the code while visually inspecting all the registers. The IDE will be very helpful while developing the CAN bootloader.

The LPCXpresso IDE is free to use for programs that are smaller than 128kB. As most development for the nodes is in a high level language like eLua or proto, a large image with the virtual machine should be flashed onto the nodes. To circumvent this restriction the built-in UART bootloader of the LPC1769 is used. To work with the UART bootloader cables are used to communicate between the computer and processor over UART. These cables can also be used to send debug information between the hostcomputer and the processor. We can keep this in mind while developing for the LPC1769.

Facilities

The faculty has an electronics room with most common equipment like soldering irons, breadboards and jumper wires. We can use these as we like for making test setups for the node.

The research group also has 2 Saleae Logics which are logic analyzers. With those logic analyzers we can monitor signals in the hardware which is very useful while debugging.

Design

After the orientation project we focussed on the main problem: the CAN bootloader for the LPC1769. We tried to do a complete as possible design of the software before starting implementation of the bootloader.

Overview LPC1769

The memory of the LPC1769 looks like Figure C.3. Our program must reside in the Flash memory along with the user program.

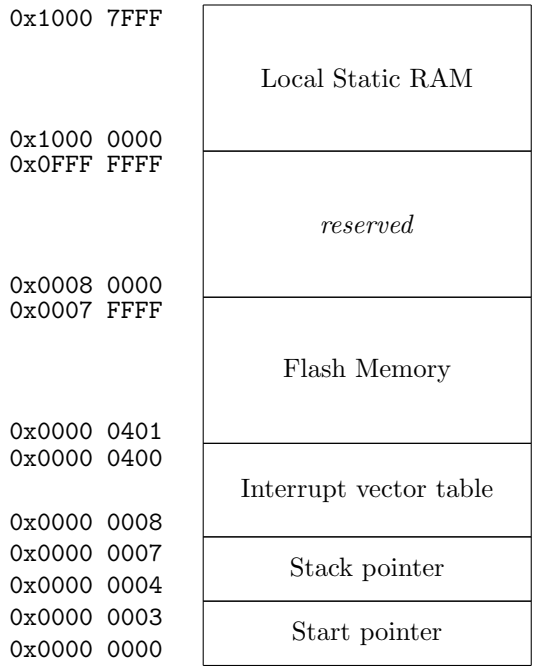


Figure C.3: Relevant parts of the LPC1769 memory

The LPC1769 has a lot of built-in peripherals, which are hardware functionalities. One can interact with these peripherals via registers. The CAN peripheral consists of 2 registers, which are needed to be used to perform all functionality. Implementing all functionality can be done with interrupts or by polling the registers. In the case of polling the registers, it is called a blocking implementation since no other function can be performed by the processor while waiting for something.

Functions and requirements

After analysis of what the bootloader was supposed to do we defined the following main function.

> A bootloader can update the user program in flash memory.

We use the bootloader as the program that can flash the node and the user program as the program that is flashed by the bootloader. After functional analysis of the problem we identified requirements to within implement the bootloader.

- The bootloader must work on the LPCXpresso LPC1769.

- The bootloader must be able to flash the node while the programmer can only communicate with the node over CAN.
- The bootloader must be able to program one specific node in the network.
- The bootloader must be able to detect which nodes in the network are in bootloading mode.
- The node must be able to go into bootloading mode from all states of the user program.
- The bootloader should be able to detect all nodes in the network with the bootloader software;
- The bootloader should be able to detect write errors and be able to communicate them to the programmer.
- The bootloader should use as little ROM memory as possible.
- The bootloader should be as transparent as possible to the programmer that uses the bootloader. With this requirement we mean that the programmer that is using this bootloader shouldn't have to think about it, he should not notice that there is a bootloader in the ROM where his program also is.
- Flashing all 200 nodes in the network should be able to be done in less than 5 minutes.

We will discuss how we are planning on implementing different aspects of the bootloader and why we want to do it that way.

Location of the bootloader

The bootloader has to be in ROM with the user program, there is no other option. For the place of the bootloader we identified two bootloader location concepts mixed with the use of interrupts or no interrupts:

- Compiling the bootloader with the user program;
- Placing the bootloader at the top of the flash memory and implementing all the peripheral functions polling;
- Placing the bootloader at the top of the flash memory and implementing the peripheral functions with interrupts;
- Placing the bootloader at the bottom of the flash memory.

Compiling in user program

In this concept you make a function that the user program should call before executing its own program. The bootloader and the user program are than in the same image.

We identified some problems with this approach. Foremost the bootloader code is somewhere among the user application code, which means that the code the bootloader is replacing is also the place where the bootloader code itself is. This could be solved by linking the bootloader code like it is in RAM and then have a piece of code that copies the bootloader to RAM and starts it.

Another problem is that if someone accidentally flashes the wrong image or made a mistake in the program the node becomes bricked. The programmer will have to go to the physical location of the node and reprogram it via the JTAG or UART bootloader.

The bootloader itself is also not transparent to the user. The user application has to change to facilitate the bootloader. This makes the bootloader very platform bound and less re-usable.

Top of flash polling

In this concept we put the bootloader code at the top of the flash. We overwrite the start pointer during bootloading by the bootloader to point to the bootloader at the top while saving the original start and stack pointer value. So the bootloader is called at startup, it can then set up everything needed for the bootloading and then start the user program. Every interaction with the peripherals is done via polling the registers. So the interrupts are disabled when the bootloader starts so that user application interrupts do not execute.

We have also identified some problems with this approach. The CAN bootloader has to be used to load the user program into flash, programming via the JTAG interface or the UART bootloader will overwrite the bootloader if the bootloader is not in the image.

The user program might try to use the memory where the bootloader is placed. Some user programs use the flash memory for storing variables needed between reboots. Using the top of the flash for that purpose is common practice.

If the bootloader runs during the user program the bootloader does not know the settings of the clock and other peripherals. This makes the bootloader very unreliable since the speed of the CAN peripher-

als is related to the speed of the main processor clock. The user program can also disable the CAN peripheral completely and interfere with the bootloader.

A requirement is that the program must be able to go into bootloader mode at all times without powering down the network to force all processors to restart. The user application must restart the node in case the nodes needs to be reprogrammed, so the implementation is not completely transparent to the programmer.

Keeping time on an ARM processor is done via the systick interrupt but in this concept interrupts are disabled. Since the processor has a known clock the time each instruction takes can be deduced. Waiting for a certain amount of time can be approximated by looping a constant amount. The bootloader does not have any hard real-time constraints.

Since the user program and bootloader are now completely separate every program that does not use the top of the flash can be flashed onto the node without modification. Only restarting the node via CAN is then not implemented.

Top of flash interrupts

Same as previous concept but now instead of polling the bootloader is implemented with interrupts. To achieve this the bootloader must overwrite all interrupts during downloading and have them points to the bootloader code, the same thing that has to happen to the start pointer and the stack pointer.

Now the bootloader can intercept all CAN messages so the bootloader can restart the node when the re-program message comes, even during the user program. The bootloader is now completely transparent to the programmer. The user program can still interfere with the bootloader by changing the CAN settings or the processor clock. Most user programs change the clock to use an external hardware crystal. This would change the speed of the CAN peripheral disrupting communication with the programmer.

Bottom of the flash

The previous concepts both placed the bootloader at the top of the flash. The bootloader can also be placed at the bottom, the linker script of the user program then has to be changed to not use that piece of flash memory.

The difference between the top and bottom flash concepts is essentially that the bottom of the flash concept interferes with most user programs while at the top of the flash the bootloader only interferes with some user programs.

Chosen concept

The bootloader will be implemented at the top of the flash. This is because this interferes with less user programs than a implementation at the bottom of the flash.

The bootloader will be implemented using polling for all peripherals and timing. Overwriting interrupts is hard to implement and no real gain is achieved. The bootloader does not become more transparent to the programmer since the programmer still has to change settings for the CAN such as the peripheral clock divider.

CAN Protocol

We must design a protocol for the nodes to talk over the CAN bus. At the time of writing is there no standard protocol for CAN bootloaders. This bootloader must also be able to efficiently support flashing multiple devices, which most current bootloader protocols don't support. We will refer in the next section to the programmer as the node that is flashing other nodes in the network. The programmer is also a LPCXpresso LPC1769 with shield.

Initialization

In the initialization phase of the protocol the programmer figures out which nodes are on the bus. The programmer first gets all the nodes on the bus into bootloading mode. All nodes at reset wait for about 1 second on the message 0x100. If a node receives that message in that time they will go into bootloading mode and wait for further commands. The user program should contain code that if the node receives the 0x100 message that the node should restart. The programmer at the start of the bootloading sends the 0x100 message for an amount of time with as goal to get every node in bootloading mode.

Now that every node is in bootloading mode, the programmer should know which nodes are on the CAN bus. The programmer sends a 0x101 message on the bus as a request for every node to register itself. All nodes then respond with a 0x102 message with their ID as data. A lot of nodes will experience bit errors in this phase. The nodes should just continue until they have sent their ID. The ID of the node is based on the unique processor ID.

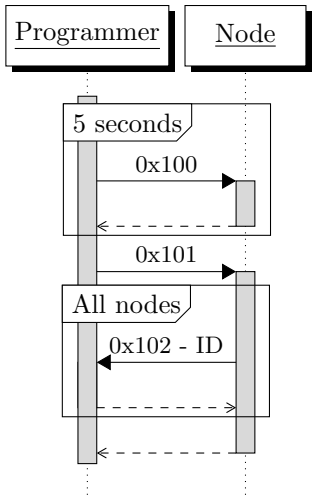


Figure C.4: The initialization period

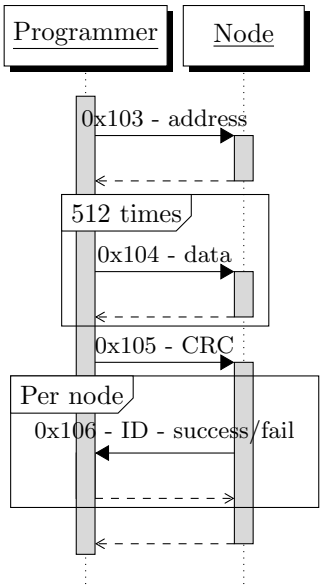


Figure C.5: Sending a sector 4kB over the CAN bus

After a certain amount of time the initialization period is over and the programming of the nodes can begin.

Downloading

The nodes now need to get flashed with new firmware. This is done by first selecting all the nodes that we want to flash with new firmware. The programmer does this by sending a 0x102 message with the node ID to every node that needs to be flashed. So if a node sees a 0x102 message containing its ID it starts listening to the data from the programmer. If a node was not selected for flashing it disregards the data from the programmer.

The programmer sends the new program in blocks of 4kB, the biggest chunk of data a node can copy to flash in a single IAP command. The node must keep track of what sectors need to be cleared and what sectors do not need to be cleared. The node also automatically clears the flash sector. The 4kB are sent per 8 bytes in message 0x103. After the 4kB, the programmer sends a 4 byte CRC in message 0x104. With that CRC the nodes can determine if the data they received is correct. Every node that is listening to the data needs to respond to the 0x105 message with a 0x106 confirmation message containing information of whether CRC passed or not. If a node does not respond to the 0x105 message or if a node failed the CRC the programmer sends the entire section again upto 3 times. After 3 times the flashing fails.

The bootloader must take the following constraints into account:

- Interrupts must be disabled or interrupt handlers must reside in RAM during flash programming
- You must prepare the sectors you are going to write to with an IAP command before trying to write to it.
- You can flash blocks of 256, 512, 1024 or 4096 bytes.
- IAP commands use the top 32 bytes of RAM.

If the programmer wants to write to the first sector the bootloader must change the start pointer to point to the start of the bootloader. The bootloader must however save the start of the user program, it saves it in the Eeprom on the LPCXpresso together with the stack pointer.

Executing

Now some nodes are flashed, they all have to start up the new program. The programmer sends the message 0x107 to reset all the nodes. The nodes just restart and startup the bootloader. If the bootloader does not receive the message 0x100 the user program is started.

Implementation plan

We want to stepwise implement all functionality. For all the functions we want to make a minimal working

Table C.1: CAN IDs and their meaning while bootloading

ID	Length	Data	Meaning
0x100	0	-	Go into bootloading mode, reset into bootloader
0x101	0	-	All nodes in the network register
0x102	4	Node ID	Node with Node ID listen to data
0x103	2	Address	Address of the coming 4kB block, 256 bit addressed
0x104	8	Flash data	Data to be flashed
0x105	4	CRC	CRC of the data to be flashed
0x106	5	Node ID + CRC/flash correct	The Node ID and if the CRC/flash was correct
0x107	0	-	Reset the node

example showing that it is possible with the least interfering factors. We also want to work in parallel on features. Right now we are thinking of producing the following small applications to in the end get a bootloader:

- Application at top of flash.
- Application at top of flash that calls back to application at the bottom of flash.
- Application that reads and writes to eeprom.
- Application that flashes an unused sector and reads it back.
- 2 node CAN network that sends data with CRC.
- 2 node CAN network that implement communication protocol.
- 2+ node CAN network that implement communication protocol.
- Bootloader!

This all should finally produce a working bootloader.