CAN Bus Bootloader for STM32 Embedded Microcontrollers

Zavatta Marco, Yin Zhining [*]
Supervisor: Martino Migliavacca, AIRLab

July 12, 2012

Abstract

A boot loader is a necessity for any microprocessor; with it users can flash the applications they want into memory to make the microprocessor do what they want. Otherwise, nothing is executable. Our project aims at developing a software CAN bus boot loader for STM32F10X family. STM32F10x family microcontroller contains both Serial port (USART) and CAN bus port. STM only embeds a USART-based boot loader for consumers, although a CAN solution would carry many advantages, first and foremost the possibility of bootloading many devices simultaneously due to the bus nature of CAN. Our solution is to adopt the application-level protocol used by STM in the embedded boot loader, implementing it on top of CAN bus, so that compatibility and reuse of existing tools can be maximized. Our boot loader enables to:

- Transfer an object file from an host (PC) to the device Flash memory
- Read the contents of the device Flash memory from a given address
- Jump to a given address and start execution

Keywords: CAN bus, bootloader, STM32

[*]marco.zavatta@mail.polimi.it 766437, zhining.yin@mail.polimi.it 766473

Contents

List of Figures

1 Introduction

A bootloader is a piece of software that intermediates between the device reset and the firmware that is going to be run. Its basic feature is to program instructions into the memory of the processor or, depending on the user desire, let run instructions from a specified memory address. A bootloader has to be programmed in the program memory of the microcontroller just once, using a conventional programmer (e.g. through JTAG). After this, the microcontroller can be programmed without a programmer. ST Microelectronics refers to this process as In-application Programming.

Once in the microcontroller, the bootloader is such programmed that each time after reset it starts running like any conventional program, as it lies at the default program counter value of the processor after reset. What it does however is different from a regular program. First of all, depending on what type of bootloader it is, it starts "listening" for incoming bytes via a specific interface. For instance, a USART bootloader will listen to the USART buffer of the micro, checking for incoming bytes. If the bytes start arriving, the bootloader will identify the actions that the user tells it to perform. If the desired action is to program the memory, it will grab incoming data and write them in the program memory in the sequence it receives them and at predefined locations. Once all bytes have been received, the bootloader executes a jump at the start of the memory zone it has received and then the user application starts running. The above example is only one of the ways in which a bootloader may function.

The most common one used to be the USART bootloader [1]. They are still widely used today as it is a common interface for many microcontrollers. This type of bootloader basically removes the need for a programmer. You only need a PC software (cheap and easy to make/find) and a serial cable. Gradually, after the appearance of USB interfaces on micros, the USB bootloader came into existence. It performs the same functions and is pretty handy to use. Another more special type of bootloader is the SPI/I2C bootloader. Basically, the way it works is similar to any bootloader, but instead of listening to UART/USB it tries to read data from an attached device, for instance an EEPROM memory, with an I2C or SPI interface. This bootloader is designed for allowing easy upgrade of software in the field. You could attach the SPI memory in a socket, and a firmware change might mean as little as changing an IC in a socket. Of course in order to do this, you need to have physical access to the device in question.

The CAN bootloader, as the one constructed in this project, is extensively used in the automotive industry. Most of the devices to be found in a car have a CAN port nowadays. Due to this fact, bootloaders that work with the CAN interface of the micro have been developed, and therefore you may upgrade the firmware of such a device without even opening its housing or without even taking it out of the circuit, due to the networking prone nature of CAN.

2 Technology Used in the Project

We will now review the communication technologies and the microcontroller platform for which the bootloader was developed. We will give a general overview, expanding the description on the features of the microcontroller which are more relevant for the purpose of the project.

2.1 Embedded Communication Technologies

2.1.1 USART

The Universal Syncronous Asyncronous Receiver/Transmitter is a serial communication mechanism, independent of the actual serial signalling standard used, which provides a way to transmit and receive bytes in a full-duplex way. In our project we use the asyncronous mode.

To establish a UART communication, two sides must agree on the serial signalling, bit rate, packet length (usually 8 or 9 bits) and nature and duration of start and stop bits. Being an asyncronous protocol, there will be a default logic value on the channel. An incoming packet is detected by a receiver when the default logic value is changed by the start bit of the incoming packet. From this point on the receiver will start sampling the channel level at each bit cycle for as many bits as agreed. If the two sides are compatible, after the data bits it will come the stop bit. The receiver turns the sequence of bits into a word and passes it to the application.

2.1.2 CAN

The Controller Area Network (CAN) standard is a communication technology developed by BOSCH in 1986. The 2.0 version was released in 1991 [2]. The latest version features data rates up to 1 MBit/s for network lengths of up to 40m. Single or double wire CAN standards exist. It is widely applied in automotive systems. It is a bus-based, serial, broadcast, asyncronous (no clock is sent over the wires) protocol which uses frame-level priority-based arbitration as the medium access control mechanism. It spans many layers of the ISO/OSI model, see Figure 2 for an overview.

The priority is assigned to each message on the bus and is expressed by the arbitration field in the frame (also called identifier). The mechanism relies on a binary model of dominant and recessive bits where dominant is a logical 0 and recessive is a logical 1. The logical levels imposed by the devices on the bus combine when starting to send the frame, and if one device places a dominant bit and another device places a recessive one, the channel will have a dominant value. The device that placed the recessive level will recognize the difference and will back-off due to lost arbitration. An all-zeros identifier will therefore always win the arbitration. The master with highest priority will win the arbitration and start transmission, the others will back-off and start listening on the bus for incoming frames. It is interesting to note that this arbitration mechanisms doesn't add time overhead to the communication except the time used to transmit the identifier. It doesn't add hardware or wiring overhead neither, as the mechanisms is distributed on the nodes, there is no centralized arbiter and the logic happens directly on the bus wires.

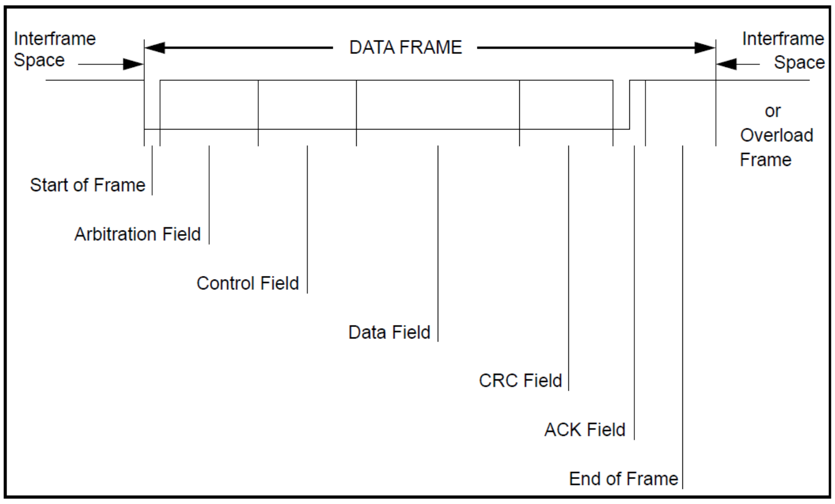


Figure 1: CAN Data Frame Fields

The detailed CAN protocol stack is shown in Figure 2. Message filtering is usually provided by the CAN controller by means of registers where allowed identifiers can be specified. In that case only the compliant incoming messages will be passed onto the software. This particular feature of CAN enables efficient multicast communication on an essentially broadcast medium.

The frames can be of various types (data, error...) and extension. The data frame is shown in Figure 1. The most common frame fields are:

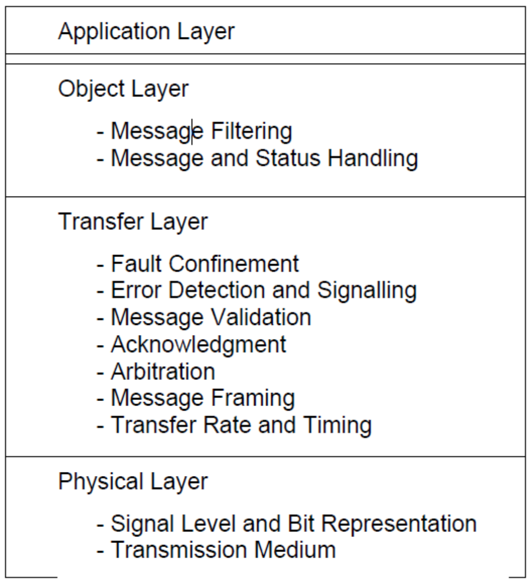


Figure 2: Layered Structure of a CAN Node

- Arbitration or identifier field
- Data, up to 8 bytes per frame
- Checksum byte

It is interesting to note that each frame enforces a CRC byte to be sent. CAN controllers pass onto the software only valid data. Many error bits are present for the SW in CAN controller registers to monitor the communication status. This is unlike USART where checks are simpler e.g. parity bit or hardware flow control to prevent buffer overflow. Also, hardware flow control is often not implemented.

Each member of the CAN network has its own clock generator, usually based on a quartz oscillator. The length of the bit time (i.e. the reciprocal of the bit rate) can be configured individually by setting a prescaler (in our case by writing specific CAN controller registers) for each CAN node, creating a common bit rate even though the CAN nodes' oscillator periods may be different. Anyhow, the frequencies of these oscillators are not absolutely stable. As long as the variations remain inside a specific oscillator tolerance range, the CAN nodes are able to compensate for the different bit rates by resynchronizing to the bit stream. According to the CAN specification, the bit time is divided into four segments (see Figure 3): the Synchronization Segment, the Propagation Time Segment, the Phase Buffer Segment 1, and the Phase Buffer Segment 2. Each segment consists of a specific, programmable number of time quanta. The Synchonization Segment is when, according to the internal oscillator, and edge is expected to occur. If an edge occurs out of this time segment, a resynchronization of the internal oscillator is performed. The Propagation Time Segment accounts for delays caused by the edge propagation time between two nodes. It must be set so that a sequence of more than two bits is enured to be sampled within the window given by Phase Buffer Segment 1 + Phase Buffer Segment 2 by all nodes on the network. This window is when the sampling of the logical bit level occurs.

CAN, unlike the simple USART, has all functionalities that facilitate networking over a single bus. A device equipped with a CAN controller can be attached to the bus wires and begin to interact with other devices. Priorities can be quickly readjusted by changing the identifiers sent with the messages. CAN controllers are usually integrated into the microcontroller chip, as it is the case in our STM32, although a CAN trasceiver is needed outside the chip. NXP recently marketed the first microcontroller with on-chip CAN transceiver.

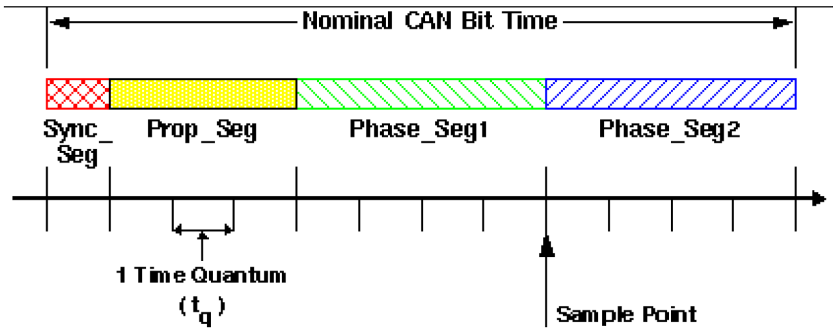


Figure 3: Bit Timing Parameters of a CAN Node

2.2 STM32 ARM Cortex-M3 microcontroller

The STM32 Cortex-M3 microcontroller is designed by ST Microelectronics. It features a 32-bit ARM Cortex-M3 as processing core and embeds many other peripherals such as communication interface controllers. The one used for this project is STM32F103CBT6 [4], which falls into the medium-density range. It is placed into a PCB which connects to it, among other things, buttons, LEDs and a CAN transceiver. Interconnections within the micro are based on AMBA buses, each communicating through bridges and each having a CLK prescaler which can be set by the user through appropriate registers. The architecture of the microcontroller is shown in Figure 4 and the clock tree in Figure 5. For this project, we used the GPIO controllers on APB2 and the CAN controller on APB1. The STM32 we used has the memory map shown in Figure 6. It has 128 kBytes of Flash memory, starting at 0x0800 0000 and 20 kBytes of SRAM at base address 0x2000 0000. Refer to [5] for further details.

2.2.1 Memory Organization

As shown in Figure 6, the 32 bit addressing space is used for multiple purposes. Some addresses are available to the user, some others are reserved or unused. Unlike other microcontrollers, there isn't dedicated memory space for custom bootloaders; they must be placed in common flash. Therefore, any mass erase of the Flash memory by software must be carefully checked not to erase the custom bootloader code.

The Flash Memory organization is based on a main memory block containing 128 pages of 1k bytes starting from address 0x0800 0000. Flash memory also contains a information block divided into system memory and option bytes. The system memory is where embedded boot loader is installed by the vendor during fabrication. The option bytes are configured by the user depending on the application requirements. For example, by configuring option bytes, it is possible to write and read protect the Flash memory.

2.2.2 Low-level STM32 Cortex-M3 Programming

Featuring an ARM core, the STM32 is programmed by ARM assembly instructions. The Cortex-M3 uses the Thumb instruction set, which is a reduced instruction set with instructions 16 bit long. The memory is viewed in little endian. Software has two privilege levels, namely unprivileged and privileged, which restrict the ability of the software to manipulate hazardous components. For example the Cortex-M3 manual specifies that the software triggering of a device reset can only be performed by Privileged code. Privileged code can be for instance a SVCall body or any other exception handler code.

The exception model of the core treats reset as a special exception type. All the exception handler base addresses are stored in the vector table of the program (Figure 7), which is pro-

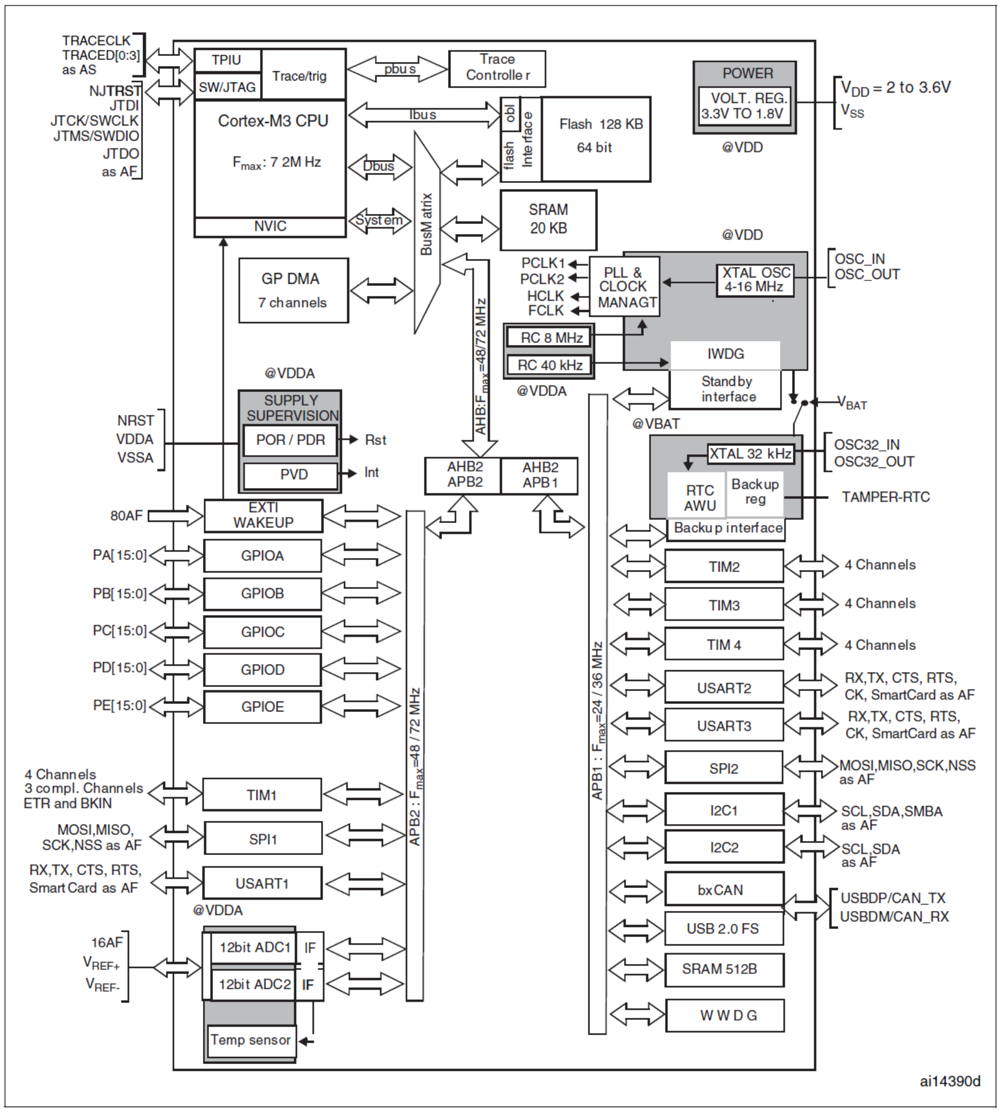


Figure 4: Architecture of the STM32 Family

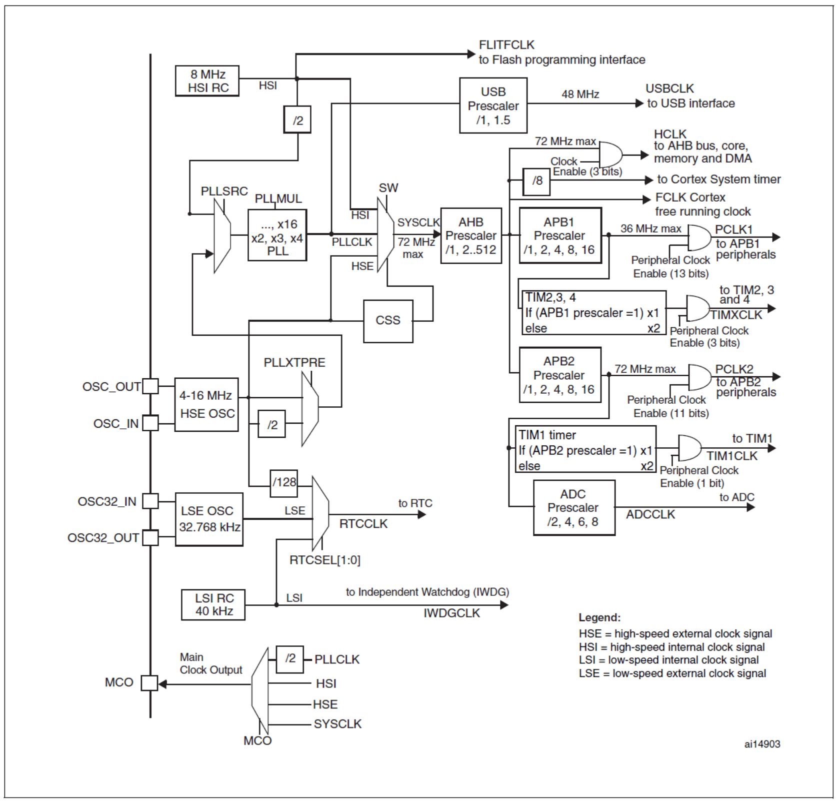

FLITFCLK
to Flash programming interface
8 MHz HSI RC
HSI
/2
USB Prescaler /1, 1.5
48 MHz
USBCLK to USB interface
72 MHz max
HCLK to AHB bus, core, memory and DMA
Clock Enable (3 bits)
/8
to Cortex System timer
FCLK Cortex free running clock
PLLSRC
PLLMUL
SW
..., x16 x2, x3, x4 PLL
HSI
SYSCLK
72 MHz max
AHB Prescaler /1, 2..512
APB1 Prescaler /1, 2, 4, 8, 16
36 MHz max
PCLK1 to APB1 peripherals
Peripheral Clock Enable (13 bits)
PLLCLK
HSE
CSS
TIM2,3, 4 If (APB1 prescaler =1) x1 else x2
to TIM2, 3 and 4
TIMXCLK
Peripheral Clock Enable (3 bits)
PLLXTPRE
APB2 Prescaler /1, 2, 4, 8, 16
72 MHz max
PCLK2 to APB2 peripherals
Peripheral Clock Enable (11 bits)
OSC_OUT
OSC_IN
4-16 MHz HSE OSC
/2
TIM1 timer If (APB2 prescaler =1) x1 else x2
to TIM1
TIM1CLK
Peripheral Clock Enable (1 bit)
/128
ADC Prescaler /2, 4, 6, 8
to ADC
ADCCLK
OSC32_IN
OSC32_OUT
LSE OSC 32.768 kHz
LSE
to RTC
RTCCLK
RTCSEL[1:0]
LSI RC 40 kHz
LSI
to Independent Watchdog (IWDG)
IWDGCLK
Legend:
HSE = high-speed external clock signal
HSI = high-speed internal clock signal
LSI = low-speed internal clock signal
LSE = low-speed external clock signal
Main Clock Output
MCO
/2
PLLCLK
HSI
HSE
SYSCLK
MCO
ai14903

Figure 5: Clock Tree of the STM32 Family

duced by the compilation process. At reset, the program counter jumps into the reset handler address and exectution is started. In the event of an SVCall, for example, the corresponding address is looked up in the table and the handler executed. Using our development tools, it is possible to specify which function call has to be referenced for each entry of the exception vector table.

2.2.3 Embedded Peripherals and their Usage

The microcontroller embeds many peripherals external to the processing core (communication controllers, CRC calculators, DMA etc..). Their technological characteristics can be found in STM32 manuals, together with the state-diagrams specifying their usage. The communication between the core and the peripherals is established through registers that can be addressed as memory by the software. Their time validity, their meaning etc.. can be found in the respective manuals. For example the CAN peripheral registers block starts at address 0x4000 6400 and its general structure is the following:

- Master control register at offset 0x00
- TX mailboxes at offset e.g. 0x188
- Identifier filter registers at offset 0x200
- ...

By setting and reading the content of these memory addresses at valid times it is possible, for example, to send and receive CAN packets, to set the communication bandwidth and to filter incoming packets based on their indentifier fields.

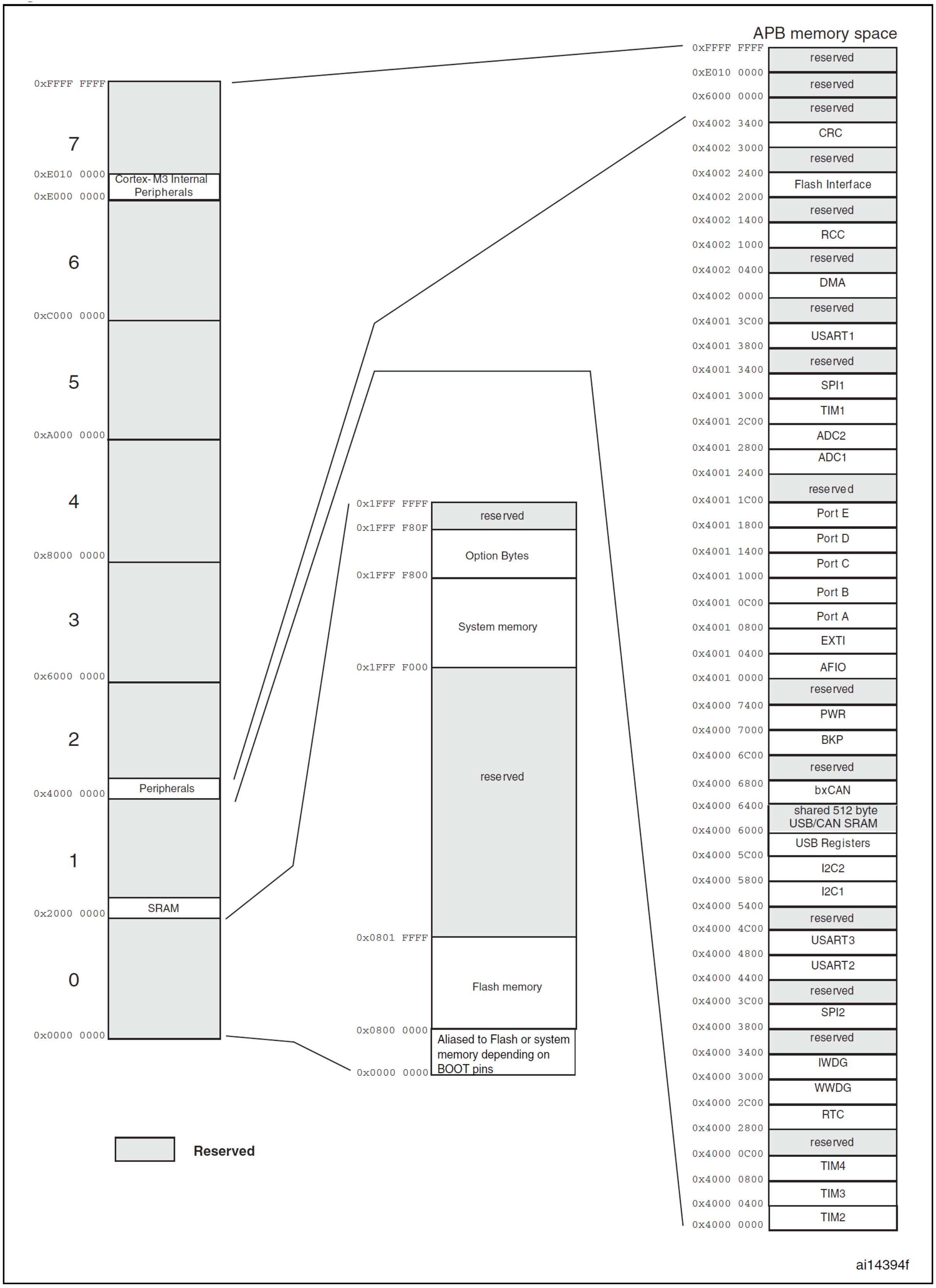


Figure 6: Memory Map of the STM32 Family

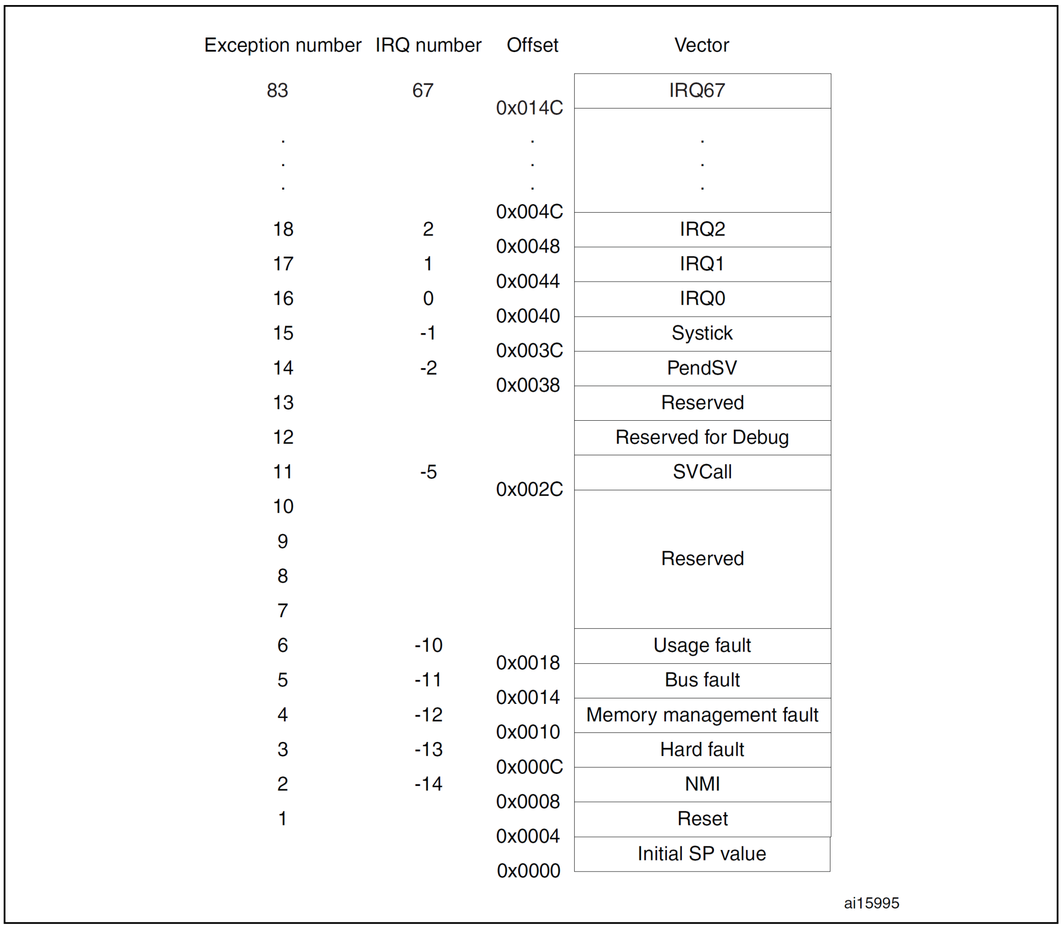


Figure 7: Vector Table Structure for the Cortex-M3 Core

2.2.4 Embedded Bootloaders

All devices of the SMT32 family already have a bootloader, besides JTAG, capable of using a serial peripheral. It is stored in ROM memory ("System Memory" area, Figure 6) and inserted during fabrication. Some pins are reserved on the pinout to be able to select system memory as boot space. The micro used for this project, STM32F103CBT6, only embeds a USART bootloader (unlike other models e.g. STM32F2xx). This fact is the driving force behind the development of a custom CAN bootloader. As noted earlier, having not any explicitly reserved space for a bootloader in the Flash memory, any additional custom BL has to be inserted into common Flash space.

The protocol used above the serial signalling by the embedded bootloaders is specified in [6]. It is a stateless, command based protocol. It embodies usual transport protocol services such as error detection (thorugh checksums) as well as the application-level flows needed to perform bootloading. The host sends commands to the device, which executes commands without knowledge of what it has performed previously. A single command works in this way: the device waits for an incoming byte and for its negation. Depending on the byte, a different command is executed. The command set is reported in Table 1. This protocol has been chosen for our custom bootloader as well for its simplicity and for interoperabilty with existing tools. This choice will be further motivated later.

	Command Code	Description
Get	0x00	Gets the version and the command set supported by the bootloader
Get Version & Read Protection Status	0x01	Gets the bootloader version and the Read Protection status of the Flash memory
Get ID	0x02	Gets the chip ID
Read Memory	0x11	Reads up to 256 bytes of memory starting from an address specified by the host
Go	0x21	Jumps to user application code located in Flash or SRAM memory
Write Memory	0x31	Writes up to 256 bytes to the RAM or Flash memory starting from an address specified by the host
Erase	0x43	Erases from one to all the Flash memory pages
Extended Erase	0x44	Erases from one to all the Flash memory pages (mutually exclusive w.r.t. Erase)
Write Protect	0x63	Enables the write protection for some sectors
Write Unprotect	0x73	Disables the write protection for all Flash memory sectors
Readout Protect	0x82	Enables the read protection
Readout Unprotect	0x92l	Disables the read protection

Table 1: Command Set of the Embedded STM32 Bootloader

2.3 Linking for ARM processors through GNU's `ld`

Linking is usually the concluding step when compiling a program. Its role is to combine several input object files together, relocate their data and tie up their symbol references. The linker produces an executable file, which is obviously platform-dependent. The format adopted by ARM for its Cortex processors' object files is based on the ELF (Executable and Linkable Format) standard. The linker may also produce raw binary files to be loaded in memory at a known address, which is platform-dependent as well.

An object file consist of many sections, among which the notable ones are `.text`, `.data`, `.bss`. The `.text` section contains the assembly instructions (coded into binary) that will be executed. The `.data` contains the declared and initialized variables of the program at known memory addresses, which can be direclty referenced from within the code of the `.text` section. Similarly the `.bss` contains declared but uninitialized variables. Each of the elements in these sections must eventually be given an absolute address in memory. Given this framework, it is the job of the linker to finally resolve the memory addresses used for instance by jumps and to assign memory address in which the program symbols' values will be found.

The linking process in GNU's `ld` is controlled by a linker script, which is a text file describing the layout of the output file as the user or other applications expect it to be. The following code snippet shows a portion of our linker script: it can be noticed that it specifies the origin addresses of different physical memories as well as the mapping of the `.text` section in the Flash memory. The `.data` section is expected to be in RAM but it is loaded in Flash to retain data over resets or power offs.

```
MEMORY {
   RAM (RWX) : ORIGIN = 0x20000000 , LENGTH = 20K
   EXTSRAM (RWX) : ORIGIN = 0x68000000 , LENGTH = 0
   FLASH (RX) : ORIGIN = 0x08000000 , LENGTH = 126K
   EEMUL (RWX) : ORIGIN = 0x08000000+510K, LENGTH = 2K
}
SECTIONS {
   /* vector table and program code goes into FLASH */
   .text : {
   ...
   *(.text .text.*)
} >FLASH
.data : ALIGN (8) {
   ...
   *(.data .data.*)
   . = ALIGN (8);
   *(.ram)
   *(.ramfunc*)
   ...
} >RAM AT>FLASH
```

In a multi-application scenario, for two applications to communicate they must agree on a shared memory address where to find each other. This is the case in our bootloader of the starting address of the loaded application, from which the bootloader writes the application and then on which it jumps on. This address must be the same as the one used in the loaded application linker script, otherwise the first instruction to be executed by the loaded application will not be found there (although all references -within- the loaded application would still be consistent).

3 Design Considerations

Protocol alternatives and reasons behind the choice of the second:

- YModem: YModem is a protocol family for file transfer used between modems. It is a transport protocol which positions between the low communication layers (USART or CAN-bus) and the application logic (such as memory operations). YModem has some advantages, for example, it uses block transmission and CRC error check. We tried to basically use YModem Protocol at the beginning, but finally we gave up, it is based on the following reasons:
  - Most important, YModem is a protocol family, it is not just a single protocol, and there are a large number of mutually incompatible YMODEMs. But we have to make sure the YModem In boot loader at device side must be coherent to the YModem implemented at PC side.
  - There is no existing coherent YModem implementation at both device side and PC side. And we check through the codes, we think it will be very complex to modify the existing implementation or start from scratch by our own.
  - Since YModem is complex it will cause great influence on overhead of the transmission
  - YModem implements CRC as checksum, but error detection is already provided by CAN, then CRC in Y-Modem is a waste of processing time.

- STM custom boot loader protocol (Section 2.2.4): for the implementation details, we will explain it later. Here we just explain why we choose this protocol as our final solution.
  - It is simple and just contains the most basic functionalities, so it will cause very little overhead which we can neglect.
  - ST's custom bootloader already specifies the application-level interactions needed for a boot loading process. In contrast, using YModem we would have ha to define a custom one.
  - This is STM boot loader protocol; by using this protocol we can make it compatible to the original STM boot loader (PC side), so it will be convenient for users to use.
  - There is already an existing PC side open-source compatible boot loader implementation to start from and possibly customize.

4 Tools and Development Environment

- OS: Linux (Ubuntu 11.10)
- Compilation Toolchain: MentorGraphics Sourcery CodeBench for ARM EABI Lite Edition (based on GNU toolchain)
- Development Environment: Eclipse with CDT
- Board: Provided By Martino Migliavacca (Figure 8)
- Programmer: JTAG, openOCD
- CAN-Bus host adapter: Peak PCAN USB-CAN converter with driver installed on the machine

5 Implementation

5.1 Protocol Flow

The command flow of a standard usage of our bootloader can be seen in Figure 9. These command invocations eventually bring an application to be written in the device and exectued. Each of these commands can be turned off; refer to Section 6 for details.

5.2 Device Side

5.2.1 Architecture

The architecture of the bootloader is shown in Figure 10. We structured it in this way to provide a useful abstraction layer for communication and specific device settings.

The communication abstraction provides a unique way to initialize a communication device, send and receive bytes through it. If a new communication peripheral has to be used with the same bootloader logic, only a change at the lower layer is required. These abstractions are implemented by `cal.c`, `cal.h`. The core abstracted functions in `cal` are:

```
int32_t cal_init(void);
int32_t cal_sendbyte(uint8_t b);
int32_t cal_receivebyte(uint8_t *c, uint32_t timeout);
```

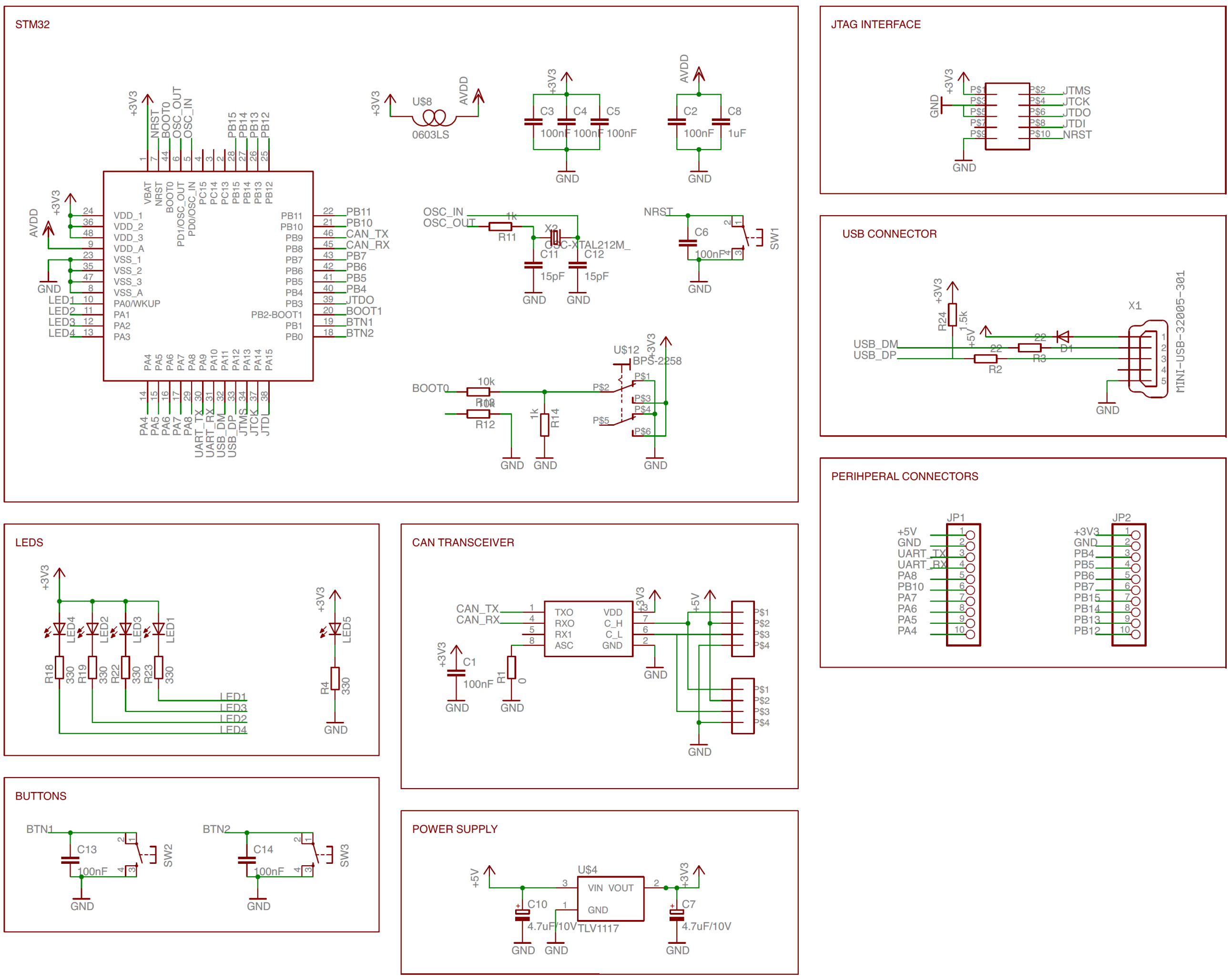


Figure 8: Development Board Schematic

All the other functions related to the abstraction mechanism in `cal.c` are very similar to the above three.

The device settings abstraction provides a unique way for the bootloader to set up the underlying device e.g. clock tree settings and the necessary operations of the device compone-nents e.g. Flash memory controller (excluding communication peripherals, which are handled by `cal.c`). The abstraction layers rely on STM libraries, which have been used to speed up the development.

The protocol commands implementation builds on top of the functions provided by the abstraction layers and occasionally uses some library functions as well.

5.2.2 Library Usage

We decided to use STM libraries to speed up the application development. Among many, three are the main ones: `stm32f10x_can.h`, `stm32f10x_flash.h`, `stm32f10x_usart.h`. They provide useful functions to act on the peripherals registers and set them according to their work-flow. For example, a useful library function to send a byte through CAN is `void CAN_Receive (CAN_TypeDef* CANx, uint8_t FIFONumber, CanRxMsg* RxMessage)` which embodies detailed receive FIFO queues selection and management according to the CAN specification.

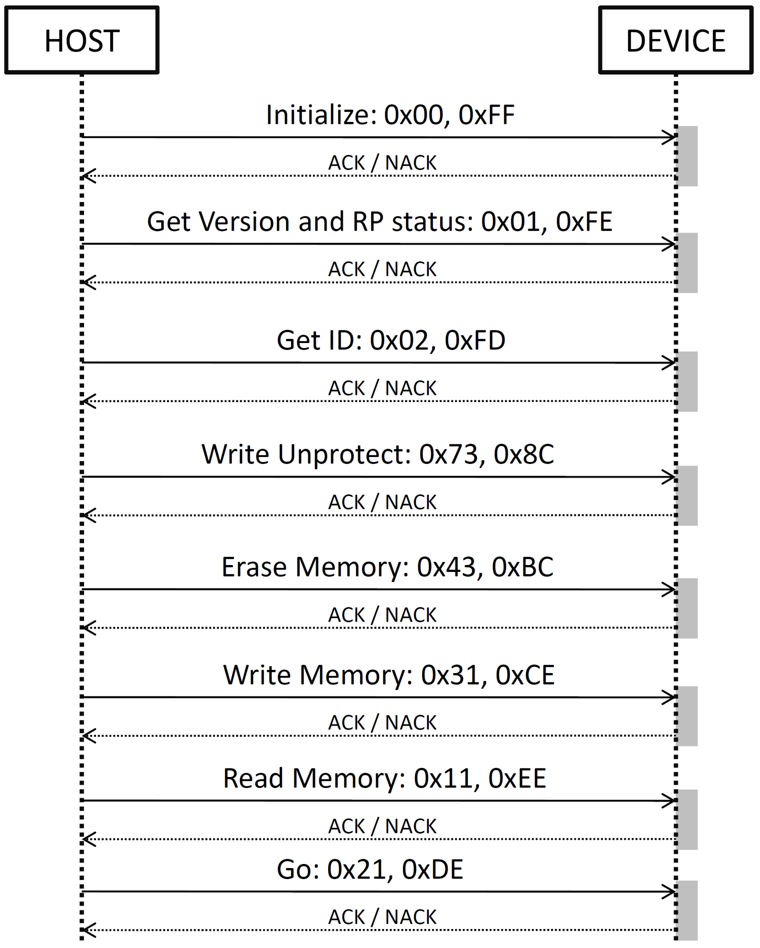


Figure 9: Interactions Between Host and Device During our Bootloading Process

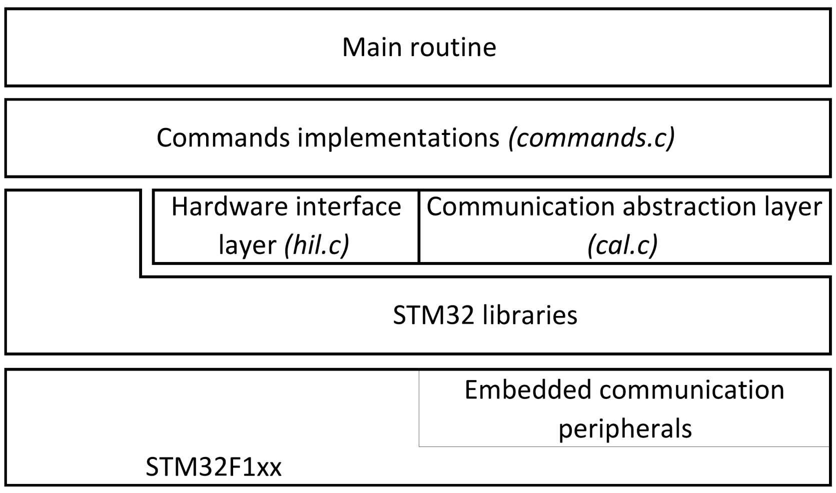

Main routine
Commands implementations *(commands.c)*
Hardware interface layer *(hil.c)*
Communication abstraction layer *(cal.c)*
STM32 libraries
Embedded communication peripherals
STM32F1xx

Figure 10: Bootloader Architecture

5.2.3 Communication Abstraction Layer

We implemented both the logic for USART and for CAN (although a USART bootloader is redundant as it is already embedded by ST during fabrication, see Section 2.2.4). Our actual implementation of this layer is useful to distinguish between USART and CAN usage.

Regarding the specific USART usage: the device main parameters are the baud rate and all the charachteristics of the packet. To initialize the device we set the baud rate to 115200 bit/s and the packet format with 1 stop bit, 8 bit as world length with no parity bit and hardware flow control disabled. We also needed to ensure that the USART signal are mapped to the actual pysical ones unsed on the board, as the microcontroller offers more than one option. For the specific registers that hold this information refer to the code or STM USART manual.

After initialization, the two main operations are send and receive a byte. We used the STM libraries to do the job. This is the code of the receive byte handler:

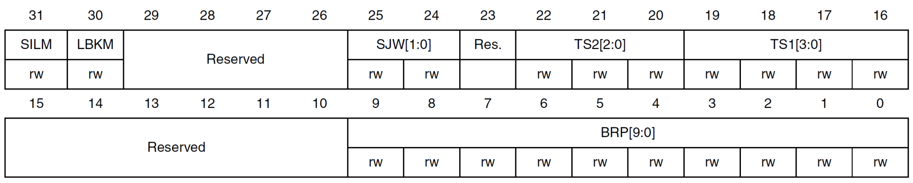
```
while (timeout-- > 0) {
    if ( USART_GetFlagStatus(USART1, USART_FLAG_RXNE) != RESET)  {
        *c = USART1->DR;
        return 0;
    }
}
return -1;
```

In this case the library function is only used to see if the receive data register is not empty. This means that we can acutally proceed to copy meaningful data from the receive data register. If for a while the data register does not become valid, a timeout occurs and the function exits with an error status. The send operation is performed in a very similar fashion by using a check on the transmit data register empty flag.

Regarding the specific CAN usage: the device main paramenters are as well as USART the baud rate and the characteristics of the packet. In CAN we also need to set the indentifiers filters. Unlike USART, the behavior of the CAN controller is driven by a state machine. It is therefore needed to enter initalization mode when initializing the controller. We also needed to ensure that the CAN signals are mapped to the actual pysical pinouts on the board, as the microcontroller offers more than one option. We configured the communication timing (Figure 3) in this way: baud rate of 1 MBit/s (using BRP word), with the bit timing parameters set to:

```
#define CAN_TS1         (0x4)   /* length of PROP_SEG and PHASE_SEG1 */
#define CAN_TS2         (0x2)   /* length of PHASE_SEG2 */
#define CAN_SJW         (0x1)
```

obtaining a sampling point at 6/9 time quanta = 67%. Refer to Figure 11 for the CAN bit timing register details. The identifier filters are zeroed, meaning that any packet is passed on



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
SILM	LBKM	Reserved				SJW[1:0]		Res.	TS2[2:0]			TS1[3:0]			
rw	rw					rw	rw		rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						BRP[9:0]									
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Figure 11: CAN bit timing register (CAN_BTR)

the application. Also, the loop back mode has been very useful for self testing: with this setting the controller internally bridges the CAN output to the CAN input so that packets sent appear as received.

5.2.4 Hardware Interface Layer

We implemented this layer to operate in a uniform way on the device settings. Functions that set up the microcontroller according to our needs are contained here and are called as the first thing when the bootloader is started, for example the clock tree initialization. Others operations that belong to `hil.c` are those to act on the Flash memory. One peculiar example of customized function that fits our needs is the mass erase of the Flash. As our bootloader is placed in common Flash memory, we need to provide a mass erase of Flash memory function that ensures not to erase the bootloader itself. Due to this requirement, we have not used the library function for this operation. We rely on the STM library for other common Flash operations.

5.2.5 Protocol Commands

As mentioned we used ST's embedded bootlaoder protocol also for our bootloader. Although we have made slight modifications:

- Memory Protection: we choose not to implement commands and related checks that handle memory protection and unprotection against unwanted read/writes. The mechanism is complex and is not completely controllable by software. In particular when an unprotection sequence is started, the flash controller automatically starts a globlal erase of the Flash which we cannot control. Unfortunately our bootloader is in the Flash memory itself and we cannot allow it to be erased. This is not a problem for ST's embedded bootloader because it is located in a memory area untouched by the global erase. This is the main reason why we implemented a custom global erase function in the Hardware Interface Layer. Since we do not have memory protection, we cannot implement the read out protection (ROP) check mandated by the protocol in most commands.
- Only Flash memory can be written by the Write Memory Command: in the protocol specification, you can find that the Write Command can also be used to write in RAM and Flash memory option bytes. We decided that for our purposes support for Flash writing is enough.
- Only a global erase can be triggered by the Erase Memory Command

We will now explain a little deeper two important command classes:

- Command Go: this command enables the jump to a specific memory address (which will likely be the starting memory address of the firmware just written by the Write Memory Command). The mechanism that we used to perform the jump is a function pointer. Let us assume that the application base address is A, than the Reset_Handler start address is given by the *content* of the memory cell A+4. In other words, the vector table of the program (Figure 7) begins at address A.
- Write Memory Command and Read Memory Command: most operations on the Flash memory, notably the write operation, are to be done word or half-word wise. In the Write Memory command we receive a stream of bytes from the host (max 256 bytes per command call). We therefore pack them into a word before being written on the Flash. In the following snippet it is possible to see the function that validates the received address and the nested loop that encapsulates word-by-word the received bytes which are stored in `databuffer`.

```
switch (hil_validateaddr(addr)) {
    case 1:  //case FLASH
        for (j=0;j<(number+1);j=j+4) {
                            for (i=0; (i<4 && (j+i)<=number); i++) {
                                    bytes[i]=databuffer[j+i];
                             }
                             FLASH_ProgramWord(addr+j,*(uint32_t*)bytes);
                  }
                  cal_SENDACK();
                  break;
...
}
```

5.2.6 Code Size and Optimization

At the beginning the bootloader code size was more than 20 kbytes (21 memory pages). We optimized the code with the aid of the compiler and linker (flags `-ffunction-sections -fdata-sections` for the compiler and `-gc-section` for the linker in order to keep data and functions in separate sections in the object file so that unused sections can be stripped by the linker) and reached below 12 kbytes (12 memory pages). After deciding that it is enough to choose the communication peripheral to use at compile time, therefore stripping the code that handled the unused peripheral, we were able to reach below 9 kbytes (9 memory pages) when USART is compiled and 11 kbytes (11 memory pages) when code for CAN is compiled.

5.3 Host Side

5.3.1 Reused Template

We started the implementation from an existing program called `stm32ld` by Bogdan Marinescu at `https://github.com/jsnyder/stm32ld`. This program is the host side compatible to STM's embedded bootloader and it can work though USART. We extended it with one command implementation, namely the Read Memory command which was not present. We also added the support for CAN though the library provided by the PCAN converter manufacturer [9]. This reuse saved a lot of development time.

5.3.2 PCAN Library Usage

We used a USB-CAN converter to talk from the host side to the device. The adapter is manufactured by PEAK Systems. We compiled and installed the provided driver on our system. We used the driver from our `stm32ld_cbbl` though a dynamic linkable library also provided by PEAK called `libpcan`. `libpcan` includes the interface to the driver and the command constants.

During testing we faced a problem that some packets sent by the device were not received by the CAN host (through the adapter). Inserting very small delays between each packet delivery on the device side solved the problem. This issue still has to be solved in a proper manner. We are sure that the packets were correctly sent by the device since other listeners on the bus could receive them all.

6 Workflow and Use Case

This section describes how to use the tools that we built. The tools are designed to work in a Unix environment and have been tested under Linux Ubuntu 11.10. The project consists of two programs: `CBBL` which is the device side bootloader that will be flashed into the device memory;

stm32ld_cbbl, which is the host side loader compatible with CBBL . Refer to the tools described in the previous sections to set up the environment. First of all, the user has to flash the compiled binary file of CBBL into the device memory. The file firmware.bin into the CBBL directory is complied for the platform described in the previous secions and linked to address 0x0800 0000. Refer to Makefile, Makefile.common and stm32f1x_md.ld for details and customization of the toolchain settings. This file is compiled to be used with the PCAN-USB adapter. If USART is to be used instead the bootloader must be recompliled changing the conditional compilation switch that can be found in ./src/cal.h.

Once the bootloader is loaded into the device, it will start executing every time a device reset happens. If no button is pressed during the reset, than it is assumed that the user wants to execute a previously loaded application. Thus the bootloader will change the program counter value to the default base address 0x0800 3000 (hardcoded into CBBL) and hand over execution to anything which is written there. There is no support for varying this address except for manually changing it in the sources and recompiling.

By pressing BTN1 during external reset (see board schematic in Figure 8) it is possible to enter in bootloader mode, which assumes that an host will be talking to CBBL . Then CBBL will start listening for the incoming protocol bytes from the selected communication peripheral.

The host side, stm32ld_cbbl, has the following usage:

```
:~$ ./stm32ld_cbbl -help

./stm32ld_cbbl {-usart,-can} {device path e.g. /dev/ttyUSB0} [-write, firmware file]
          [-read download file]] [-noerase] {-defaultbaseaddr,(-custombaseaddr value)}

arguments marked with {} are mandatory unless going for -help
arguments marked with [] are optional
order of the first two arguments should be respected
examples:
./stm32ld_cbbl -usart /dev/ttyUSB0 -write firmwaretowrite.bin -defaultbaseaddr
./stm32ld_cbbl -can /dev/pcanusb0 -custombaseaddr 0x08007000
            -read readflashmemory.bin -write firmwaretowrite.bin
switches description:
-write write specified file into Flash memory from given address
-read read Flash memory into specified file from given address
 neither -write nor -read jump to specified memory address and execute
-defaultbaseaddr use hard-coded base address 0x0800 3000 as the first
            Flash address where the write/read/jump operations will begin
-custombaseaddr use the specified value as the base address
            value must be in the format 0xY
-noerase do not erase the Flash memory before writing
```

The default base address of the host side program can be changed by acting on #define STM32_FLASH_START_ADDRESS in ./stm32ld.h.

For example, let us assume that we are in a multi-application scenario. We want to load an application at 0x0800 3000 (default address). We also want to flash, though CAN, a second custom firmware on the device on a different memory area, read it after it has been flashed and start execution of the firmware just flashed. Let us assume that the first application is 15 kbytes long. Then a suitable address for the second application to avoid overlapping can be 0x0800 7000, which is different from the default.

The first step is to compile CBBL with the following defines in ./src/cal.h:

```
/* Communication peripheral selection ------------------------------------ */
/* Uncomment the intended device -----------------------------------------*/
```

```
//#define USART 1
#define CAN 2
```

We also need to compile the firmwares to be loaded with a modification in the linker script to account for the base memory address. Assume the firmwares are `./stm32ld_cbbl/firstfirmwaretowrite.bin` and `./stm32ld_cbbl/secondfirmwaretowrite.bin` . We will flash `CBBL`'s binary file in the device with a usual programmer. After all is cabled-up, reset the device with the proper button while pressing BTN1. Start the host side loader by:

```
$ ./stm32ld_cbbl -can /dev/pcanusb0 -defaultbaseaddr
          -write firstfirmwaretowrite.bin
```

To Flash the second firmware we must take care of changing the base address and to skip erasing the memory. We reset the device pressing BTN1 and launch the host side by:

```
$ ./stm32ld_cbbl -can /dev/pcanusb0 -custombaseaddr 0x08007000
        -read readflashmemory.bin -write secondfirmwaretowrite.bin -noerase
```

When the procedure ends, two things will happen: the device will start executing the program just flashed from the custom base address; the file `readflashmemory.bin` will contain the contents of the device Flash memory, from the custom base address to the end of the Flash memory space.

7 Future Work

Our boot loader works in a proper way, but still there are some improvements that can be made:

- Library substitution: In our implementation, we use library provided by STM, for example: stm32f10x_can.c. Actually we have cleaned the unused libraries, but to do code optimization, it will be better to also clean the unused functions inside the libraries.
- Optimization of CAN packets usage: now we configure CAN in a way that each packet transmitted only contains 1 byte which obviously is not the best configuration. We can let one packet contain more bytes to improve the transmission efficiency.
- Implementation of remaining commands and memory protection management: the original protocol contains almost 20 commands, we just implement around 10 commands, since it is enough for our boot loader. But if necessary, all commands (includes memory protection/un-protection and so on) can be implemented to extend the usage of boot loader. For the details, which command should be implemented for which reason, please check the public manual.
- Optimize the complier and linker option: now our boot loader is around 10k which will use 3 pages in Flash memory, we hope by changing compiler & linker option we can shrink the boot loader size to fit into two pages since there is not a lot space of Flash memory at all.
- It would be interesting to provide a mechanism for the bootloader to update itself. An hint in this direction is to move the bootloader into SRAM and flashing through it a new bootloader into the base Flash address. This comes with the risk of loosing the bootloader if a shock occurs, as it would reside in volatile memory during the process.

References

[1] Elettronica Open Source. *What a Microcontroller Bootloader Is and How It Works?* http://dev.emcelettronica.com

[2] Robert Bosch GmbH. *CAN Specification, version 2.0.* 1991.

[3] Hartwich F., Bassemir A. *The Configuration of the CAN Bit Timing.* 6th International CAN Conference, Turin

[4] ST Microelectronics. *STM32F103x8-xB Datasheet.*

[5] ST Microelectronics. *RM0008 STM32 Reference Manual.*

[6] ST Microelectronics. *AN3155 USART protocol used in the STM32 bootloader.*

[7] ST Microelectronics. *PM0056 STM32 Cortex-M3 programming manual.*

[8] ST Microelectronics. *PM0075 STM32F10xxx Flash memory microcontrollers programming manual.*

[9] PEAK System. *PCAN-USB - USB to CAN Interface User Manual V2.1.2.* Available at `http://www.peak-system.com/produktcd/Pdf/English/PCAN-USB_UserMan_eng.pdf`

[10] Chamberlain S., Taylor I.L. *Using `ld`, the GNU linker.* Version 2.14

[11] Forsberg C. *XMODEM/YMODEM protocol reference.* 1988