FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Programming of AT32UC3C family of microcontrollers and IGLOO nano FPGAs through Bluetooth
Student:	Bc. Ján Sučan
Supervisor:	Ing. Radek Dobiáš, Ph.D., MBA
Study Programme:	Informatics
Study Branch:	Computer Systems and Networks
Department:	Department of Computer Systems
Validity:	Until the end of winter semester 2019/20

Instructions

An industrial partner needs to find a solution for wireless programming of ETCS balises that have AT32UC3C MCUs and IGLOO nano FPGAs planted. The wireless connection is realized by serial connection over Bluetooth. The control system communicates with the MCU through the Bluetooth.

Study the documentation for AT32UC3C MCUs and IGLOO nano FPGAs, existing means for a Bluetooth communication used by AŽD Praha s.r.o., and the DirectC reference implementation of FPGA ISP from the Microsemi company.

Design methods for erasing, writing and verification
- of the program FLASH memory of the MCU and for executing the program using its USART interface,
- of a bitstream in FPGA array and FlashROM of IGLOO nano by the MCU.

Implement all the designed methods in a firmware for AT32UC3C. Execution of the operations with the MCU and FPGA will be based on commands from the control system.

Test the target functionality of the resulting implementation.

References

Will be provided by the supervisor.

prof. Ing. Pavel Tvrdík, CSc.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague May 31, 2018

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Programming of AT32UC3C family of microcontrollers and IGLOO nano FPGAs through Bluetooth

Bc. Ján Sučan

Department of Computer Systems
Supervisor: Ing. Radek Dobiáš, Ph.D., MBA

May 3, 2019

Acknowledgements

Thanks to:

* My supervisor for his advice, remarks and guidance.
* Maegan Rubino for improving my English and for the language correction of my master's thesis.
* Meili Hollingsworth for having improved my English and for helping me find an English language editor.
* Radek Dobiáš and Jan Konarski from AŽD Praha s.r.o for their patience, understanding, and help.
* My grandmother for having supported me during my studies and for her countless blessings.
* Jozef Masár, my long-time fellow player, for his undying friendship and for the best secondary school years.
* My friends Katka and Peter.

To my dear parents, thanks for their infinite support, for not being angry with me even when the could have been and for encouraging me to walk my own way in life. Everybody would want to have such heroes on their side. I love you.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

I further grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to AŽD Praha s.r.o. company. AŽD Praha s.r.o. is entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on May 3, 2019 .....

Czech Technical University in Prague
Faculty of Information Technology
© 2019 Ján Sučan. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Sučan, Ján. *Programming of AT32UC3C family of microcontrollers and IGLOO nano FPGAs through Bluetooth.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Cieľom tejto diplomovej práce je vytvorenie nástrojov umožňujúcich bezdrôtový zápis, čítanie a verifikáciu programovej pamäte mikrokontrolérov rodiny AT32UC3C, poľa a FlashROM FPGA IGLOO nano, a ich otestovanie. Na základe analýzy existujúcich prostriedkov pre Bluetooth komunikáciu používaných v AŽD Praha s.r.o., literatúry k mikrokontrolérom rodiny AT32UC3C a dokumentácie k referenčnej implementácii DirectC pre ISP FPGA od firmy Microsemi sú navrhnuté a implementované nástroje pre bezdrôtové programovanie. Výsledkom práce sú nástroje pre zápis, čítanie a verifikáciu programovej pamäte mikrokontrolérov rodiny AT32UC3C, poľa a FlashROM FPGA IGLOO nano cez Bluetooth.

Programovacie nástroje sú používané vo firme AŽD Praha s.r.o. ako súčasť programového vybavenia ETCS balíz. Sú prospešné vývojovým pracovníkom, ktorí sa zaoberajú bezdrôtovým programovaním systémov osadených týmito programovateľnými súčiastkami.

V prílohe práce je možné nájsť zdrojové kódy implementácie bezdrôtového programovania.

Kľúčové slová návrh a implementácia bezdrôtového programovania, mikrokontroléry, AT32UC3C, FPGA, IGLOO nano, Bluetooth, komunikačný protokol, C, DirectC, AŽD Praha s.r.o.

Abstract

The aim of this masters's thesis is to develop tools that allows writing, reading and verification of a program memory of microcontrollers of AT32UC3C family, array and FlashROM of IGLOO nano FPGAs, and its testing. Based on analysis of existing means for a Bluetooth communication used by the AŽD Praha s.r.o., literature for AT32UC3C family of microcontrollers, and documentation for the DirectC reference implementation of FPGA ISP from the Microsemi company tools for wireless programming are implemented. The result of the thesis are tools for writing, reading and verification of a program memory of microcontrollers of AT32UC3C family, array and FlashROM of IGLOO nano FPGAs through Bluetooth.

The programming tools are used by AŽD Praha s.r.o. company as a part of software equipment of ETCS balises. They are beneficial to developers who deal with wireless programming of systems that have these programmable devices planted.

Source codes of the programming tool can be found in attachment of this thesis.

Keywords design and implementation of wireless programming, microcontrollers, AT32UC3C, FPGA, IGLOO nano, Bluetooth, communication protocol, C, DirectC, AŽD Praha s.r.o.

Contents

List of Figures

List of Tables

Introduction

Programmable electronic devices have been used in data processing and controlling over the last few decades and this trend continues. The programmable devices are functioning in many different environments. In many of them, the devices can be accessed for programming through a physical connection of programming tools. But there are some environments where the devices are hard to access or they can not be accessed physically for programming tools at all. This can be, for example, for ease of use for a customer or for safety or security reasons. In such cases, the standard means of programming are hard to use or they can't be used at all.

Motivation

Because in many applications of programmable electronic devices they can not be physically accessed by programming tools I decided to develop tools for wireless programming of such devices.

The resulting tools will be beneficial to developers who deal with wireless programming of the systems that have these programmable devices planted.

The industrial partner, AŽD Praha s.r.o. (hereafter referred to as "AŽD"), will use the tools for wireless programming and configuration of ETCS balises.

Objectives

Objective of the theoretical part of this thesis is to study existing means for a Bluetooth communication used by AŽD, literature for AT32UC3C family of microcontrollers, and documentation for the DirectC reference implementation of FPGA ISP from the Microsemi company.

After an agreement with the industrial partner, the requirements for the resulting programming tool will be particularized. The practical part of the thesis deals with design and implementation of firmware for writing, reading,

and verification of a program memory of microcontrollers of the AT32UC3C family, array and FlashROM of IGLOO nano FPGAs through Bluetooth, and also for executing a program from the program memory of the MCU. The resulting implementation is tested for meeting the required functionality.

Structure of the thesis

This thesis is divided into five main chapters. Each chapter except the chapter 1 State-of-the-art contains two specific sections. The first section concerns AT32UC3C MCUs and the second section concerns IGLOO nano FPGAs.

The first chapter describes basic function of ETCS balise, the AŽD infrastructure for remote controlling and monitoring of its balises in the field, existing library functions and communication protocol developed by the company, and hardware I used for the development of the programming tools.

In the second chapter I specify the assignment by agreement with the industrial partner and analyse aspects of the MCU and FPGA programming needed to develop the wireless programming tools.

The third chapter deals with design of two programs for AT32UC3C: one for programming of its own Flash memory and the other for programming connected FPGA.

The fourth chapter is dedicated to the implementation of the wireless programming tools.

In the fifth chapter the resulting tools are tested for meeting the required functionality.

Chapter 1

State-of-the-art

1.1 A Eurobalise

'Balise is an electronic beacon or transponder placed between the rails of a railway as part of an automatic train protection (ATP) system. Transmission device (passive transponder) that can send telegrams to an on-Board subsystem passing over it. A balise which complies with the European Train Control System specification is called a Eurobalise.' [1]

'The system function of balise information transmission system is accomplished by two signal transmission processes, i.e., the tele-powering transmission process where the on-board balise transmission module radiates energy waves to activate the balise to start to work and the up-link signal transmission process where the balise transmits important control information to the train subsequently.' [1]

The telegram transmitted by the balise contains information about the next section of the railway. The information can be the speed limit, the railway track gradient, works on the track, location of the train, etc. [2]

1.2 Programming of balises by AŽD

The life of every balise made by AŽD (hereafter referred to as "balise") is tracked by a central server called PServer. Every balise has its unique identification. The information tracked is e.g. an initial hardware configuration, a version of current firmwares for programmable devices of the balise electronics and a history of the firmware versions that have been programmed before. PServer uses a special communication protocol which can be implemented by a system that wants to save/get the information into/from it. This system can be, for example, a computer at a factory where the balises are made and configured, a handheld computer device or mobile phone used by technicians in the field. The PServer also tracks firmware updates and notifies the systems that the updates are available and where to get the updated firmware

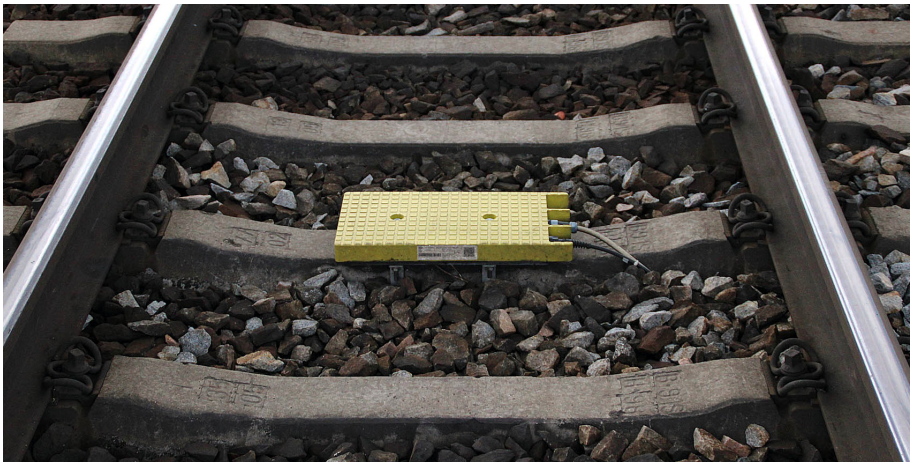

Figure 1.1: A prototype of AŽD balise mounted in a track (Archive of AŽD)

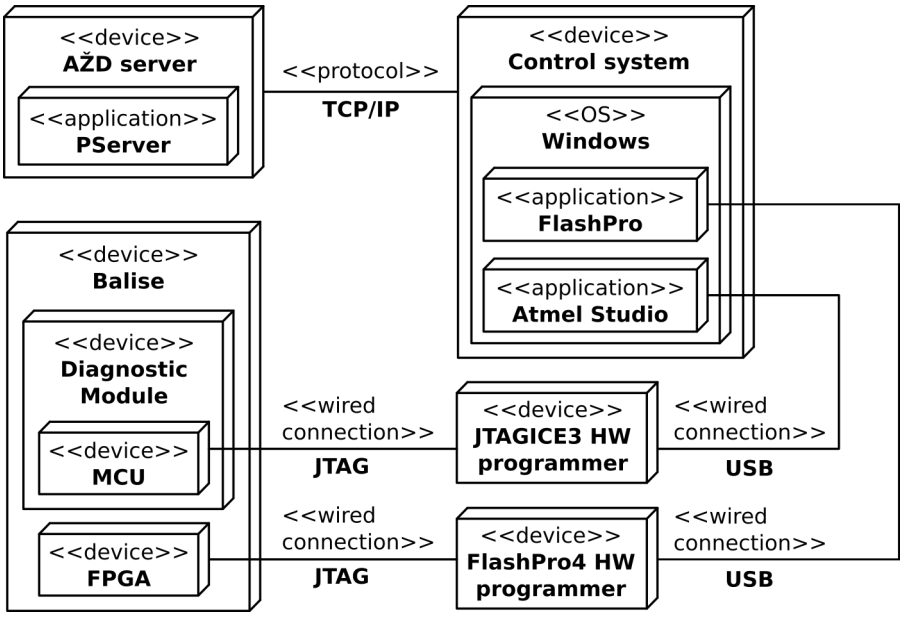

Figure 1.2: Deployment diagram of the balise wired programming

files. The system can then download the updates, load them into a balise and save the information about the updating into the PServer. Currently the firmware is programmed through a wired interface and the aim of this work is to support loading of the firmware wirelessly.

1.3 Diagnostic module

The diagnostic module (DM) is a part of the balise and it is intended for remote controlling and monitoring of the balise. The module is realized as a separate piece of PCB which is inserted onto the main PCB of the balise. It is also used during development of the balise to provide insight to its functioning. It measures logical levels and analog signals of the main electronic circuits of the balise and sends them to the control system which further processes them.

The DM is based on AT32UC3C2512C MCU. The MCU communicates with the control system through the on-board RN-42 Bluetooth module.

1.4 YUP communication protocol

The YUP protocol was designed by AŽD for communication among the DM and control system with intention of using it for future applications and products. It is a point-to-point protocol over a byte-oriented link layer and consists of three layers: data layer, frame layer, and COBS layer. The function of the data layer is to transfer messages from transmitting side to receiving side. If the message is too big it is divided into two or smaller chunks so that each chunk can fit into one frame of the frame layer. The frame layer ensures a reliable transfer of frames between the communicating sides and can handle loss and corruption of frames. Each frame consists of a header section and a data section. Both of these sections are secured against corruption by CRC. The COBS layer with '$' as a delimiter character employs byte stuffing for synchronization of frames and by that it prevents two or more successive '$' characters from occurring on the link layer. This is needed because three successive '$' sent through the Bluetooth module causes the module to enter its configuration mode and thus intercepts data sent through it which treats them as configuration commands. [3] Although it is possible to disable the configuration mode, it is needed to control GPIO pins of the Bluetooth module. One of the GPIO pins is used for HW reset of the MCU.

1.5 Existing firmware for the diagnostic module

There is an existing library of functions and macros for the DM in AŽD called DiagnosticModule. It was developed by me for AŽD in the scope of development of a program for the DM for measuring voltages and logical levels in the balise. The library makes the creation of programs for the DM faster and easier. It contains an implementation of the YUP protocol and other functionalities that can be shared between programs for the DM.

The DiagnosticModule library has been developed using Atmel Studio IDE. The distribution of Atmel Studio comes with the GNU Compiler Collection and Atmel's Advanced Software Framework. 'The ASF provides drivers

and modules developed by Atmel to reduce a development effort of programs for its MCUs. It simplifies the usage of microcontrollers by providing an abstraction to the hardware through drivers and high-value middlewares. ASF is a free and open-source code library designed to be used for evaluation, prototyping, design and production phases.' [4] GCC is sufficient because DM is not a safety-critical part of the balise. If it were, a certified C compiler would have to be used.

The library is divided into six parts: clock, comm, diagnostic_module, flash, protocol, and utils. Each part has its own directory in the source tree of the library and each directory contains C source codes and header files. The names of the directories are equal to the names of the library parts. The source of information for the following subsections, which describe those parts, is [5].

1.5.1 diagnostic_module

This directory contains the main header file `diagnostic_module.h` that includes header files from the other parts of the library. Thus only this one header file needs to be included in a program that uses this library instead of including the many header files. File `diagnostic_module.c` contains function `diagnostic_module_init()`. It takes no arguments and returns no value. It initializes clock source of the MCU, tables for fast CRC-32Q computation and USART interface for communication with the Bluetooth module.

In file `debug.c` there is function `debug_usart_printf()`. It has the same arguments as the standard C `printf()` function but it sends the output characters to the Bluetooth module through USART not using YUP protocol. This function is intended only for debugging purposes and so the `debug.h` is not included in the main header file.

File `firmware_identification.c` contains function `firmware_identification_string()`. It constructs an identification string of a DM application by concatenating a string with the name and the version of the application and placing the resulting string into a provided output buffer.

Header file `parameters.h` is intended to hold values specific for DM for parametrization of the library functionality. It defines a delimiter character used for COBS in YUP protocol.

1.5.2 clock

This directory contains support for initializing the clock source of the MCU and unctions for blocking and non-blocking waiting. Function `pll_use_as_main_clock()` from `pll.c` switches to PLL as the main clock source. It configures the crystal oscillator for use with PLL and then PLL itself. It also handles unlocking the clock registers when writing to them. The function sets the main clock to be 64,512 MHz.

File `wait.h` contains functions and macros for waiting. The code uses a special system register `SYSCNT` as a clock value. The register is 32-bit wide and it is incremented on every tick of a system clock. With the main clock frequency 64,512 MHz, it overflows approximately every 66 seconds. Because of this, it is not possible to wait longer than that. Function `wait_get_deadline()` takes the number of milliseconds as an argument and returns the information about when the time will be up. The information is saved in special data type `deadline_t`. It is a structure containing the value of the `SYSCNT` register at the time when `wait_get_deadline()` was called and the number of milliseconds to wait. Function `wait_has_deadline_expired()` takes a pointer to this data type and determines if the time is up. These two functions implement non-blocking waiting. Function `wait_ms()` implements blocking waiting and uses the previous two functions. Macro `WAIT_ACTIVE_LOOP()` is used in special cases for blocking waiting when calling a function would add an unnecessary time overhead. It is a simple `for` loop with its body being a single `nop` Assembler statement. The number of iterations of the `for` cycle is specified as the macro argument.

1.5.3 comm

This directory contains support for the first two layers of communication between DM and the control system. The lowest layer is USART. The next layer above it implements COBS framing of the data that is sent and received through the Bluetooth. The COBS framing was used to eliminate the sending and receiving of a '`$`' ASCII character. When the Bluetooth module detects a continuous sequence of three '`$`' characters, it switches itself to configuration mode. This captures all other communication except the sequence for exiting the configuration mode. The configuration mode can be disabled but it was decided to leave it enabled in case the control system wants configure the module. Instead, the COBS framing was chosen as a solution.

File `usart.c` contains function `usart_init()` for initializing the USART interface of the MCU for communication with the Bluetooth module and two functions for sending and receiving a byte through USART. `usart_send_byte()` sends one byte. `usart_receive_byte()` receives one byte within an provided time deadline. It indicates the result by its return value.

File `bt.c` contains implementation of the COBS layer. Its interface is similar to `usart.c`. It contains an initialization function `bt_init()` and a two pairs of functions for sending and receiving data. `bt_receive_data()` is used for receiving multiple bytes through Bluetooth within a provided time deadline but without using a COBS. It is used only for debugging purposes. `bt_receive_cobs_data()` is doing the same but it does use a COBS. The next two functions `bt_send_data()` and `bt_send_cobs_data()` send multiple bytes without or with a COBS.

1.5.4 protocol

Implementation of the YUP protocol is located in the `protocol` directory. This implementation uses functions for sending and receiving data from the `bt.c` from the `comm` part of the library.

Header file `return_codes.h` contains definitions of the return codes used by the implementation. Some codes are used internally and the others are used in communication with the control system.

The lowest level of YUP protocol, sending and receiving data frames, is implemented in `frame.c` and `frame.h`. These functions and data structures are not intended to be used by a code outside the library. Data type `frame_t` represents the YUP data frame. Getting and setting data fields of the frame is done by using getter and setter functions. Their names start with `frame_get_`, `frame_is_` and `frame_set_` prefixes. The functions can get, test and set protocol versions, flags, data length, relation ID, sequence number, header checksum, data checksum, and data. `fram_init()` function is used to initialize the layer. `frame_send()` sends a frame and `frame_receive()` tries to receive a frame within a provided time deadline.

Functions for sending and receiving data through YUP protocol are in `data.c`. `data_receive()` receives data and places them to a provided output buffer of a given size. The size is the maximum number of bytes to receive. The function has one output parameter into which it saves the number of bytes actually received. `data_send()` function sends data saved in provided buffer.

1.5.5 utils

Helper functions and macros are placed in this directory.

`SYSREG_SET()` and `SYSREG_GET()` macros wrap `__builtin_mtsr()` and `__builtin_mfsr()` functions from `system_registers.h` are used to set and get a value of the MCU's system registers. Macros `SYSREG_COUNT_GET()`, `SYSREG_COUNT_SET()`, `SYSREG_CPUCR_GET()`, and `SYSREG_CPUCR_SET()` manipulates with the `COUNT` and `CPUSR` system registers.

`byte_buffer.h` contains macros for saving/loading multi-byte integer values to/from byte arrays as little-endian. These macros are `SET_UINTn_FROM_BYTES()` and `GET_UINTn_FROM_BYTES()` where n denotes size of multi-byte value in bits. It can be 8, 16 or 32.

`crc32q.c` implements computing of CRC32-Q checksum. `crc32q_init()` function initializes table for quick computation of CRC. `crc32q()` functions computes checksum of data in provided buffer. It is based on the CRC function from [6] licensed under The FreeBSD Project license.

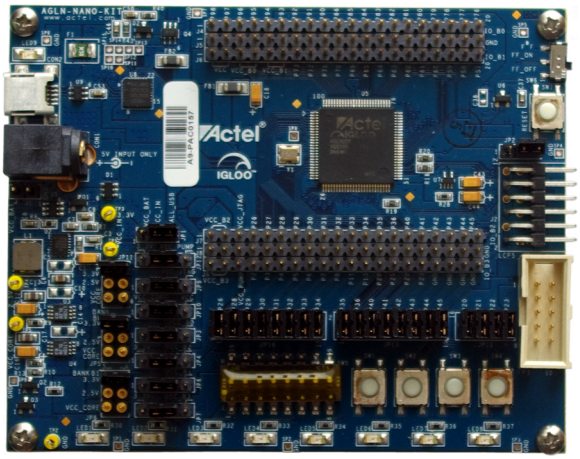

Figure 1.3: IGLOO nano Starter Kit

1.6 Development hardware

For the development of the wireless FPGA programming support, the company has bought an FPGA IGLOO nano Starter Kit. Two modifications have been made to it so that it can be connected to the development piece of the DM. The white connector in the bottom right corner of the figure 1.3 has been added and its signals have been connected to the FPGA by soldering five zero-ohm resistors onto the bottom side of the PCB. [7] The Starter kit can be operated more easily by firmware developers than a prototype of the balise. Also, the impacts of potential damage during a development process would not be so serious. The Starter Kit is based on AGLN250V2 FPGA as opposed to AGLN125 in the balise but this doesn't make any difference in their programming method because both of the FPGA models are members of the same FPGA family.

A piece of the DM used for the firmware development is shown in figure 1.4. It has been modified by AŽD to ease the firmware development process. The white connector, in the top centre of the figure, was added for connecting the DM with the IGLOO nano Starter Kit. The red USB connector with the voltage regulator in the white case makes it possible to power the DM from a USB port of a PC.

An Atmel JTAGICE3 programmer (1.6) is used for programming the MCU and debugging the firmware. Support for this programmer is included directly in the distribution of Atmel Studio IDE. Programming and debugging operations with the MCU's Flash program memory can be executed directly from the GUI of the Atmel Studio.

Figure 1.7 shows a simplified schematic connection of the main parts of the wireless programming solution: the Bluetooth module, the DM's MCU, and FPGA. Only components and signals important for firmware developers

are shown. PD27 and PD28 pins of the MCU have TxD and RxD signals of USART0 as their peripheral functions, respectively. PD30 controls both VJTAG and VPUMP voltages for FPGA programming. The transistors are P-Channel MOSFET so the active logical level of PD30 to switch on the voltages is logical 0.

FlashPro4 hardware programmer together with FlashPro v11.8 software, which is available for download from [8], is used for programming of the FPGA.

Figure 1.4: Diagnostic module used for the firmware development

Figure 1.5: FlashPro4 programmer

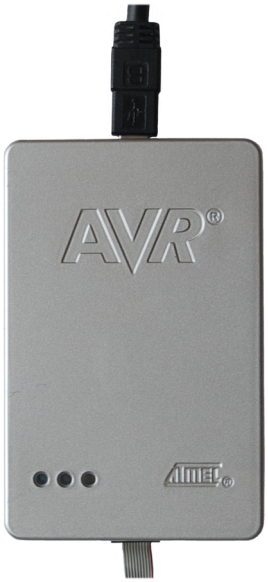

Figure 1.6: Atmel JTAGICE3 programmer

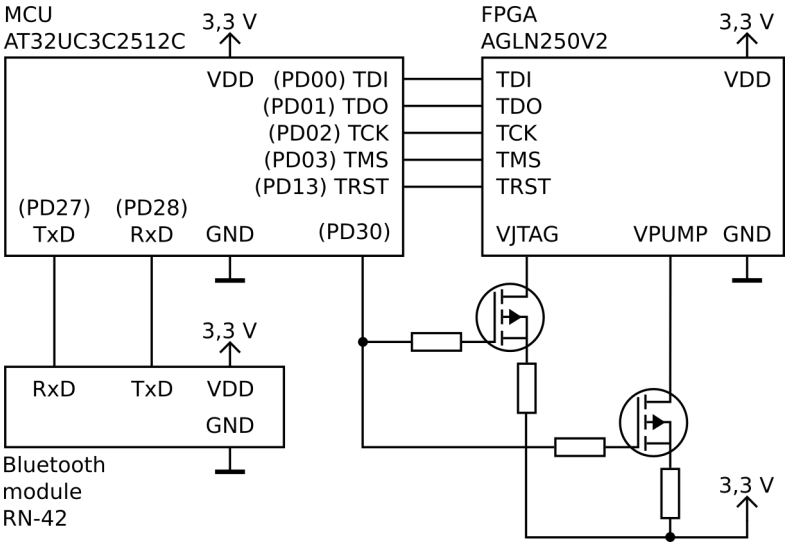

Figure 1.7: Simplified schematic of interconnection of the Bluetooth module, the MCU, and the FPGA

Chapter 2

Analysis

2.1 Programming of AT32UC3C

The source of information for subsections concerning the Flash memory of the MCU is [9]. This information is needed to design a firmware for AT32UC3C for erasing, writing and verifying its Flash program memory and for executing programs from the memory.

2.1.1 Specification of the assignment by agreement with the industrial partner

By agreement with the industrial partner, the requirements for an AT32UC3C MCU programming tool were specified more precisely. The requirements are divided into functional and non-functional. Particularization of functional requirements:

- The tool will support erasing, writing and verification of the MCU's program Flash memory.
- It will support operation to execute a program code from the memory.
- The tool will use the YUP communication protocol developed by AŽD.

Particularization of non-functional requirements:

- Granularity of the operations with the Flash memory will be blocks of multiple bytes.
- The verification will be realized by the control system by reading back content of the memory.
- The tool will not be able to FALSH memory pages where is it saved.
- The model of communication will be Master-Slave. Master will be the control system and Slave will be DM. Slave will respond only when requested by Master.

2.1.2 Flash memory mapping

AT32UC3C has a 32-bit physical address space. The Flash memory is mapped at address 0x80000000. The size of Flash memory differs between MCUs of the AT32UC3C family and ranges from 64 KB to 512 KB.

2.1.3 Erasing, reading, and writing the Flash memory

Erasing, reading and writing of the Flash memory is managed by an on-chip Flash controller. The read operation does not need any direct interaction with the Flash controller. Data from Flash memory can be read by reading words from the corresponding area of the MCU's address space. Erasing and writing can be accomplished only by instructing the Flash controller by writing to its command register FCMD. The smallest unit for erasing and writing the Flash memory is a page. The length of the page is 512 B.

The command register has the following fields:

- KEY (Write Protection Key, bits 31 to 24) should be written with the value 0xA5 to enable execution of the command defined by the CMD field.
- PAGEN (Page Number, bits 23 to 8) holds number of the page that the command will operate with.
- CMD (Command, bits 5 to 0) defines command to execute. Value for the Erase Page command is 2, for Write Page it is 1 and for Clear Page buffer is 3.

Errors are signalized by bits in the Flash Status register. Bit PROGE (Programming Error Status, bit 3) signalizes invalid values in the Flash Command register. Bit LOCKE (Lock Error Status, bit 2) signalizes programming of a locked region of the memory. A value of 1 means that an error has occurred. These bits are clear when the register is read. Readiness of the Flash Controller to accept another command is signalized by bit FRDY (Flash Ready Status, bit 0). If it is 0, the controller is ready to accept another command. The Flash Command register should not be written without checking the FRDY bit first.

Write operation is done through a page buffer. The page buffer is word-addressable and can only be written not read. The program can write to the buffer by writing to the part of address space where Flash memory is mapped. The size of the Page Buffer is equal to size of the memory page. Writing to address A relative to page $P > 0$ is the same as writing to the address A in page 0. Bits of page buffer can only be set from 1 to 0. All bits of of the buffer can be set to 1 by issuing the Clear Page Buffer command. Content of the Page Buffer is written to the Flash memory by issuing the Write Page command.

2.1.4 Protection of the Flash memory pages for a bootloader

The MCU provides two ways of protecting the memory pages from being accidentally modified by the firmware. The first is Region lock bits and the second is BOOTPROT bits. All of these bits are located in Flash General Purpose Fuse register Lo (FGPFRLO). They can be programmed by commands to the Flash controller.

For use of the Region lock bits, the memory is divided into 16 areas. Size of each area is the size of Flash memory divided by 16 (32 KiB for AT32UC3C2512C). Each area has its own Region lock bit.

BOOTPROT bits are intended to protect the continuous area of Flash memory that starts at Flash address 0. BOOTPROT bits form a 3-bit value that sets size of the protected area. They provide a higher level of granularity for the size of the protected area (0, 1, 2, 4, 8, 16, 32 and 64 KiB) than the Region lock bits. The protection of the bootloader area by BOOTPROT bits can be further improved by setting the Security bit that is located also in FGPFRLO. 'If the security bit is set, only an external JTAG Chip Erase can clear BOOTPROT, and thereby allow the pages protected by BOOTPROT to be programmed. No internal commands can alter BOOTPROT or the pages protected by BOOTPROT if the security bit is set.'

The General-purpose fuses can be also manipulated externally through JTAG. `atprogram.exe` command line utility which is shipped with Atmel Studio and supports programming of the fuses through Atmel JTAGICE3 programmer.

2.1.5 Relocation of object code

To build programs that can be correctly executed when placed from a base address other than the MCU's reset vector address, their object code must be relocated to the base address. This is done by the linker. The linker executes a script that defines which code sections will be placed at which addresses in the MCU's address space.

I examined the linker script which Atmel Studio uses for building standard programs for AT32UC3C2512C relocated to the MCUs reset vector address. For standard installation of Atmel Studio the linker script is located at `C:\Program Files (x86)\Atmel\Studio\7.0\toolchain\avr32\avr32-gnu-toolchain\avr32\lib\ldscripts\avr32elf_uc3c2512c.x`.

There are two important lines in the script that control size and base address of the Flash memory. These lines are shown in the short listing 2.1. The line that starts with the `FLASH` keyword defines offset and size of the Flash memory. The line that starts with the `PROVIDE` keyword defines code entry point address.

The GNU `ld` linker has option `-T`. Argument to this option is a path to a linker script that is to be used when linking object files.

Source code 2.1: Default linker script for AT32UC3C2512C

```
...
MEMORY
{
    FLASH (rxai!w) : ORIGIN = 0x80000000, LENGTH = 512K
    ...
}
SECTIONS
{
    /* Read-only sections, merged into text segment: */
    PROVIDE (__executable_start = 0x80000000); . =
       ↪ 0x80000000;
    ...
```

2.2 Programming of IGLOO nano FPGA

2.2.1 Specification of the assignment by agreement with the industrial partner

By agreement with the industrial partner, the requirements for an IGLOO nano FPGA programming tool were specified more precisely. The requirements are divided into functional and non-functional. Particularization of functional requirements:

- The tool will support erasing, writing and verification of the FPGA's array and FlashROM.
- The tool will use YUP communication protocol developed by AŽD.

Particularization of non-functional requirements:

- The verification will be realized by the firmware.
- The tool will not support encryption of array or FlashROM.
- Model of communication will be Master-Slave. Master will be the control system and Slave will be DM. Slave will respond only when requested by Master.
- The VJTAG and VPUMP voltages for FPGA programming will be controlled by a command from the control system.

2.2.2 IGLOO nano FPGA and the FlashROM

IGLOO nano FPGAs are flash-based FPGAs. 'In flash-based FPGAs, nonvolatile memory cells hold the configuration pattern right on the chip, and even if power is removed the contents of the flash cells stay intact. Thus when the system restarts, the FPGAs power up in microseconds, saving time and allowing the system to recover quickly from a power failure or a restart.' [10] To program the FPGA array means to write configuration pattern of the logical blocks and their interconnection to the flash-based memory cells.

The FlashROM is a non-volatile memory integrated in IGLOO nano FPGA. Its capacity is 1 kbit and it can be used to store user defined application-specific information. The balise will use it to store the data of the telegram. 'The FlashROM can be programmed via the JTAG programming interface, and its contents can be read back either through the JTAG programming interface or via direct FPGA core addressing. Note that the FlashROM can only be programmed from the JTAG interface and cannot be programmed from the internal logic array.' [11]

2.2.3 Microsemi DirectC

DirectC is a reference implementation of FPGA ISP from the Microsemi company. It supports 'AGL, AFS, A3PL, A3PEL, A3P/E, A2F, M2S, M2GL, RTG4, and MPF families.' [12] The documentation [12] is available in the archive that can be downloaded after a registration. It contains a brief description of the source codes, information about modifications needed to port it to a new microprocessor, and the .DAT file format for programming data.

Chapter 3

Design

I decided to implement the wireless programming support in two separate programs for the MCU. This division will ease maintenance of these software projects and will make it possible to update the FPGA programming support wirelessly and independently of the MCU's Flash memory programming support.

I named the program for the wireless programming of MCU's Flash memory DMBootloader, and the program for the wireless programming of FPGA DMAppFpgaProg. The prefix DM stands for *Diagnostic Module* and the string *App* denotes an application for DM. The *application* means a relocated program that is intended to be run from a base address other than reset vector of the MCU or in other words, a program intended to be loaded by DMBootloader. DMBootloader itself will be placed by the MCU's reset vector and will be the only part of firmware that needs to be written to the Flash memory by a wired programmer. Every other firmware will be able to be written to the memory by DMBootloader.

3.1 Development software

DMBootloader and DMAppFpgaProg will be based on the DiagnosticModule library that was developed using Atmel Studio. In order to stay consistent with the development tools already in use, DMBootloader and DMAppFpgaProg will be developed using the same IDE. Also, a selection of development tools used for adding the Flash writing support to the library is bound by the tools the library was developed with.

3.2 Communication with the control system

The wireless communication between a program for DM and the control system will be described only at the application layer of a protocol stack shown in

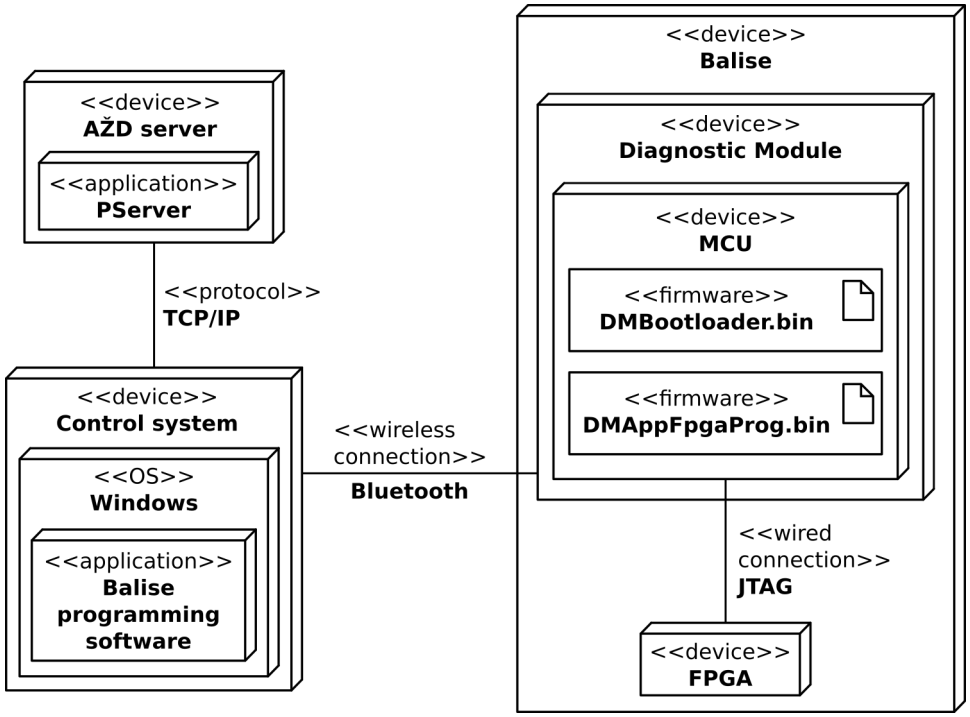

Figure 3.1: Deployment diagram of the balise wireless programming

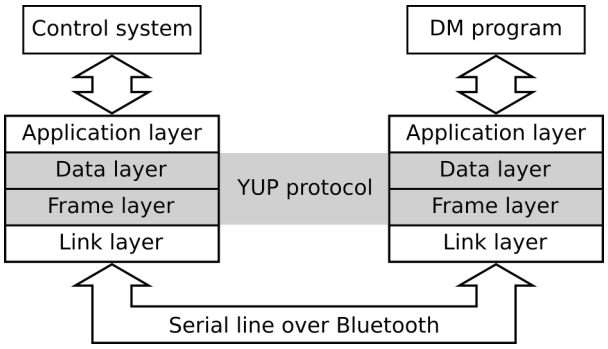

Figure 3.2: Communication layers

the figure 3.2. Communication at lower layers are handled by the YUP protocol that is implemented in the DiagnosticModule library. Both programs must support the same command for getting their identification string. This command will be used by the control system to find out which program is currently running in DM and how it can communicate with it.

3.3 DMBootloader

DMBootloader will be a program for erasing, writing, and reading pages of Flash memory of the MCU of the DM and for executing an application program from a page of the memory. After the execution, DMBootloader will lose control over the MCU. It will gain control again after the execution passes at the reset vector address of the MCU. This can be done by the control system which can enter the configuration mode of the Bluetooth module and control its GPIO pin connected to the reset circuits of the MCU. The execution can also be passed to the reset vector address by jumping there in software but doing this doesn't initialize registers. Because of that, it is expected that DMBootloader will gain control only after a hardware reset of the MCU.

Instead of a private support for DMBootloader, I decided to implement support for the operations with the Flash memory and with an execution of an application program in the DiagnosticModule library. The functionality can be used by application programs to continuously store information in the FLASH memory pages.

3.3.1 Division of the Flash memory address space

DMBootloader will occupy a continuous set of pages in the Flash memory. In order to prevent DMBootloader from overwriting its own pages, I decided to divide address space into two continuous sets of pages. The first set is called physical page space and the second set is called virtual page space. The physical page space contains all pages of the Flash memory. The virtual page space only contains the pages that are not used by DMBootloader and it is a subset of the physical page space. The virtual page space is located at the end of the physical page space. The size of the physical page is equal to the size of the virtual page. Page numbering is relative to a selected page space. For example, if DMBootloader occupies physical pages from 0 to (N - 1), the physical page N is the virtual page 0, the physical page (N + 1) is the virtual page 1, and so on. The division is shown in the figure 3.3. The exact number of pages for DMBootloader will be determined during the implementation phase from the size of its program binary.

3.3.2 Application layer communication protocol

The application layer protocol consists of five commands:

- read page (Read),
- write page (Write),
- erase page (Erase),
- execute firmware (Exec),
- get identification string of the application (GetId).

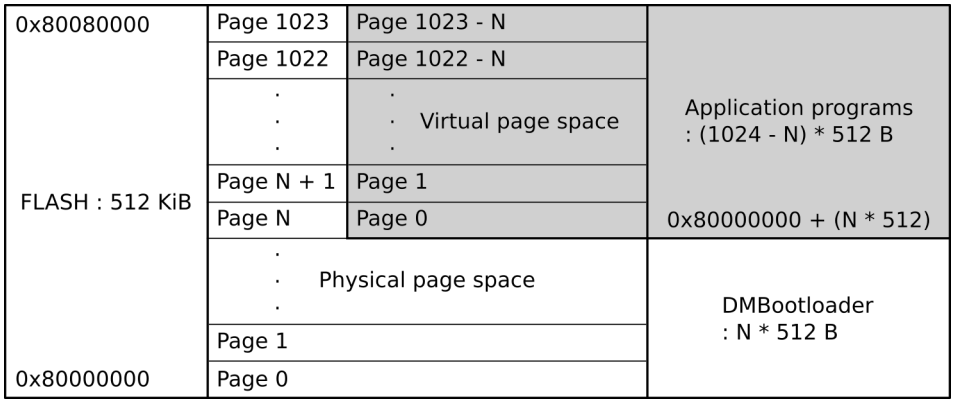

Figure 3.3: Division of the Flash memory address space

As an argument, these commands take the page number of a page within the virtual page space. The format of these commands and replies is shown in attachment C.

3.3.3 Design of DMBootloader

DMBootloader will be based on a loop for receiving and executing commands from the control system. Before entering the loop, the DM will be initialized by a function from DiagnosticModule library. When a command datum is received, it is checked for validity. In case it is not valid, a reply containing the error code is sent to the control system. After the validation, a corresponding function for the Flash memory is called and its result is coded together with possible data from a page (when reading a page) and is sent to the control system. At a level of the source code, the command loop is infinite. It can be exited by receiving a valid command to execute the program from a page. A reply to this valid command always indicates success and must be sent to the control system before the program in the given page is being executed and DMBootloader loses control over the MCU. The main activity diagram of DMBootloader is shown in figure 3.4. The diagrams of the Erase, Write, and Read commands can be found in attachment D.

3.3.4 Flash memory erasing, writing, and reading support for DiagnosticModule library

The interface of the implementation will only work with pages from the virtual page space. This is a partial security measure for protection of the pages occupied by DMBootloader. The interface functions will be:

- `flash_write_virtual_page(page_number, data, data_length)`,
- `flash_read_virtual_page(page_number, data)`,

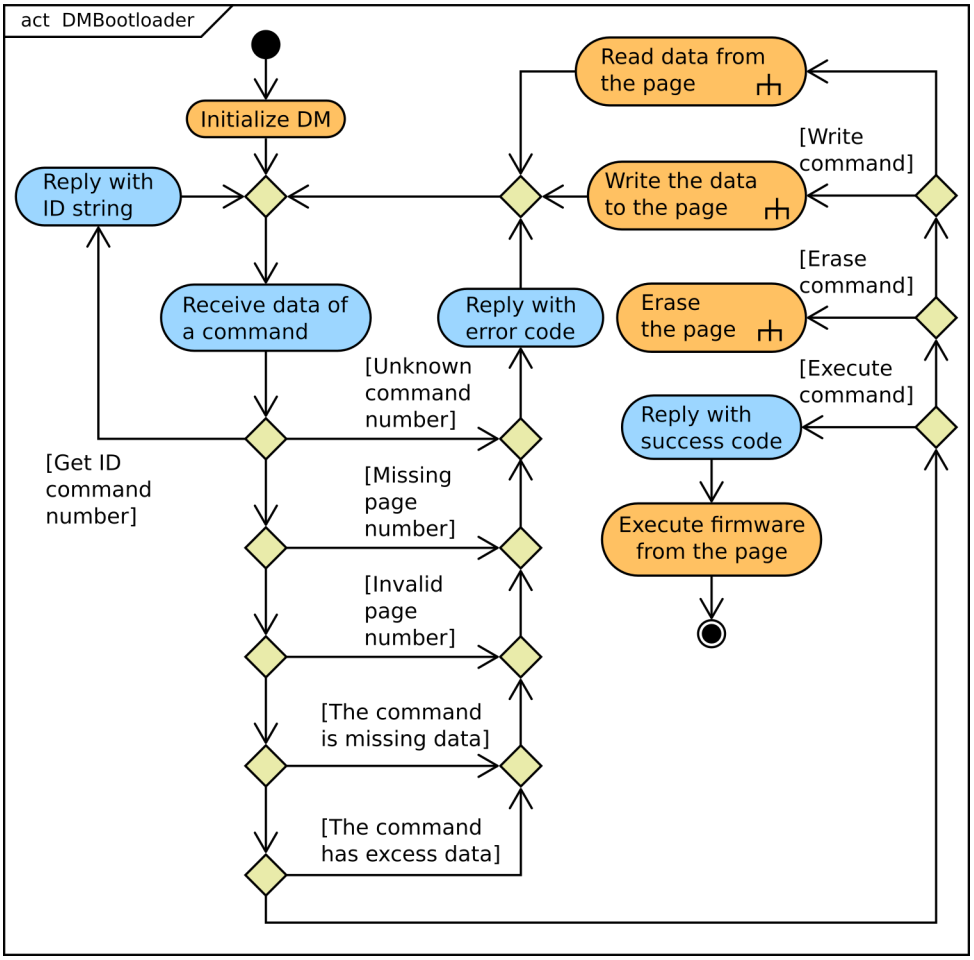

Figure 3.4: The main activity diagram for DMBootloader

- `flash_erase_virtual_page(page_number)`.

The `page_number` is the number of virtual pages to operate with, the `data` is a pointer to a buffer for bytes to be written or read to or from the page, and the `data_length` is the number of bytes in the buffer. The number of bytes written to the page does not have to be equal to the size of the page. In this case, the remaining bytes of the page will be set to `0xFF`.

Reading of the Flash memory is the most simple operation because it can be done without direct interaction with the Flash memory controller. In a program, the bytes can be read through a pointer to array of bytes.

To erase a page, the program must first wait until the controller is ready to be given a command. The command key and the page number is then written to the FCMD register because it is again needed for the controller to finish the command. Finally, the result of the operation is determined from

the error flags in FSR register.

Writing a page is similar to the erase operation with the difference that before writing a command to FCR register, the page buffer must be filled with data to write. Before writing to the page buffer, it must be cleared by a command of the controller.

3.3.5 Support for executing program from a virtual page for DiagnosticModule library

The functionality for executing a program from a given page is related to the Flash memory erasing, writing, and reading support so I also decided to add it to the library. Similarly, to interface the Flash memory support, its interface will allow an execution of the program from a page in the virtual page space. The interface function will be
`exec_application_firmware_at_virtual_page(page_num)`.

3.3.6 Design of an application program

By default, Atmel Studio creates programs that are intended to be placed from the reset vector address. Because DMBootloader is placed from the reset vector address in the Flash memory, the application programs must be placed from other addresses. This can be done by relocating the object code of the program at the link time and by providing a customized linker script to the linker. Relocated programs that use interruptions must adjust reset vectors addresses accordingly.

3.4 DMAppFpgaProg

This application will be a wrapper for the DirectC code and it will serve as an interface between the control system and the FPGA programming support implemented in DirectC. Changes made to the DirectC code from the Microsemi company should be minimal in order to make upgrading easier in the future. Both code that calls DirectC and code that is called by DirectC are considered to be part of the wrapper.

DMAppFpgaProg will be intended to be loaded by DMBootloader. Thus, the application code will need to be relocated. The exact base address will be determined during the implementation phase after the number of pages for DMBootloader becomes known.

3.4.1 Application layer communication protocol

The application layer protocol consists of seven commands:

- execute DirectC operation (Exec),

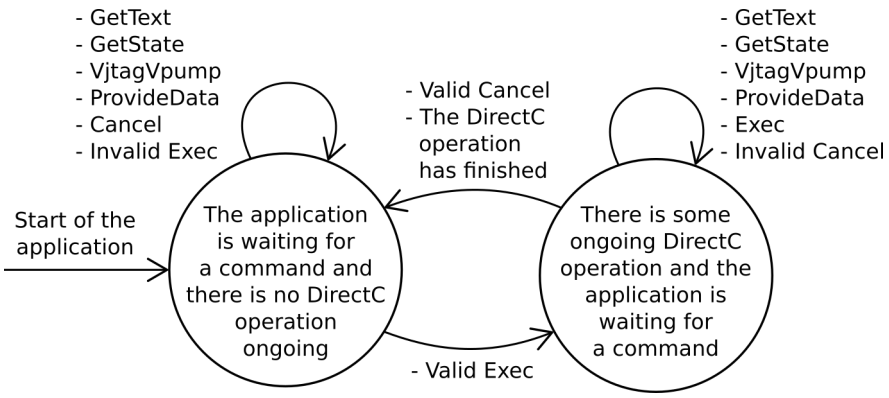

Figure 3.5: Transitions between the application states

- get text output of last executed DirectC operation (GetText),
- get DMAppFpgaProg state (GetState),
- cancel ongoing DirectC operation (Cancel),
- provide data from .DAT file to current DirectC operation (ProvideData),
- switch on/off FPGA JTAG power supply (VjtagVpump),
- get identification string of the application (GetId).

The format of the commands and replies is shown in attachment E. The term *ongoing* means that the operation has been started but not yet finished. It has been paused by passing control to the wrapping code of the application. This will happen when the operation finishes or requests a page from a .DAT file. DMAppFpgaProg will be able to communicate with the control system only when the DirectC code is not being executed.

The commands will be processed by an infinite loop. In the loop, a command datum will be received, a zeroth byte containing command code will be extracted, and a function for processing the command will be called. In case an invalid command code is received, a reply with an error code will be sent to the control system and waiting for the next command will start.

From the communication point of view, the application can be in one of two states:

1. the application is waiting for a command and there is no DirectC operation ongoing,
2. there is some unfinished DirectC operation and the application is waiting for a command.

Transitions between these two states based on a received command is show in figure 3.5.

3.4.2 Interfacing DirectC

The single entry point to the DirectC code is the `dp_top()` function. When called, the wrapping code loses control and execution of the DirectC code begins. Following this, it can call the functions that a user needs to implement, e.g. functions for setting and getting states of JTAG signals or a function for getting the next data page from the .DAT file.

The function for getting the data page needs to send a message to the control system and needs to wait until the page is provided so execution of the DirectC code can continue. The message from the Slave is treated as a reply in the Master-Slave model, which was agreed to in subsection 2.2.1. The Master can only send requests and the Slave can only send a message to the Master when requested. If the message from the Slave for requesting the page is lost before it is received by the Master, the Slave is not allowed to resend the message because it has not been requested by the Master to do so and it violates the Master-Slave model.

A solution to this problem is to enable the Slave to receive requests from the Master after the DirectC function before the next .DAT page is called. This can be done by calling a function that receives requests from the Master (a protocol loop function) again after sending the message to the Master. After doing so, there will be two stack frames for the protocol loop function on the MCU's stack. The first stack frame is for the function called right after the initialization of the application and the second is for the function called by DirectC code for obtaining the next .DAT page.

If the expected message from the client is lost, the Master can contact the Slave to find out what happened. It can inquire about the state of the client and obtain information about the data that the client is waiting for from the Master. This process complies with the Master-Slave model.

The last issue to solve is how to return to the DirectC function in order to continue FPGA programming. A potential solution is simply to return the protocol loop function to its caller, which is the DirectC function, but I didn't choose this option for a clear reason: it would be a non-standard handling of a command in the protocol loop. The loop is supposed to run infinitely and a received command must not cause its termination. Returning from the loop would mislead a reader of the source code and indicate that the loop is being terminated. Instead, I decided to use the `setjmp()`/`longjmp()`.

The `setjmp()`/`longjmp()` mechanism consists of two functions: `setjmp()` and `longjmp()`. The parameter of both of these functions is a variable of type `jmp_buf`. It's a buffer into which `setjmp()` saves the current environment of the program execution that can later be restored by `longjmp()`. After calling `longjmp()` the program, the execution environment is restored from the buffer and the program begins execution in the restored environment. When called directly, the `setjmp()` returns to 0. When 'call' through `longjmp()`, it returns a non-zero value. The value can be specified as an argument to the `longjmp()`

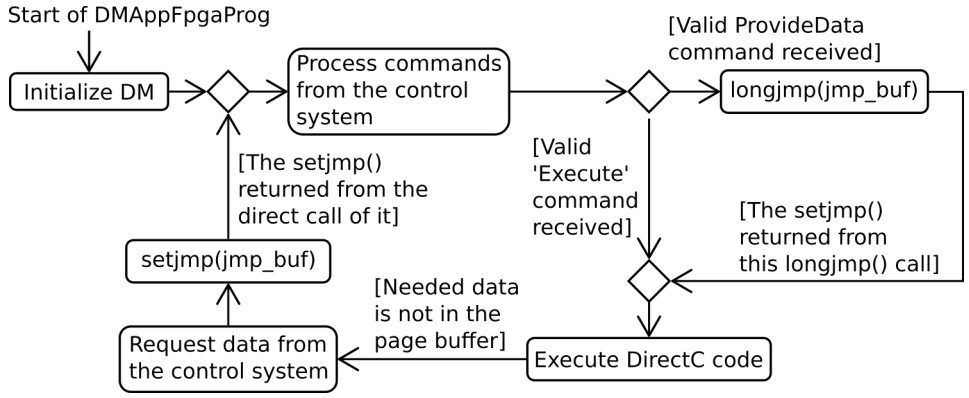

Figure 3.6: Providing data to DirectC operation

call. If a zero value is passed by mistake, the `setjmp()` returns 1 instead. `setjmp()` can be viewed as the C label for a `goto` statement and `longjmp()` can be viewed as a `goto` statement. The main difference is that `setjmp()` saves the execution environment and `longjmp()` restores it.

The negative aspect of using `setjmp()`/`longjmp()`, compared to just simply returning from the protocol loop, is that it consumes more RAM for the buffer into which the execution environment is saved.

The first positive aspect is that by using the `setjmp()`/`longjmp()`, the location of where the execution of the program will continue will be clearer. The hint is the name of the `setjmp()`/`longjmp()` buffer (e.g. `provide_data`). The second positive aspect is that it will be possible to easily cancel ongoing DirectC operations without modifying the source code of DirectC. Without `setjmp()`/`longjmp()`, it will be needed to modify DirectC in order to handle the case when the function for getting the next .DAT page returns. This is because the operation cancel was requested and no .DAT page was provided. With `setjmp()`/`longjmp()`, the DirectC code can be removed from the stack as it has never been called.

The program execution flow for the case of providing data to an ongoing DirectC operation is shown in figure 3.6. Figure 3.7 shows the flow of cancelling the ongoing DirectC operation.

The main activity diagram of DMAppFpgaProg is split into two diagrams. The first, which is in figure 3.8, shows the protocol loop and the processing of commands that concern the use of `setjmp()`/`longjmp()`. The second diagram shows the processing of the remaining commands and can be found in attachment F.

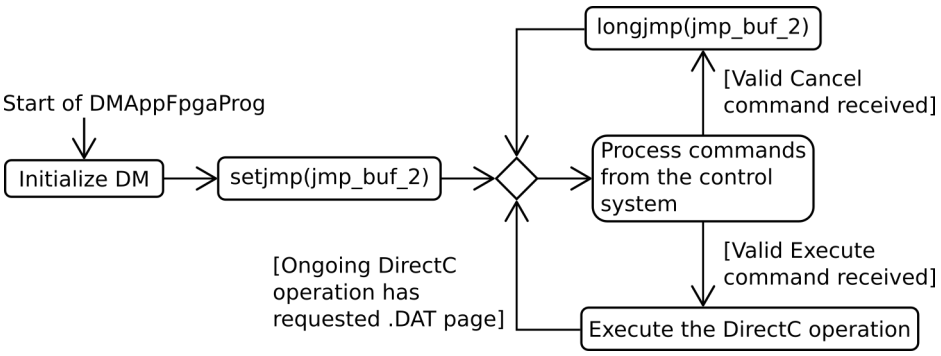

Figure 3.7: Cancelling DirectC operation

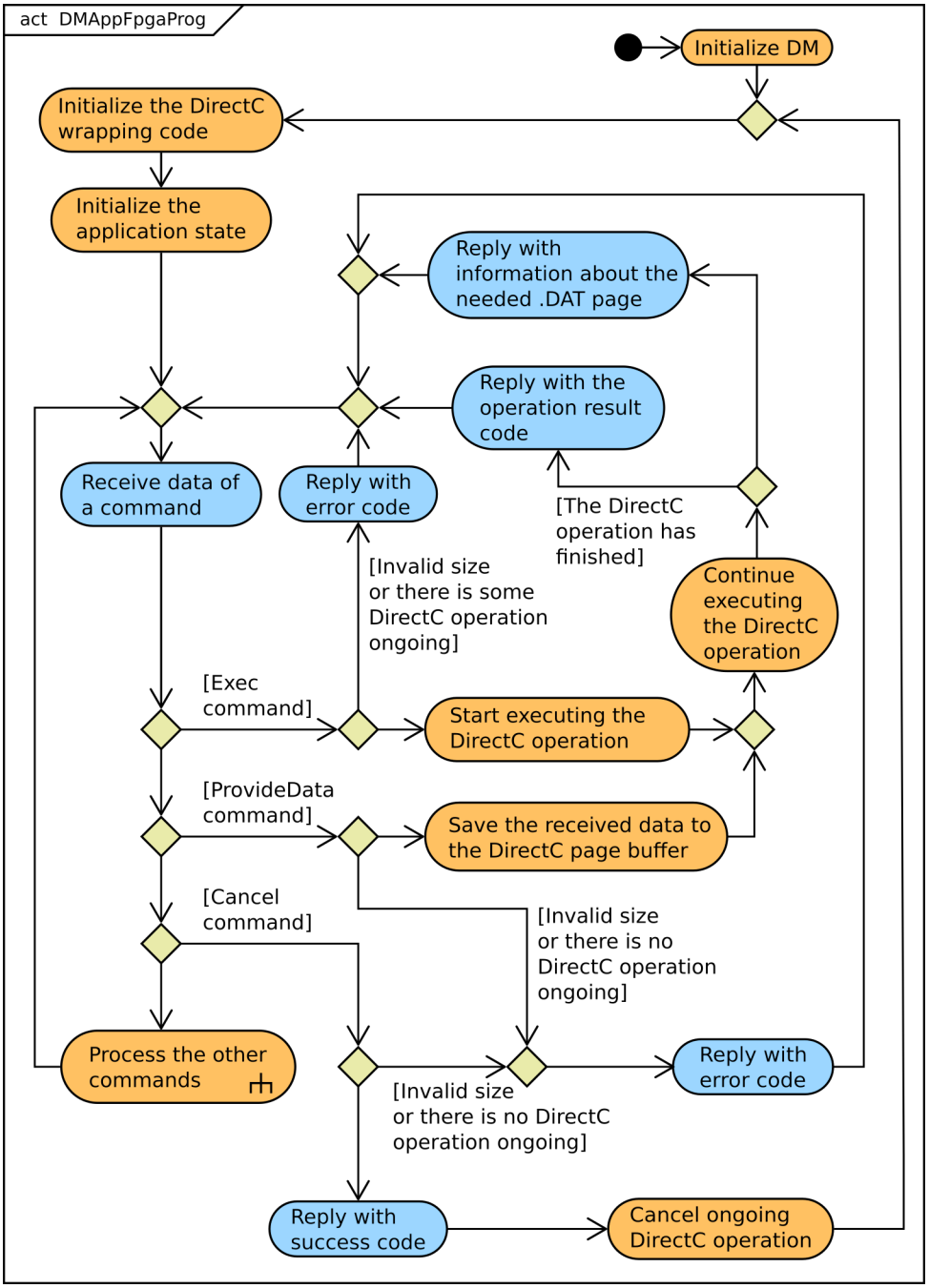

Figure 3.8: Processing commands that concern use of `setjmp()`/`longjmp()`

Chapter 4

Realization

4.1 DMBootloader

4.1.1 Adding Flash writing and reading support to DiagnosticModule library

Source files for the Flash memory manipulation support were placed in a separate directory `flash` in the source tree of the library.

The header file `flash.h` defines macro `FLASH_IS_VALID_VIRT_PAGE_NUMBER()` for checking validity of a number of the virtual page number and provides access to the interface functions implemented in `flash.c`

- `flash_write_virtual_page()`,
- `flash_read_virtual_page()`,
- `flash_erase_virtual_page()`.

for writing, reading and erasing of a virtual page with given number. These functions check if the number of the virtual page provided to them is valid and if it is, they call a corresponding function from set

- `flash_write_page()`,
- `flash_read_page()`,
- `flash_erase_page()`.

These functions work with a page from a physical page space. As an argument, they take a physical page number. The caller function uses macro `FLASH_REAL_PAGE_NUMBER_FROM_VIRT()` to convert a virtual page number to a physical page number. Boundary page numbers for defining the range of the physical and virtual page spaces were added to the DM parametrization header file `parameters.h`.

In `flash.c`, some helper functions are defined for interfacing the Flash memory controller of the MCU:

- `flash_is_error()` checks error flags of the Flash controller if an operation with a locked page was requested or an invalid command was written into the Flash Command Register,
- `flash_wait_until_ready()` blocks until the Flash controller is able to receive the next command,
- `flash_execute_command()` waits until the controller is able to receive the next command. Then it writes the command code and the page number to the Flash Command Register and waits for the operation to complete by calling `flash_wait_until_ready()`. Eventually, it evaluates the result of the operation by calling `flash_is_error()`,
- `flash_clear_page_buffer()` erases the Flash page buffer and thus prepares it for holding a new datum for a memory page,
- `flash_write_page_buffer()` writes content of the page buffer to page with the given number.

The writing to a page in function `flash_write_page()` is done by the following steps

1. the page is erased,
2. the page buffer is cleared,
3. the page buffer is filled with data to write to the page,
4. and `flash_write_page_buffer()` is called.

For reading a page, no contact with the Flash controller is needed. Data can simply be read through a C pointer from addresses in the address space of the MCU where Flash memory is mapped.

4.1.2 Adding support for executing program from a virtual page for DiagnosticModule library

Source files where support for the execution is implemented are `exec.c` and `exec.h`. They are located in the `flash` in the source tree of the library.

`exec_application_firmware_at_virtual_page()` passes control at the zeroth byte of the virtual page and the number of which is given in its argument. After calling the function, it is expected that it will never return to the caller. The function converts the virtual page number to the address in the Flash memory, where the program control will be passed, and provides it to the `exec_application_firmware_at_address()` function, which jumps to that address. The jump is implemented by type casting the `uint32_t` value of the address to a pointer and then to a function that has no parameters. It then returns `void` by calling that function. The implementation is shown in listing 4.1. If there is an application program placed at that address, the C runtime initialization code, included in GCC distribution, clears the stack by initializing the stack pointer. Thus, the application program has the same stack space as if it is run as the non-relocated single application.

Source code 4.1: Function for executing program from a given address

```
void
exec_application_firmware_at_address(uint32_t address)
{
    void (*application_firmware)(void) = (void (*)(void))
       ↪ (address);
    application_firmware();
}
```

4.1.3 Implementation of the application

DMBootloader is implemented in three source files:

- `main.c` contains the main function. It initializes the DM and calls the protocol loop to handle communication with the control system,
- `protocol.c` and `protocol.h` implements the application protocol and processing commands from the control system.

The application uses functions from the DiagnosticModule library. In order to build the application, I had to include a path for the compiler and the library search path as well as the name of the static library archive for the linker. In Atmel Studio, these settings are accessible from the main menu

- Project → DMBootloader properties... → Toolchain → AVR32/GNU C Compiler → Directories
- Project → DMBootloader properties... → Toolchain → AVR32/GNU C Linker → Libraries

For the `protocol.h` header file to be found, I also had to add its directory and include a path for the compiler.

The program starts in function `main()` in `main.c`. `diagnostic_module_init()` from the library is called to initialize the DM and then `protocol_loop()`, from which the program will never return, is called. The only software way to exit the function is by a reception of a valid command for executing the application firmware.

Data of a command are received at the beginning of `protocol_loop()`. The reception through the YUP protocol is provided by the `data_receive()` function from the DiagnosticModule library. The function is internally implemented by actively waiting on the flag which indicate the reception of a byte through USART from the Bluetooth module and the wait time can be limited by caller-defined timeout. It is called repeatedly until data is successfully received but won't receive more bytes than the biggest valid command can have. From the received data, the zeroth byte containing command code is extracted. If the code is invalid, a reply with an error code is sent by calling `protocol_send_reply()` that further calls the library function `data_send()`

to send data to the control system through the YUP protocol. After that, waiting for the next command data begins.

If code of the GetId command is received, the library function `firmware_identification_string()` is called to get a string with identification of the program. The string is sent to the control system by `data_send()` and waiting for the next command data begins.

In case of other valid commands, the program checks whether

1. the number of a virtual page is contained in the received data,
2. the number of a virtual page is valid,
3. the command has at least a minimum number of bytes,
4. the command doesn't have more bytes than the allowed maximum.

If a check fails, an error code is sent and waiting for the next command begins. Otherwise, a virtual page number is extracted from the data and processing continues according the given command. In case of

- Write and Erase commands, the corresponding library function (`flash.h`) is called and code indicating the success or failure of the operation is sent to the control system,
- Read command, the resulting code is sent together with the 512 B of data of the page.
- Exec command, the reply must be sent first because after the execution, DMBootloader will lose control over the MCU. The reply always contains a code of success. Then the program jumps to the zeroth byte of the page requested. It does so by calling `exec_application_firmware_at_virtual_page()` from which it will never return.

4.1.4 Deployment of DMBootloader

DMBootloader is deployed in three steps:

1. Use of the wired programmer to write DMBootloader and the Flash memory of the MCU.
2. Write BOOTPROT fuse bits according to the size of the memory area occupied by DMBootloader.
3. Write the Security bit of the MCU.

After these steps, the only way the bootloader can be modified is to issue the JTAG Chip Erase command. The use of the `atprogram.exe` command line utility is to perform these steps shown in listing 4.2. For the source of information on using the utility, I used its built-in help message. This can be showed by executing it with the `--help` option. The resulting binary image of DMBootloader has 8868 bytes so I chose the size of the protected bootloader area to be 16 KiB (BOOTPROT value 2).

Source code 4.2: Using atprogram.exe to deploy DMBootloader

```
> atprogram.exe -t jtagice3 -i jtag -d at32uc3c2512c
   ↪ program --chiperase --verify --format hex --file src
   ↪ \impl\DMBootloader\DMBootloader\Release\DMBootloader
   ↪ .hex
Firmware check OK
Programming and verification completed successfully.

> atprogram.exe -t jtagice3 -i jtag -d at32uc3c2512c write
   ↪  -fs --values FFF5FFFF
Firmware check OK
Write completed successfully.

> atprogram.exe  -t jtagice3 -i jtag -d at32uc3c2512c
   ↪ secure
Firmware check OK
Secure bit has been set.
```

4.2 DMAppFpgaProg

4.2.1 Porting DirectC to AT32UC3C MCU

When porting DirectC to the MCU, I was following instructions in [12], but there were still some issues that needed to be solved. The following errors were detected by the compiler:

- There was one excess unpaired `#endif` directive in `dpuser.h` that was causing compilation to fail. This directive was commented out.
- The source file `JTAG/gpio_hw_interface.c` was not used and it was causing compilation errors because it used functions not available in the distribution of Atmel Studio.
- The type of `DPCHAR` in `dpuser.h` was redefined from `signed char` to `char`. The signed type was causing compilation errors that pointer targets in the passing argument of `dp_display_text()` differed in signedness. C language standard says that signedness of `char` is implementation defined. It also says that string literals are of type `char[]`. In the compiler implementation used, building DMAppFpgaProg `char` was an unsigned type and it was causing the error.
- Function `int_to_dec_int()` in `dputil.c` contained unused parameter and was causing a compilation error. This was solved by removing the parameter from the function definition and declaration.

The file `dpuser.c` contains functions to be defined by the user of DirectC. I didn't want to implement the functions directly in that DirectC source file. Instead, the required functionality was implemented in the wrapper source file `directc_wrapper.c`. The functions from `dpuser.c` call their correspond-

Source code 4.3: Redirection of the DirectC functions to the wrapper code

```
DPUCHAR jtag_inp(void) {
  return dcw_jtag_inp();
}

void jtag_outp(DPUCHAR outdata) {
  dcw_jtag_outp(outdata);
}

void dp_delay(DPULONG microseconds) {
  dcw_delay(microseconds);
}

void dp_report_progress(DPUCHAR value) {
  dcw_dp_report_progress(value);
}

void dp_display_text(DPCHAR *text) {
  dcw_dp_display_text(text);
}

void dp_display_value(DPULONG value,DPUINT descriptive) {
  dcw_dp_display_value(value, descriptive);
}

void dp_display_array(DPUCHAR *outbuf,DPUINT bytes,
                      DPUINT descriptive) {
  dcw_dp_display_array(outbuf, bytes, descriptive);
}
```

ing functions from `directc_wrapper.c` and return their return values. This redirection is shown in the source code listing 4.3.

The call to `dcw_get_page_data_from_external_storage()` was added in `dp_get_page_data()` in `dpcom.c`. `dp_get_page_data()` is called when a page needed from the .DAT file is not in the DirectC page buffer and the part of its functionality for getting the page must be defined by the user of DirectC.

The usage of the JTAG TRST signal was disabled in the original DirectC source code in `dpuser.h`. Because JTAG connection between DM and FPGA in the balise uses this signal, it was enabled by redefining a preprocessor symbol `TRST` from a zero to a non-zero bit mask (to 0x10).

Declaration of C preprocessor symbols `ENABLE_G4_SUPPORT` and `ENABLE_G5_SUPPORT` in `dpuser.h` was commented out to disable parts of the code with support for unused models of FPGAs. The declaration of symbols `DISABLE_CORE_SPECIFIC_ACTIONS` and `DISABLE_FROM_SPECIFIC_ACTIONS` in `G3Algo/dpG3alg.h` was also commented out to allow for erasing, writing and verification of FPGA array and FlashROM.

In function `dp_init_com_vars()` from `dpcom.c`, which is called before execution of every DirectC operation, initialization of global variables

`current_block_address`, `current_var_ID` and `image_size` was added. The global variables were initialized only in their definitions but this was causing a problem at runtime. Without the variables being initialized before every DirectC operation, the DirectC could request .DAT page of an invalid size because of the last values used by the previous DirectC operation (e.g. more bytes than .DAT file for the current operation has).

4.2.2 Implementation of the application

The program starts in function `main()` in `main.c`. The `main()` calls `diagnostic_module_init()` from DiagnosticModule library to initialize DM, `fpga_init()` to initialize I/O for controlling VJTAG and VPUMP voltages for FPGA, and eventually `protocol_loop()`, from which the program never returns back to the `main()` function.

File `fpga.c` contains an initialization function `fpga_init()` and two functions for switching VJTAG and VPUMP voltages on and off: `fpga_power_supply_on()` and `fpga_power_supply_off()`.

The two `setjmp()`/`longjmp()` buffers are defined in `setjmp_buffers.c` as global variables and are declared as `extern` in the `setjmp_buffers.h` header file in order to be accessible by the other parts of the application. `setjmp_buffer_cancel_operation` is for returning to the beginning of `protocol_loop()` after cancelling the ongoing DirectC operation. `setjmp_buffer_provide_data_to_operation` is for returning to the ongoing DirectC operation when a requested page from the .DAT file has been provided by the control system.

`protocol_loop()` saves the program state to `setjmp_buffer_cancel_operation` buffer, initializes DirectC wrapping code and state of the application, and calls `protocol_loop_wait_for_command()` for processing command from the control system.

The initialization function of the DirectC wrapping code `dcw_init()` from `directc_wrapper.c` configures I/O for communication with an FPGA through JTAG. In the source file, there are also other functions and data structures for interfacing DirectC, which are described in the next paragraphs.

`application_state.c` contains a global structured variable `application_state` which holds the state of the application. There are two application states as described in 3.4.1. The variable also contains code of ongoing DirectC operation, offset of a page from .DAT file, and size of the page requested by the operation.

`protocol_loop_wait_for_command()` functions similarly as `protocol_loop()` of DMBootloader. It receives data of a command and calls a corresponding function to process it. After it has been processed, waiting for the data of the next command begins. Checking if the size of the command is valid is done in its processing function.

Source code 4.4: Use of `setjmp()` and `longjmp()` for canceling ongoing DirectC operation

```
void
protocol_loop(void) {
  setjmp(setjmp_buffer_cancel_operation);
  dcw_init();
  application_state_init();
  protocol_loop_wait_for_command();
}

void
protocol_cmd_cancel_current_operation(
                            size_t bytes_received) {
    ...
    protocol_send_reply(PROTOCOL_OK);
    longjmp(setjmp_buffer_cancel_operation, 1);
}
```

`protocol_cmd_get_application_id()` processes GetId command. It calls the library function `firmware_identification_string()` to get a string with an identification of the program.

`protocol_cmd_get_application_status()` sends information from the `application_state` variable to the control system. When a DirectC request for a page from the .DAT file is lost, the control system can issue the GetState command to get information about the page requested so it can deliver the needed data to the DirectC.

`protocol_cmd_fpga_power_supply_ctrl()` switches VJTAG and VPUMP voltages for FPGA based on a value in the command.

`protocol_cmd_cancel_current_operation()` processes command for cancelling the ongoing DirectC operation. It is simply done by calling `setjmp()` with `setjmp_buffer_cancel_operation` buffer as an argument. It effectively means entry to the `protocol_loop()` functions as if it was called for the first time. The parts of the source code showing the `setjmp()` and `longjmp()` calls are shown in listing 4.4.

`protocol_cmd_execute_operation()` executes the DirectC operation. It saves the code of the DirectC operation to be executed to `application_state` and calls `dcw_execute_operation()` with the operation code as an argument. The function might not return in case of the operation is cancelled by calling `longjmp()` to enter the `protocol_loop()`. If the function returns, it means that the operation is finished. The application state changes to indicate that there is no ongoing operation and a reply containing the DirectC return code of the operation is sent to the control system. After calling the `protocol_cmd_execute_operation()`, a reply is always sent to the control system. Either it is a message about that the operation that is finished or it is a request for a page from the .DAT file sent from the

Source code 4.5: The printf-like function that writes the output characters to a buffer

```
void
dcw_print(const char * format, ...) {
  const size_t free_chars =
    DCW_DISPLAY_BUFFER_SIZE - dcw_display_buffer_index;
  if (free_chars > 0) {
    va_list args;
    va_start(args, format);
    char * const buf =
      dcw_display_buffer + dcw_display_buffer_index;
    const size_t chars_written = vsnprintf(buf,
                                           free_chars + 1,
                                           format, args);
    va_end(args);
    if ((chars_written > 0) &&
        (chars_written <= free_chars)) {
      dcw_display_buffer_index += chars_written;
    }
  }
}
```

`dcw_get_page_data_from_external_storage()` function.

The DirectC code prints out text strings with information about progress of the ongoing operation. For unification of the text, output `dcw_print()` function, which is shown in listing 4.5, was implemented. It has the same parameters as the standard C `printf()` function but it adds the output characters to `dcw_display_buffer[]`. The variable `dcw_display_buffer_index` holds the number of characters written to the buffer, excluding the terminating null byte. The `dcw_display_buffer_clear()` clears the buffer by setting the `dcw_display_buffer_index` to 0. The DirectC code calls functions

- `dcw_dp_display_text()`,
- `dcw_dp_display_value()`,
- `dcw_dp_display_array()`,
- `dcw_dp_report_progress()`.

to print out different types of information. All of these functions were made to call `dcw_print()`. Listing 4.5 shows the unification of the text output functionality.

The control system can issue a GetText command to get content from the buffer. The command is processed in `protocol_cmd_get_operation_log()`.

The function `dcw_execute_operation()` clears the buffer for a text output of the DirectC code, sets the DirectC variable `Action_code` to the code of operation to execute, and calls the DirectC entry function `dp_top()`. Its return value is used as the return value of `dcw_execute_operation()`.

Source code 4.6: Unification of the DirectC text output functionality

```
void
dcw_dp_display_text(const char *text) {
  dcw_print(text);
}

void
dcw_dp_display_value(DPULONG value, DPUINT descriptive) {
  switch (descriptive) {
    case HEX:
      dcw_print("%X", value);
      break;
    case CHR:
      dcw_print("%c", value);
      break;
    default: // DEC, DPULONG
      dcw_print("%lu", value);
      break;
  }
}

void
dcw_dp_display_array(DPUCHAR *value, DPUINT bytes,
                     DPUINT descriptive) {
  if (bytes >= 1) {
    dcw_dp_display_value(value[0], descriptive);
  }
  for (DPUINT i = 1; i < bytes; ++i) {
    dcw_print(", ");
    dcw_dp_display_value(value[i], descriptive);
  }
}

void
dcw_dp_report_progress(DPUCHAR value) {
  dcw_print("Progress: %d", value);
}
```

The DirectC also needs to have functions for platform dependant I/O for JTAG signals and to measure time. `dcw_jtag_inp()` is used to get the value of JTAG input signal TDO. For setting JTAG output signals TDI, TCK, TMS, and TRST, `dcw_jtag_outp()` is used. It receives byte with its bits representing logical levels of the signals and sets the MCU output pins accordingly. Listing 4.7 shows the implementations of these functions. `dcw_delay()` implements blocking waiting. It is based on the `COUNT` system register of the MCU and waits for a given number of microseconds. The function is shown in listing 4.8.

`dcw_get_page_data_from_external_storage()` is called by the DirectC code to get a page of .DAT file. It formulates a request for the control system, sends the request to it, and saves the information about the page to the `application_state` variable so it can be retrieved by the control system in

Source code 4.7: JTAG input and output functions

```
DPUCHAR
dcw_jtag_inp(void) {
  const unsigned port = DCW_TDO / 32U;
  const unsigned pin = DCW_TDO % 32U;
  const uint32_t m = (1 << pin);

  return ((DPUCHAR)
          ((AVR32_GPIO.port[port].pvr & m) >> pin)) << 7;
}

void
dcw_jtag_outp(DPUCHAR outdata) {
  const unsigned directc_output_masks[] =
    {TDI, TCK, TMS, TRST};
  const unsigned gpio_outputs[] =
    {DCW_TDI, DCW_TCK, DCW_TMS, DCW_TRST};

  const unsigned output_count =
    sizeof(gpio_outputs) / sizeof(gpio_outputs[0]);

  for (unsigned i = 0; i < output_count; ++i) {
    const unsigned port = gpio_outputs[i] / 32U;
    const unsigned pin = gpio_outputs[i] % 32U;
    const uint32_t m = (1 << pin);
    const unsigned value =
      outdata & directc_output_masks[i];

    if (value == 0) {
      AVR32_GPIO.port[port].ovrc = m;
    } else {
      AVR32_GPIO.port[port].ovrs = m;
    }
  }
}
```

Source code 4.8: The function for blocking waiting

```
void
dcw_delay(DPULONG microseconds) {
  uint32_t a = 1;
  uint32_t b = 0;

  while (microseconds > 0) {
    if (b < a) {
      a = SYSREG_COUNT_GET;
    } else if ((b - a) >= DCW_COUNT_DIFF_FOR_1_US) {
      --microseconds;
      a = SYSREG_COUNT_GET;
    }
    b = SYSREG_COUNT_GET;
  }
}
```

case of the request is lost. At this place, the state of the program is saved to the `setjmp_buffer_provide_data_to_operation` buffer. If this was a direct call to `setjmp()`, the `protocol_loop_wait_for_command()` is called to receive commands. If the `setjmp()` returned because of a call to the `longjmp()` in `protocol_cmd_deliver_data_to_operation()`, the DirectC variables for information about a page in the page buffer are set and the `dcw_get_page_data_from_external_storage()` returns to its caller. The caller is the DirectC function for getting data from the .DAT file, which now has the needed data in the page buffer and can continue with the execution. The `longjmp()` call takes the size of the page received from the control system as an argument. This is how the information about the size of the page gets to the `dcw_get_page_data_from_external_storage()` function and is then passed to the DirectC code. The parts of the source code showing the use `setjmp()` and `longjmp()` calls for providing data to the ongoing DirectC operation are shown in listing 4.9.

4.2.3 Relocation of the application

For DMBootloader, a Flash memory area of 16 KiB (pages 0 to 31) was selected. Thus, DMAppFpgaProg is relocated to the address 0x80004000, which is the start address of page 32, with the remaining memory area of 496 KiB. The linker script, modified according to the information in subsection 2.1.5, is located in the root directory of the Atmel Studio project for DMAppFpgaProg and its filename is `linker_script.x`. In Atmel Studio, the linker option `-T` can be set in project properties accessible from the main menu Project → DMAppFpgaProg properties... → Toolchain → AVR32/GNU C Linker → Miscellaneous.

Source code 4.9: Use of `setjmp()` and `longjmp()` for canceling ongoing DirectC operation

```
// protocol.c
void
protocol_cmd_deliver_data_to_operation(
  const uint8_t * const buf, size_t bytes_received) {
  ...
  // OK, copy the received data to the DirectC page buffer
  memcpy(page_global_buffer, buf + 1U,
         bytes_received - 1U);
  longjmp(setjmp_buffer_provide_data_to_operation,
          (int) (bytes_received - 1U));
}

// DirectC/dpcom.c
void
dcw_get_page_data_from_external_storage(
  DPULONG image_requested_address,
  DPULONG * const return_bytes,
  DPULONG * const start_page_address,
  DPULONG * const end_page_address) {
  // Send the request for the .DAT page to the Master
  ...
  // Save the information about the needed page to
  // the application state
  ...

  // The return value of setjmp() is:
  //   - 0 when the setjmp() was called by this function
  //   - greater than 0 when the setjmp() was "called"
  //     by the longjmp()
  const int setjmp_ret =
    setjmp(setjmp_buffer_provide_data_to_operation);

  if (!setjmp_ret) {
    protocol_loop_wait_for_command();
  } else {
    // The longjmp() was called, i.e. the page was
    // provided by the Master and the data are already
    // saved in the DirectC page_global_buffer
    *return_bytes = (DPULONG) setjmp_ret;
    *start_page_address = image_requested_address;
    *end_page_address = image_requested_address +
                        *return_bytes - 1;
  }
  // Continue execution of ongoing DirectC opearation
  // with the data it requested
}
```

Chapter 5

Testing

Tests of DMBootloader and DMAppFpgaProg were designed as system gray box tests. The system testing tests a system as a whole for meeting specified requirements. Gray box testing tests functionality at a level of interfaces but uses a partial knowledge of the inside of the system. For testing purposes, a PC will act as the control system.

I decided to create a simple automated test based on Bash shell scripting language. I chose it because I have more experience in scripting in Unix shell environment than in the MS Windows native command shells.

The major problem that required solving was how to communicate with the DM through the YUP and from the shell scripts. For this purpose, I implemented command-line utility `yup-comm`.

The two main principles of the testing are:

- Send a command to the DM, receive a reply, check if the reply is equal to the expected reply.
- Do an operation with a target device (MCU, FPGA) by the FW in the DM, check the expected result by the original HW programmer (Atmel JTAGICE3 for the MCU, Microsemi FlashPro4 for the FPGA).

The designed tests will be used for the testing of DMBootloader and DMAppFpgaProg during their future development in AŽD. Even when the tests are automated in Bash shell, it will not be possible to run them on a Unix-like OS because FlashPro and `atprogram.exe` used for controlling the HW programmers are available only for MS Windows. The test configuration is shown in deployment diagram 5.1.

5.1 The software environment

The tests were conducted in the MS Windows 7 SP1 operating system. A Unix environment with shell and C compiler was provided by MSYS2 version

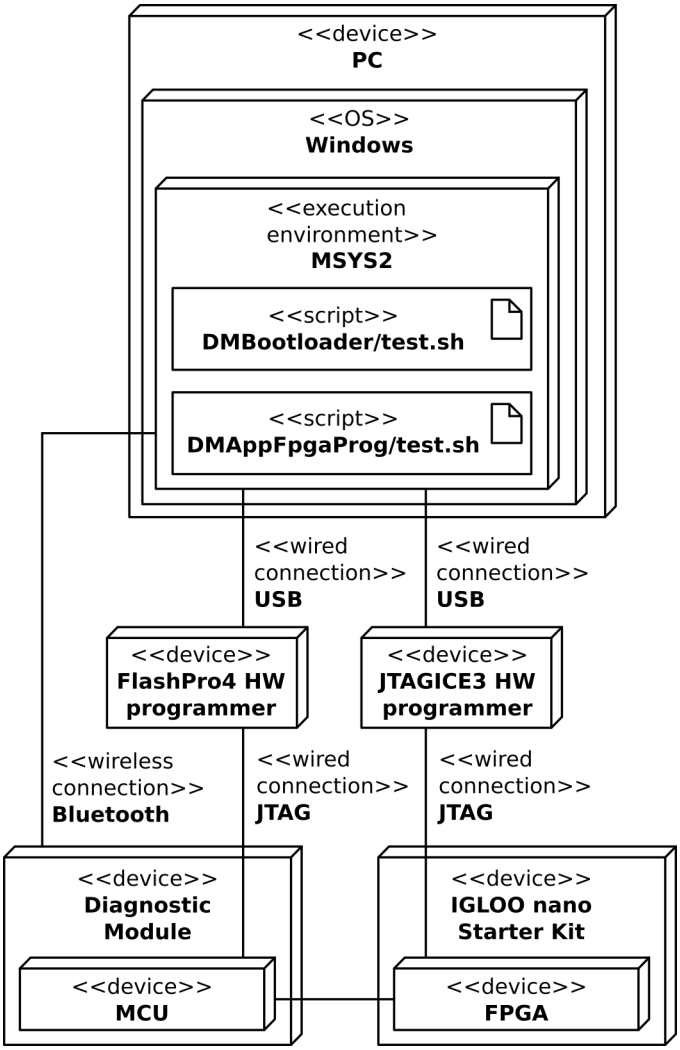

Figure 5.1: Deployment diagram for the testing of the wireless programming support

20161025. 'MSYS2 is a software distro and building platform for Windows. At its core, it's an independent rewrite of MSYS, based on modern Cygwin (POSIX compatibility layer) and MinGW-w64 with the aim of better interoperability with native Windows software. It provides a bash shell, Autotools, revision control systems and the like for building native Windows applications using MinGW-w64 toolchains. It features a package management system to provide easy installation of packages, Pacman.' [13] All examples of command line in this chapter are executed in the bash shell of MSYS2.

5.2 Communicating with DM from Windows 7

For communication between the DM and the operating system, the Bluetooth module of the DM and the PC must be paired. This is done by clicking *Start Menu* → *Devices and Printers* → *Add a device* and following the instructions. After the new device has been added and the OS has finished installation of drivers, a new COM port appears. The COM port number can be found by right-clicking the device in the *Devices and Printers* window, selecting *Properties* and switching to the *Services* tab.

5.3 YUP communication utility

In order to be able to communicate with the DM through the YUP and from the shell scripts, the YUP communication utility was implemented. In doing this, I found out that opening a COM port takes a few seconds. At first I implemented the communication utility in one executable which opened a COM port, sent a command, and finally closed the COM port. Consequently, doing so for every command of the DM testing took a long time. To speed up the testing, the utility was divided into two executables that communicated with each other: `yup-comm-server` and `yup-comm-client`. `yup-comm-server` opens a COM port and receives commands from `yup-comm-client`. `yup-comm-client` sends a single command to `yup-comm-server`, waits until the server receives the reply from DM, and then exits. In this way, a COM port is opened just once and every command for the DM uses this session. This lasts until testing is finished. I measured the speed-up by executing the `1_basic` group of commands, described in subsection 5.7, three times for both the opening and the closing a COM port. There were no retries in opening the COM port, as described in subsection 5.3.2. The average duration of the execution, with opening and closing the COM port for every command, was approximately 24 seconds. The average duration when the COM port was opened and closed only once was approximately 9 seconds. The speed-up is approximately $24/9 = 2.67$.

To implement the YUP protocol, I ported its implementation from the DiagnosticModule library for the DM to Windows 7. To compile the source codes and build the program executables, made by the GCC C compiler and Unix and are already pre-installed in MSYS2, were used.

5.3.1 Porting YUP implementation to Windows 7

The source codes of the YUP implementation in the DiagnosticModule library are in directories `comm`, `protocol` and `utils`. I copied the directories into the YUP communication utility source tree and made the following changes to them:

- Unused header file `utils/system_registers.h` was removed.

- `#include` directives for ASF header file `<avr32/io.h>` were removed.
- `#define` directive for defining the COBS delimiter was added to `comm/bt.c`. In the DiagnosticModule library, the delimiter is defined in `diagnostic_module/parameters.h` but the header file also contains information about page spaces that are not needed for the YUP communication utility.
- `comm/usart.c` and `comm/usart.h` for interfacing USART interface were replaced by `comm/serial_port.c` and `comm/serial_port.h` which contain functions for interfacing COM port in Windows. For a source of information on how to handle a COM port, I used [14].
- In `comm/bt.c` a global variable `bt_serial_port` was added. It holds a handle of COM port to use for sending and receiving bytes, and it is initialized in `bt_init()`.
- Unused functions `bt_module_to_usart_pins()`, `bt_send_data()` a `bt_receive_data()` were removed.
- Functionality for non-blocking waiting and timeouts was removed from all functions that used it to ease the porting. `#include` directives for header file `<clock/wait.h>` was also removed.
- `bt_send_cobs_data_block()` operating with the MCU's USART was sending bytes one by one but when the USART layer was replaced by COM port layer. This turned out to be slow because calls to the Windows API function `WriteFile()` are expensive. To solve this problem, a buffer was defined in the `bt_send_cobs_data_block()` and instead of writing a single bytes to the COM port, they are buffered and sent at once just before the function returns. Thus, there is just one call to `WriteFile()`.
- `#include <>` notation, used in the DiagnosticModule library was modified to `#include ""`.

5.3.2 Server part of YUP communication utility

Server and client parts communicate bidirectionally through the mailslots which is a one-way IPC mechanism in MS Windows OS and can transfer messages up to 424 bytes. For duplex communication, two mailslot channels are used. The source of information about programming with mailslots is [15]. File `mailslot.h` contains functions to create, connect to and close a mailslot, and to write and read messages to and from a mailslot.

The server creates a mailslot for receiving commands from the client and then tries to open a COM port whose name is passed to it as an command-line argument. If the first try to open a COM port fails, the program waits a few seconds and then tries again. In total, four attempts to open the COM port are made. After that, the YUP COBS layer is initialized to use the COM port to send and receive bytes and loops for receiving and processing commands from the client. The server supports three commands:

- EXIT - exits the server. When the program exits, a message to a possible client is sent to inform it that it should also exit.
- PING - is used by the client to test if the server is alive and ready to receive commands.
- CMD,REPLY - reads data from file at path CMD, sends the data through YUP to DM, reads a reply, and saves the reply data to the file at path REPLY.

After a command is processed, a message is sent to the client so it knows that command processing has finished.

5.3.3 Client part of YUP communication utility

The client part is implemented in `yup-comm-client.c`. It takes one argument that is a command for the server part. After checking the argument, a mailslot for receiving a message from the server part is created and connection to the server's mailslot is made. The argument is sent to the server as a null terminated character string. Before exiting, the program waits for an arbitrary message from the server as a confirmation, that either a reply to the command was received or an error occurred on the server side. When passing relative file paths to the server, it is important to realize that the paths are related to the server's current working directory and not to the working directory of the client.

5.4 Types of test files

There are just a few basic types of files, determined by their filenames, that contain data used to test DMBootloader and DMAppFpgaProg. The common types for both of the programs are:

- `N_cmd` are binary files that contain bytes of a command for the program in the DM. The
  `N` is the unsigned integer number. When multiple command files are placed in the same directory, the number determines their order when sending them to the DM. The name can have an arbitrary suffix, e.g. for better explanation of the command.
- `N_reply` are binary files that contain bytes of the expected reply to `N_cmd`.
- `reply` is a binary file that contains a common reply to all the `N_cmd` files in a single directory.

The file type that is only used for testing DMBootloader is `memory_image`. This is a binary file containing an image of the whole or just a part of the Flash memory of the MCU. Its contents are expected data and it's compared to data read by the Atmel JTAGICE3 programmer.

File types that are only used for testing DMAppFpgaProg with suffix:

- .DAT contains data for FPGA programming. This format is needed for Microsemi DirectC.
- .STP contains the same data as corresponding .DAT files but are used with FlashPro.

Unix file paths passed from the MSYS2 environment to the Windows executables must be converted to file paths for Windows. For this purpose, MSYS2 provides `cygpath` a utility that converts paths in both directions.

5.5 Common shell functions

Shell functions shared among main scripts for DMBootloader and DMAppFpgaProg are in `utils.sh` and `yup_comm.sh`. These files are not intended for direct execution. They are included in the main scripts by the source operator (`.`) of the bash shell. `yup_comm.sh` contains wrapper functions for the YUP communication utility executables:

- `yup_comm_start_server()` starts `yup-comm-server` and waits until it is started. This is done by sending the `ping` command to the server by `yup-comm-client`. The client exits either when the server is running and responding or when it is not running. The result of starting the server, which is the return value of the shell function, is indicated by the return value of the client.
- `yup_comm_stop_server()` sends the `exit` command to the server.
- `yup_comm_execute_dm_cmd()` takes Unix paths to a command file and to a file where data of the reply will be stored. It constructs an argument for `yup-comm-client` and runs the client with the argument.

`utils.sh` contains helper functions:

- `info()`, `msg()`, `error()` are used for formatting text messages printed to the console and for their redirection to appropriate output streams (`stdout` or `stderr`). `error()` calls `exit` after printing the error message.
- `get_cmd_num()` takes the name of a command file and echoes its number to a standard output.
- `compare_files()` compares the content of two files and indicates the result in the return value of the function. It uses the `diff` command that is not included in the MSYS2 installation by default and which I had to install in the `diffutils` package.
- `list_commands_an_replies()` creates an input for the `test()` function. It lists command reply files in a directory and outputs lines where each line has two parts. The first part is a path to a command file and the second is a path to the expected reply. The expected reply is either a common reply or, if the directory contains a reply for each command,

Source code 5.1: Running cppcheck.exe for static source code analysis

```
> cd DiagnosticModule/DiagnosticModule/flash/
> /c/Program\ Files/Cppcheck/cppcheck.exe --enable=all --
   ↪ force -I ../../DiagnosticModule *.c *.h
...
> cd ../../../DMBootloader/DMBootloader
> /c/Program\ Files/Cppcheck/cppcheck.exe --enable=all --
   ↪ force -I ../../DiagnosticModule/DiagnosticModule *.c
   ↪  *.h
...
> cd ../../../DMAppFpgaProg/DMAppFpgaProg
> /c/Program\ Files/Cppcheck/cppcheck.exe --enable=all --
   ↪ force -I ../../DiagnosticModule/DiagnosticModule -I
   ↪ DirectC *.c *.h
...
```

a reply special to the command. The lines are sorted by the command numbers.

- `init_tmp_dir()` creates a directory if it doesn't exist or deletes its content if it does exist. It also tests whether the directory is writeable.
- `check_retval()` takes a numeric value as an argument and if it is non-zero, it calls the `error()` function. Otherwise it has no effect.
- `test()` reads a line outputted by `list_commands_an_replies()`. For each line, it calls the `yup_comm_execute_dm_cmd()` saving reply to a file in the temporary directory and compares the reply to the expected one.

5.6 Static code analysis

To detect possible errors in the source codes for DMBootloader and DMAppFpgaProg which may not have been detected by the compiler, I used the static source code analysis tool Cppcheck version 1.87. Usage of the Cppcheck is shown in listing 5.1. The `--enable=all` option enables all checks supported by the Cppcheck, the `--force` forces checking of all configurations in source codes, and then the `-I` options specify paths to find the include files of the DiagnosticModule library and the DirectC so that they can also be included in the analysis.

The analysis was conducted on the source files of the Flash memory support in the DiagnosticModule library, DMBootloader, and DMAppFpgaProg. I didn't analyze the source codes of the DirectC to keep changes minimal.

In the analysis, no errors found. Cppcheck just informed me about some missing include files. I re-ran the checks with `--check-config` options to see what include files were not found. These were header files of the standard C library and the ASF but because Cppcheck does not need the standard library headers to get proper results, I didn't include them in the analysis.

5.7 DMBootloader

Data files and scripts for testing of DMBootloader are placed in their own directory. The tests are fully automated and can be run by executing main test script `test.sh`. The data files are grouped into directories and have an unsigned integer number as a prefix for their names. It determines the order in which the groups must be processed during testing. The following list briefly describes the test groups:

- `1_basic` - basic test of the application protocol implementation.
- `2_cmd_write_all_pages` - writing the whole virtual page space.
- `3_cmd_read_all_pages` - reading the whole virtual page space.
- `4_erase_random_pages` - erasing 16 random pages of the virtual page space written by tests of the `2_cmd_write_all_pages` group.
- `5_erase_even_pages` - erasing even pages of the virtual page space written by tests of the `2_cmd_write_all_pages` group. The Flash memory of the MCU is checked after each of the Erase commands.
- `6_erase_odd_pages` - erasing odd pages of the virtual page space written by tests of the `2_cmd_write_all_pages` group.
- `7_write_DMAppFpgaProg` - programming DMAppFpgaProg into the Flash memory of the MCU.
- `8_exec_DMAppFpgaProg` - execution of DMAppFpgaProg.
- `9_check_DMAppFpgaProg` - checking if DMAppFpgaProg is running by issuing GetId command.

Most of the data files in `1_basic`, `7_write_DMAppFpgaProg`, and `8_exec_DMAppFpgaProg` directories were prepared manually using HxD hex editor. Some reply files were created by issuing a command for the DM, reading back the reply, and manually checking it for correctness. Files in the remaining groups were generated automatically using the helper scripts described in the next subsection. The total number of commands is 3135.

The `1_basic` test was designed with knowledge of the protocol implementation (Greybox testing). For example, because missing a page number of a command is checked only at one place in the program and it is common for all commands, the missing page number check is done only for one command. Similarly, the `4_erase_random_pages` uses knowledge that the processing of the Erase command can not result in erasure of more than one page since there can only be one command written to the Flash Command register. Erasing of odd and even pages is an additional test that checks if the Erase command does not result in erasure of continuous areas of more than one page.

5.7.1 Helper scripts

`atprogram.sh` is a wrapper for the `atprogram.exe` utility which comes in distribution of Atmel Studio. It contains functions for:

- reading a continuous area of the Flash memory of the MCU,
- writing DMBootloader into the memory,
- setting protection of the bootloader by BOOTPROT fuse bits,
- setting the security bit.

Scripts used to generate commands, replies, and memory images are:

- `create_dmbootloader_cmd.sh` generates the headers Write, Read, and Erase commands and writes it to a file. This consists of the command code and page number.
- `generate_cmd_write_read_all_pages.sh` generates commands to write every page of the virtual page space with random data and constructs `memory_image` containing expected content of whole virtual page space that is compared with the content read by Atmel JTAGICE3 programmer.
- `generate_cmd_erase_random_pages.sh` generates commands for erasing 16 random pages of the virtual page space and constructs memory images with expected virtual page space content based on the `memory_image` generated by `generate_cmd_write_read_all_pages.sh`. After each Erase command, the virtual page space is read through the Atmel JTAGICE3 programmer and is compared to the `memory_image` corresponding command. The memory images are cumulative, i.e. the memory image for the command to erase a page have pages of the previous Erase commands that were erased.
- `generate_cmd_erase_even_odd_pages.sh` generates commands for erasing the even and odd pages of the virtual page space and constructs memory images with their expected virtual page space content based on `memory_image` generated by `generate_cmd_write_read_all_pages.sh`.
- `generate_cmd_write_DMAppFpgaProg.sh` generates commands to program DMAppFpgaProg into the MCU.

5.7.2 The main test script

The test of DMBootloader is run by executing the `test.sh` script. The steps are:

1. program DMBootloader into the Flash memory
2. lock the bootloader area pages by BOOTPROT fuses
3. wait 4 seconds until DMBootloader has finished initializing the DM
4. start `yup-comm-server`
5. initialize the directory for temporary files
6. run the test groups
7. stop `yup-comm-server`
8. set the Security bit of the MCU

Results of `3_cmd_read_all_pages`, `4_erase_even_pages`, and `5_erase_odd_pages` groups are checked also by Atmel JTAGICE3 programmer.

5.7.3 Results of the testing

Table 5.1 shows the final results of the testing of DMBootloader. One test iteration, consisting of all the test groups, takes 40 minutes. The time was measured by running the `time ./test.sh` command and is rounded to the nearest minute. The total number of commands sent to DMBootloader during one test iteration is 5119 because some commands are sent multiple times (writing memory with random data for `4_erase_random_pages`, `5_erase_even_pages`, and `5_erase_even_pages` groups).

Group	*Result*	*Commands sent*
`1_basic`	Success	10
`2_cmd_write_all_pages`	Success	992
`3_cmd_read_all_pages`	Success	992
`4_erase_random_pages`	Success	16
`5_erase_even_pages`	Success	$992 + 496 = 1488$
`6_erase_odd_pages`	Success	$992 + 496 = 1488$
`7_write_DMAppFpgaProg`	Success	131
`8_exec_DMAppFpgaProg`	Success	1
`9_check_DMAppFpgaProg`	Success	1

Table 5.1: DMBootloader test results

5.8 DMAppFpgaProg

The layout of files and directories for testing DMAppFpgaProg is similar to the one for DMBootloader. The test assumes that DMBootloader test has already passed and either DMBootloader or DMAppFpgaProg is currently running. The test groups are:

- `1_execute_DMAppFpgaProg` - the main test script expects that both DMBootloader and DMAppFpgaProg are written into the Flash memory and either one of them is running. Before executing any test group, the script sends a GetId command to the DM and checks the answer. If DMBootloader is running, it sends the Exec command. If the command is successful, the script assumes that DMAppFpgaProg has been executed. In the command counts in this chapter, these two commands are not included. They are useful during development of the tests when the DM is often restarted and when DMBootloader is gaining control.
- `2_basic` - basic test of the application protocol.

- `3_write_verify_erase_array_from` - tests of writing, verification and erasing of FPGA array and FlashROM using .DAT and .STP files containing data for array and FlashROM.
- `4_write_verify_erase_from` - tests of writing, verification and erasing of FPGA array using .DAT and .STP files containing data only for the array.
- `5_write_verify_erase_array` - tests of writing, verification and erasing of FPGA FlashROM using .DAT and .STP files containing data only for the FlashROM.

The .DAT and .STP files were provided by AŽD and are located in the `fpga_data` directory. Data files in the `1_execute_DMAppFpgaProg` and `2_basic` directories were prepared manually. Other files were created using the helper scripts. The total number of commands is 5322.

5.8.1 Helper scripts

`flashpro.sh` is a wrapper for `flashpro.exe`. It contains only one function for executing an operation with FPGA by the FlashPro4 HW programmer. This function creates a new FlashPro project in a temporary directory. Following this, a TCL script to execute is given action, runs `flashpro.exe` with the script, and checks the result of the operation by examining a log created in the temporary project directory.

When operating in paging mode, DirectC generates many .DAT page requests. To automatize a creation of commands and replies for a DirectC operation, the script `generate_page_cmds.sh` was created. It takes a path to a directory and expects to find the `1_cmd` command there which is the Exec command for DMAppFpgaProg. It sends the command to the DM, reads a reply, extracts a page offset and size from the reply, constructs `2_cmd` containing the page data requested, and sends the newly created command to DM to get the next reply for `3_cmd` to be constructed. This loop continues until the size of the reply is not 9 B which indicates that the operation has finished. The last reply is checked manually if it is the one expected.

5.8.2 The main test script

Test of DMAppFpgaProg is run by executing `test.sh` script. The steps are:

1. start `yup-comm-server`
2. initialize directory for temporary files
3. run the test groups
4. stop `yup-comm-server`

Test group `3_write_verify_erase_array_from` uses this algorithm for array and FlashROM operations:

1. write FPGA by DMAppFpgaProg
2. verify by DMAppFpgaProg
3. verify by FlashPro and FlashPro4 HW programmer
4. erase FPGA by DMAppFpgaProg
5. verify by DMAppFpgaProg
6. verify by FlashPro and FlashPro4 HW programmer

The test group `4_write_verify_erase_from` uses a similar algorithm but it uses an array. FlashROM data is saved in independent files and treats the FPGA array content as an invariant in order to be sure that operations with FlashROM does not affect the array:

1. write array by DMAppFpgaProg
2. erase FlashROM by DMAppFpgaProg
3. verify array by FlashPro
4. verify FlashROM by FlashPro
5. write FlashROM by DMAppFpgaProg
6. verify array by DMAppFpgaProg
7. verify FlashROM by DMAppFpgaProg
8. verify array by FlashPro
9. verify FlashROM by FlashPro
10. erase FlashROM by DMAppFpgaProg
11. verify array by DMAppFpgaProg
12. verify FlashROM by DMAppFpgaProg
13. verify array by FlashPro
14. verify FlashROM by FlashPro

Test group `5_write_verify_erase_array` is analogous to `4_write_verify_erase_from` except the FlashROM content is the invariant.

Results of the programming operations with FPGA cannot be seen in the same way as they are for the MCU because the programmed content of FPGA cannot be read out. It can just be compared to expected data. FlashROM content is an exception but I decided to treat it as restrictively as the array. This feature of FPGA is especially significant when checking erasure of FPGA. It has four steps:

1. write FPGA with known content
2. verify if the content is as expected
3. erase FPGA
4. verify if the content is *not* as expected, i.e. the FPGA is programmed with a different design

Due to the fact that only one DM and FlashPro4 can be connected to the Starter Kit at a time, manual intervention of test operator is needed during testing. The other reason for the presence of the operator is that FlashPro4

doesn't supply VJTAG voltage but DM can. To test VJTAG and VPUMP switching by the DM, `2_basic` tests are conducted with the voltages switched off in the Starter Kit. For the next tests, the voltages are switched on a DM does not control them. Function `operator_msg()` is called to print instructions for the test operator and to wait for key press to continue the testing.

5.8.3 Results of the tests

During testing of DMAppFpgaProg, an error in function `dp_get_data_block_element_address()` from `dpcom.c` was found. The code in the function tests whether the required data is in the page buffer or if the data must be fetched from an external storage. In the set of conditions, for the case when the needed data is in the buffer, there was a condition that tested whether the buffer contained at least `MIN_VALID_BYTES_IN_PAGE` from the requested address. For certain combinations of the .DAT file sizes and usages of page buffering, DirectC code could have requested a page that was valid but contained less than `MIN_VALID_BYTES_IN_PAGE` bytes. The DirectC operation returned an error code indicating an invalid .DAT file. The error was fixed and the tests were re-run.

Table 5.2 shows the final results of DMAppFpgaProg testing. One test iteration takes approximately 82 minutes. The time was measured by running the `time ./test.sh` command and rounded to the nearest minute. Because the testing is not fully automated, as described in subsection 5.8.2, the time also includes interventions of the test operator. The total number of commands sent to DMAppFpgaProg during one test iteration is 5322. Each command is sent only once. The commands in `1_execute_DMAppFpgaProg` are not counted as explained in the group description at the beginning of this section.

Group	*Result*	*Commands sent*
`1_execute_DMAppFpgaProg`	Success	not counted
`2_basic`	Success	31
`3_write_verify_erase_array_from`	Success	1686
`4_write_verify_erase_from`	Success	1683
`5_write_verify_erase_array`	Success	1922

Table 5.2: DMAppFpgaProg test results

Conclusion

This thesis dealt with analysis, design, realization, and testing of a firmware for AT32UC3C family of MCUs that would make it possible to erase, write, and verify the program Flash memory of the MCU and array and FlashROM of IGLOO nano FPGAs. The firmware that has been created is divided into two independent programs: DMBootloader and DMAppFpgaProg.

DMBootloader is the the only part of the wireless programming support that needs to written into the MCU's Flash memory by wired programmer. It is the first program run after reset of the MCU and it then allows to erase, write and read every Flash memory page except those occupied by itself. It can execute programs stored from any of the pages. Execution of these operations is based on commands from the control system. Verification is done by the control system through the reading of pages. Program code of DMBootloader in the Flash memory is protected against accidental modification through fuse bits of the MCU so it can be modified only through a HW programmer.

DMAppFpgaProg can be written to the program memory of the MCU and executed wirelessly through DMBootloader. It is a wrapper for the DirectC FPGA ISP support from Microsemi company. It makes it possible to interface the DirectC wirelessly and execute any operation with IGLOO nano FPGA that is supported by DirectC. Thus a user can use advanced features of DirectC, e.g. AES encryption of array and FlashROM. DMAppFpgaProg operates based on commands from the control system. During the implementation phase, a few errors in DirectC source codes were found and were fixed.

Both of the programs were tested for compliance with the requirements given in the assignment. The testing discovered an error in DirectC paging and it has been fixed. The solution of the wireless programming, implemented in DMBootloader and DMAppFpgaProg, meets all of the assigned requirements.

In the future, the results of this thesis could be further improved by making the testing process of DMAppFpgaProg to be fully automated, by implementing protection of the Flash memory pages other than those for DMBootloader,

or by implementing confidentiality and integrity in the communication protocol between the control system and the MCU.

Bibliography

[1] RailSystem. [online]. *Balise. [cit. 2018-11-28].* Available from: http://www.railsystem.net/balise/

[2] MER MEC S.p.A. [online]. *Eurobalise. 2018. [cit. 2018-11-28].* Available from: http://www.mermecgroup.com/protect/lineside-equipment/635/eurobalise.php

[3] AŽD Praha s.r.o. [private document]. *Specifikace protokolu YUP. 2018. [cit. 2019-01-31].*

[4] Microchip Technology Inc. [online]. *Advanced Software Framework (ASF). [cit. 2019-01-23].* Available from: https://www.microchip.com/mplab/avr-support/advanced-software-framework

[5] AŽD Praha s.r.o. [private document]. *Popis FW pro diagnostický modul. 2018. [cit. 2019-01-31].*

[6] *Function for CRC32 computation.* [online]. The FreeBSD Project. 2011. [cit. 2019-03-01]. Available from: https://svnweb.freebsd.org/base/stable/9/sys/libkern/crc32.c?revision=225736&view=co

[7] Microsemi Corporation. [online]. *IGLOO nano Starter Kit User's Guide. 2018. [cit. 2018-08-25].* Available from: https://www.microsemi.com/document-portal/doc_download/130838-igloo-nano-starter-kit-user-s-guide

[8] Microsemi Corporation. [online]. *Downloads: Program Debug v11.8 (Windows). 2019. [cit. 2019-02-16].* Available from: soc.microsemi.com/download/reg/default.aspx?f=ProgramDebugv11_8_WIN

[9] Microchip Technology Inc. [online]. *AT32UC3C Series - Complete Datasheet. 2012. [cit. 2018-08-06].* Available from: http://ww1.microchip.com/downloads/en/DeviceDoc/doc32117.pdf

[10] *Flash FPGAs give designers more flexibility.* [online]. Morin T., Microsemi Corp. 2015. [cit. 2019-02-12]. Available from: https://www.embedded.com/electronics-blogs/industry-comment/4438457/Flash-FPGAs-give-designers-more-flexibility

[11] Microsemi Corporation. [online]. *AT32UC3C Series - Complete Datasheet. 2012. [cit. 2019-02-12].* Available from: http://ww1.microchip.com/downloads/en/DeviceDoc/doc32117.pdf

[12] Microsemi Corporation. [online]. *DirectC v4.1 User Guide. 2018. [cit. 2018-08-06].* Available from: https://www.microsemi.com/product-directory/programming/4980-embedded-programming#downloads

[13] *MSYS2 homepage.* [online]. The MSYS2 Developers. [cit. 2019-03-13]. Available from: https://www.msys2.org/

[14] *Programovací nástroj pre mikrokontroléry rodiny ST10F.* [online]. Sučan, J. Bakalárska práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií. 2017. [cit. 2019-03-01]. Available from: https://dspace.cvut.cz/handle/10467/69320

[15] *A references, guides and tutorials on the mailslot programming with the Winsock 2 and C code.* [online]. winsocketdotnetworkprogramming.com. 2019. [cit. 2019-03-10]. Available from: https://www.winsocketdotnetworkprogramming.com/winsock2programming/winsock2advancedmailslot14.html

Appendix A

Acronyms

AES Advanced Encryption Standard

API Application Programming Interface

ASCII American Standard Code for Information Interchange

ASF Advanced Software Framework

COBS Consistent Overhead Byte Stuffing

CRC Cyclic Redundancy Check

DM Diagnostic Module

ETCS European Train Control System

FPGA Field-programmable Gate Array

FW Firmware

GCC GNU Compiler Collection

GNU GNU's Not Unix

GPIO General-purpose I/O

GUI Graphical User Interface

HW Hardware

I/O Input/Output

IDE Integrated Development Environment

IPC Inter-process Communication

ISP In-system Programming

JTAG Joint Test Action Group

MCU Microcontroller

MOSFET Metal-Oxide-Semiconductor Field-Effect Transistor

MS Microsoft

OS Operating System

PCB Printed Circuit Board

PLL Phase-locked Loop

POSIX Portable Operating System Interface

RAM Random Access Memory

SW Software

TCL Tool Command Language

USART Universal Synchronous / Asynchronous Receiver and Transmitter

USB Universal Serial Bus

Appendix B

Contents of enclosed CD

- `readme.txt` ..... brief description of content of this CD
- `src`
  - `impl` ..... source files of the firmware for wireless programming support
    - `DMBootloader` ..... bootloader for wireless programming of AT32UC3C MCUs
      - `DMBootloader`
        - `Release` ..... the executable images in binary and Intel Hex formats
    - `DMAppFpgaProg` ..... application for wireless programming of IGLOO nano FPGAs
      - `DMAppFpgaProg`
        - `Release` ..... the executable images in binary and Intel Hex formats
    - `DiagnosticModule` ..... library containing functionality that can be shared among programs for Diagnostic Module
      - `DiagnosticModule` ... source codes and include files of the library
        - `Release`
          - `libDiagnosticModule.a` ..... the static library archive
  - `testing` ..... programs, scripts, and data files for testing of the wireless programming support
    - `DMBootloader` .... test files for DMBootloader
    - `DMAppFpgaProg` ... test files for DMAppFpgaProg
    - `yup-comm` ..... source codes and executable files of the utility for communication with the firmware for Diagnostic Module through YUP communication protocol
  - `thesis` ..... source files of this thesis in LaTeX format
- `text`
  - `DP_Sučan_Ján_2019.pdf` ..... text of the thesis in PDF format

Appendix C

Format of DMBootloader commands

These tables describe the format of DMBootloader's application layer protocol. The format of multi-byte values is little-endian. The commands and replies are sets of bytes sent or received through the YUP communication protocol.

Write command (3 B to 515 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	CMD	Command code 0x00.
0x01	uint 16 bit	PAGE	Number of the virtual page to be written.
0x03	uint 8 bit	DATA[n]	Array of bytes to be written into the page from its 0th byte (including). The size of the array can vary from 1 B to 512 B. Non-specified bytes of the page will be set to 0xFF.
Reply (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Return code: 0x00 = Success 0x10 = Unknown CMD code 0x11 = Missing PAGE number 0x12 = Invalid PAGE number 0x13 = Invalid size of the command 0x30 = Error when writing the page

Table C.1: Write command

Erase command (3 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	CMD	Command code 0x01.
0x01	uint 16 bit	PAGE	Number of the virtual page to be erased.
Reply (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Return code: 0x00 = Success 0x10 = Unknown CMD code 0x11 = Missing PAGE number 0x12 = Invalid PAGE number 0x14 = Invalid size of the command 0x31 = Error while erasing the page

Table C.2: Erase command

Read command (3 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	CMD	Command code 0x02.
0x01	uint 16 bit	PAGE	Number of the virtual page to be read.
Reply (1 B or 513 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Return code: 0x00 = Success 0x10 = Unknown CMD code 0x11 = Missing PAGE number 0x12 = Invalid PAGE number 0x14 = Invalid size of the command 0x32 = Error while reading the page
0x01	uint 8 bit	DATA[n]	512 B of data from the page. This array is present only if RETCODE is equal to 0x00.

Table C.3: Read command

Exec command (3 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	CMD	Command code 0x03.
0x01	uint 16 bit	PAGE	Number of the virtual page where the program starts.
Reply (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Return code: 0x00 = Success 0x10 = Unknown CMD code 0x11 = Missing PAGE number 0x12 = Invalid PAGE number 0x14 = Invalid size of the command

Table C.4: Exec command

GetId command (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	CMD	Command code 0xFF.
Reply in case of an error (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Return code: 0x10 = Unknown CMD code
Reply in case of success (2 B to 64 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	ID[n]	String of ASCII characters containing identification of the program. The string is not terminated by null byte.

Table C.5: GetId command

Appendix D

Activity diagrams of DMBootloader

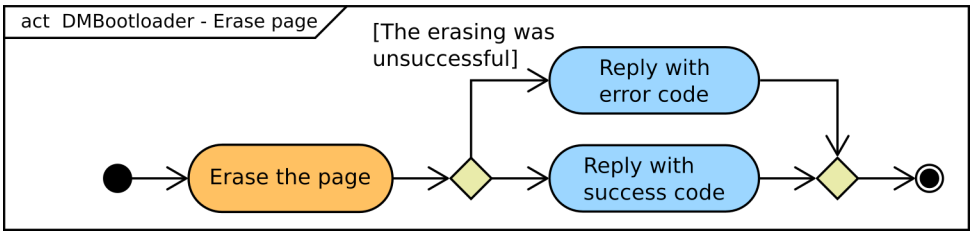

Figure D.1: Activity diagram for Erase command of DMBootloader

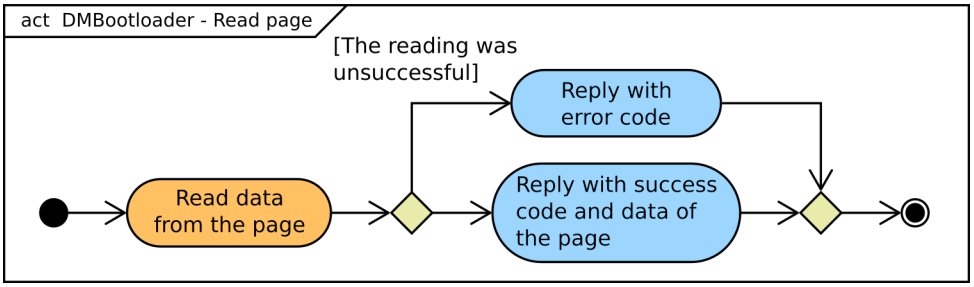

Figure D.2: Activity diagram for Read command of DMBootloader

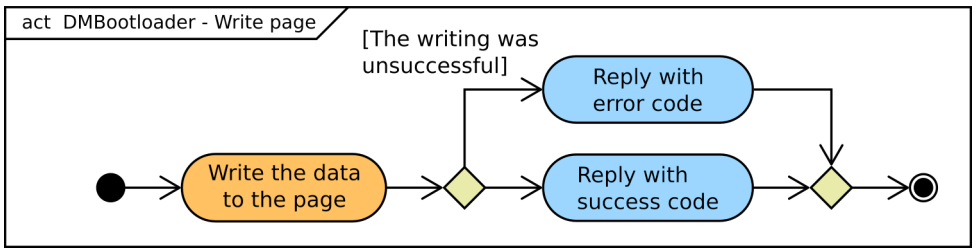

Figure D.3: Activity diagram for Write command of DMBootloader

Appendix E

Format of DMAppFpgaProg commands

These tables describes the format of DMAppFpgaProg's application layer protocol. The format of multi-byte values is little-endian. The commands and replies are sets of bytes sent or received through the YUP communication protocol.

Exec command (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	CMD	Command code 0x00.
0x01	uint 8 bit	OPCODE	Code of the DirectC operation to execute.
Reply in case of an error (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Return code: 0x10 = Unknown CMD code 0x11 = Invalid size of the command
Reply in case of the operation finish (2 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Value 0x00 for differentiation of the replies by their size.
0x01	uint 8 bit	DIRC_RET	Return value of the finished DirectC operation.
Reply in case of the operation requests data (9 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	DIRC_OP	Code of the requesting DirectC operation.
0x01	uint 32 bit	OFFSET	Offset of the page in the .DAT file.
0x02	uint 32 bit	COUNT	Number of bytes of the page.

Table E.1: Exec command

GetText command (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	CMD	Command code 0x01.
Reply in case of an error (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Return code: 0x10 = Unknown CMD code 0x11 = Invalid size of the command
Reply in case of success (1 B to 8193 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Value 0x00 for differentiation of the replies by their size.
0x01	uint 8 bit	LOG[n]	String of ASCII characters containing text output of the most recent DirectC operation. The string is not terminated by null byte.

Table E.2: GetText command

GetState command (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	CMD	Command code 0x02.
Reply in case of the application is waiting for Exec command (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Return code: 0x00.
Reply in case of an error (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Return code: 0x10 = Unknown CMD code 0x11 = Invalid size of the command
Reply in case of the ongoing operation is waiting for data (9 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	DIRC_OP	Code of the waiting DirectC operation.
0x01	uint 32 bit	OFFSET	Offset of the page in the .DAT file.
0x02	uint 32 bit	COUNT	Number of bytes of the page.

Table E.3: GetState command

Cancel command (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	CMD	Command code 0x03.
Reply (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Return code: 0x00 = Success 0x10 = Unknown CMD code 0x11 = Invalid size of the command 0x12 = There is no DirectC operation ongoing

Table E.4: Cancel command

ProvideData command (2 B to 1025 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	CMD	Command code 0x04.
0x01	uint 8 bit	DATA[n]	Data of the page (1 B to 1024 B)
Reply in case of an error (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Return code: 0x10 = Unknown CMD code 0x11 = Invalid size of the command
Reply in case of the operation finish (2 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Value 0x00 for differentiation of the replies by their size.
0x01	uint 8 bit	DIRC_RET	Return value of the finished DirectC operation.
Reply in case of the ongoing operation requests more data (9 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	DIRC_OP	Code of the requesting DirectC operation.
0x01	uint 8 bit	OFFSET	Offset of the page in the .DAT file.
0x02	uint 8 bit	COUNT	Number of bytes of the page.

Table E.5: ProvideData command

VjtagVpump command (2 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	CMD	Command code 0x05.
0x01	uint 8 bit	PWR	Whether to turn the voltages on or off. 0x00 = Turn off the voltages 0x01 to 0xFF = Turn on the voltages
Reply (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Return code: 0x00 = Success 0x10 = Unknown CMD code 0x11 = Invalid size of the command

Table E.6: VjtagVpump command

GetId command (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	CMD	Command code 0xFF.
Reply in case of an error (1 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	RETCODE	Return code: 0x10 = Unknown CMD code
Reply in case of success (2 B to 64 B)			
Offset	*Format*	*Name*	*Description*
0x00	uint 8 bit	ID[n]	String of ASCII characters containing identification of the program. The string is not terminated by null byte.

Table E.7: GetId command

Appendix F

Activity diagrams of DMAppFpgaProg

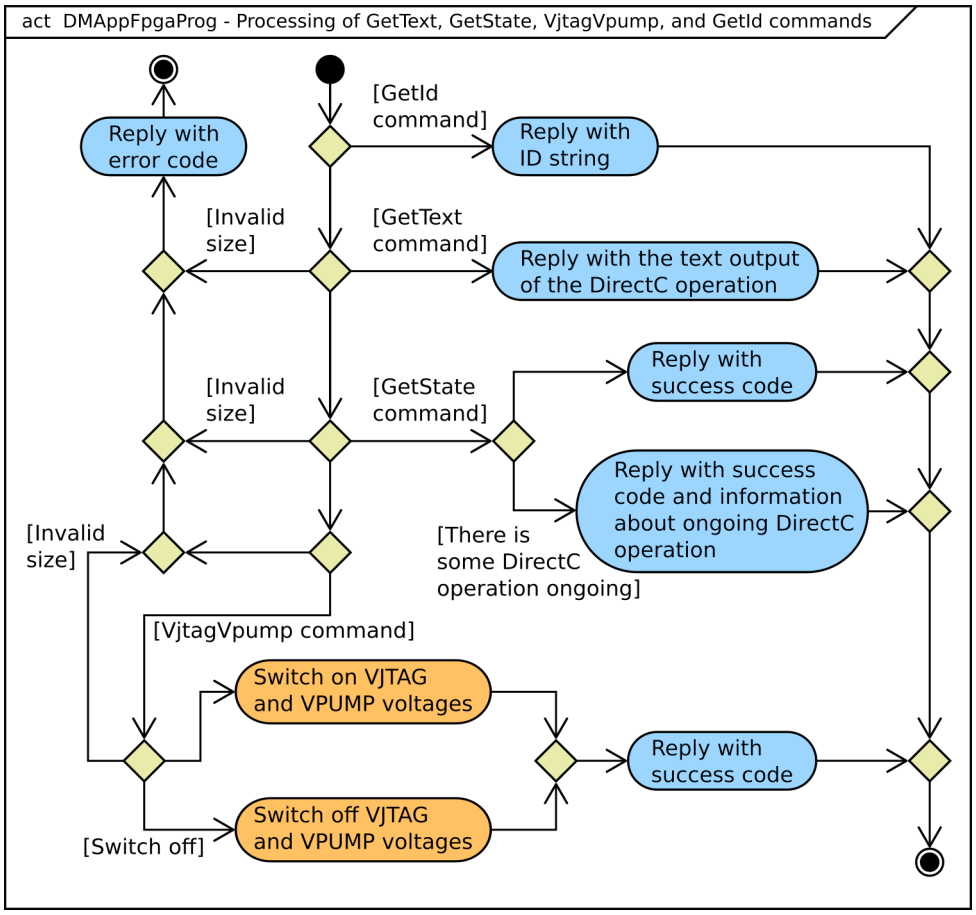

Figure F.1: Processing of GetText, GetState, VjtagVpump, and GetId commands