BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF RADIO ELECTRONICS

ÚSTAV RADIOELEKTRONIKY

PROGRAM UPDATE OF ZYNQ-BASED DEVICES

AKTUALIZACE PROGRAMU V ZAŘÍZENÍ S OBVODY ZYNQ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — Bc. Branislav Michálek
AUTOR PRÁCE

SUPERVISOR — Ing. Michal Kubíček, Ph.D.
VEDOUCÍ PRÁCE

CONSULTANT — Ing. Jan Král
KONZULTANT

BRNO 2019

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Elektronika a sdělovací technika**
Ústav radioelektroniky

Student: Bc. Branislav Michálek ***ID:*** 164338
Ročník: 2 ***Akademický rok:*** 2018/19

NÁZEV TÉMATU:

Aktualizace programu v zařízení s obvody Zynq

POKYNY PRO VYPRACOVÁNÍ:

Prozkoumejte možnosti bootování obvodů Zynq a možnosti aktualizace jejich konfigurace. Vytvořte ukázkovou aplikaci pro obvod Zynq s možností aktualizace bitstreamu FPGA i programu pro procesor přes Ethernet. Aplikaci doplňte vhodným programem spustitelným z příkazové řádky OS Windows a Linux. Řešení musí zajistat načtení záložní konfigurace v případě selhání předchozí aktualizace.

DOPORUČENÁ LITERATURA:

[1] MAXFIELD, Clive. The design warrior's guide to FPGAs: devices, tools, and flows. Boston: Newnes/Elsevier, c2004. ISBN 0-7506-7604-3.

[2] CATSOULIS, John. Designing embedded hardware. 2nd ed. Sebastopol, CA: O'Reilly, c2005. ISBN 05-96--0755-8.

Termín zadání: 4.2.2019 ***Termín odevzdání:*** 16.5.2019

Vedoucí práce: Ing. Michal Kubíček, Ph.D.
Konzultant: Ing. Jan Král

prof. Ing. Tomáš Kratochvíl, Ph.D.
předseda oborové rady

ABSTRACT

Among many which are placed on modern embedded systems is also the need of storing multiple system boot image versions and the ability to select from them upon boot time, depending on a function which they provide. This thesis describes the development of a system update application for Xilinx Zynq-7000 devices. The application includes a simple embedded HTTP server for a remote file transfer. A client is allowed to upload the boot image file with the system update from either command line application or using the web page developed for this purpose.

KEYWORDS

Zynq-7000, System on Chip, Booting, HHypertext Transfer Protocol

ABSTRAKT

Mezi mnohé požadavky kladené na moderní vestavěné systémy patří potřeba uchovávat více verzí jejich systémového obrazu (firmware, software nebo hardwareová konfigurace) a také možnost volby, který z těchto obrazů systém načte během procesu bootování, v závislosti na funkci, kterou daný obraz poskytuje. Tato diplomová práce popisuje vývoj aplikace pro zařízení s obvody Zynq firmy Xilinx, jejíž funkcí je provést aktualizaci systému. Aplikace zahrnuje jednoduchý vestavěný HTTP server sloužící ke vzdálenému přenosu souborů. Klientovi umožňuje nahrát soubor s obrazem skrze aplikaci spustitelnou z příkazové řádky, nebo prostřednictvím webové stránky, která byla navržena k tomuto účelu.

KLÍČOVÁ SLOVA

Zynq-7000, systém na čipu, bootování, hypertextový přenosový protokol

MICHÁLEK, Branislav. *Program update of Zynq-based devices*. Brno, Rok, 49 p. Master's Thesis. Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Radio Electronics. Advised by Ing. Michal Kubíček, Ph.D.

DECLARATION

I declare that I have written the Master's Thesis titled "Program update of Zynq-based devices" independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the thesis and listed in the comprehensive bibliography at the end of the thesis.

As the author I furthermore declare that, with respect to the creation of this Master's Thesis, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll., Section 2, Head VI, Part 4.

Brno ....

author's signature

ACKNOWLEDGEMENT

First and foremost, I would like to thank God Almighty for giving me the strength, knowledge, ability and opportunity to undertake the whole study. I would like to express my gratitude to my consultant Jan Král for the useful comments, remarks and engagement through the learning process of this master thesis. Furthermore I would like to thank my supervisor Michal Kubicek for his helpful advice concerning the study issues. I would like to thank all my loved ones, who have supported me throughout entire process, both by keeping me harmonious and helping me putting pieces together.

Brno ....

author's signature

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic

http://www.six.feec.vutbr.cz

ACKNOWLEDGEMENT

Research described in this Master's Thesis has been implemented in the laboratories supported by the SIX project; reg. no. CZ.1.05/2.1.00/03.0072, operational program Výzkum a vývoj pro inovace.

Brno ....

author's signature

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

This work was done as part of the InterOp ATCZ175 project of the Interreg program of the European Union. The project is co-financed by the European Regional Development Fund and the state budget of the Czech Republic.

Contents

List of Figures

Introduction

A system update is a mechanism which ensures that a embedded device running an older version of the system runs with a more recent release when the update mechanism is done. This includes updating everything that defines the system (bootloader, operation system kernel, software applications, hardware configuration, etc.), restarting running processes and eventually a reboot [1].

An ideal mechanism never ends up in an inconsistent state, always keeps the device usable (in case of update failure the device fallbacks to previous state or a recovery mode), minimizes downtime while updating, ensures integrity and security. A properly developed system update mechanism can reduce [2]:

- Reduce development cycle. The possibility of adding new functionality means all features do not need to be implemented in the original release.
- Fault management. A system update fixes defects with a patch.
- Extend the life cycle of embedded device with updated software and hardware.

This thesis focuses on the design and implementation of the system update for Xilinx Zynq Single-on-Chip (SoC) devices. The first chapter briefly introduces the Zynq devices booting stages, a boot image layout and its creation process. The second chapter describes some of the most used communication protocols used for remote file transfer.

The third chapter presents the system update application design and implementation, the device multiboot and fallback functionality and the process of creation of customized boot image file.

The last chapter summarizes the achieved results of the thesis and proposes suggestions for further development.

1 Zynq Devices Booting

Xilinx Zynq-7000 SoC is a processor-centric platform that offers software, hardware and Input/Output (I/O) programmability). Zynq-7000 integrates the ARM based Processing System (PS) with 28 nm Programmable Logic (PL) in a single device.

The key component of the PS is Application Processor Unit (APU). The APU contains one or two identical Cortex-A9 Central Processing units (CPU), each with its own 32 KB instruction and data cache, a Floating-Point Unit (FPU) for acceleration of floating-point arithmetic and so-called NEON unit for multimedia operations. Each CPU also contains standard Memory Management Unit (MMU) which is needed in Linux-based operating systems. The architectural overview of the Zynq-7000 PS is depicted on Fig. 1.1.

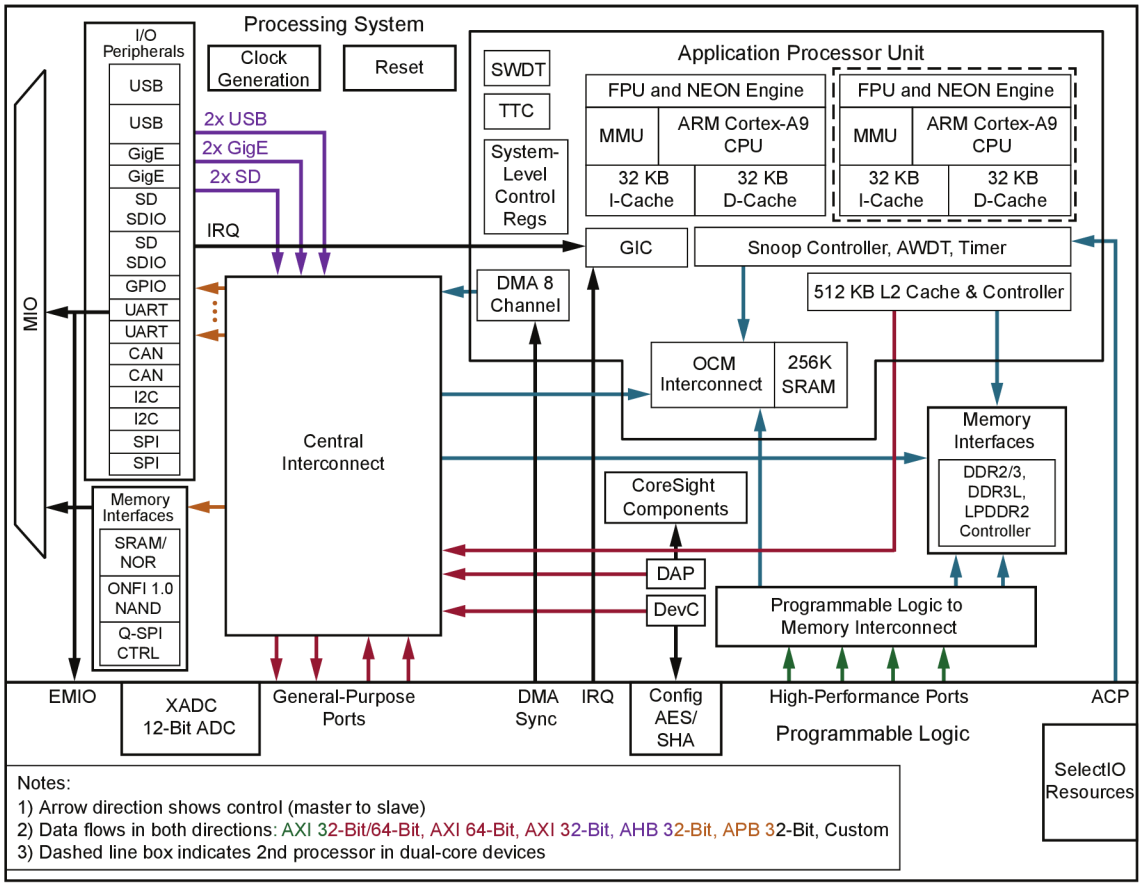

Fig. 1.1: Zynq-7000 PS architectural overview [3].

The PS also includes 256 KB of shared On-Chip Memory (OCM) for general purpose, controller for external Double Data Rate (DDR) Random Access Memory (RAM), Non-Volatile Memory (NVM) interfaces supporting NAND, 8-bit parallel NOR, and up to two quad serial peripheral interface (QSPI) NOR flash devices.

Main communication peripherals include two Gigabyte Ethernet MAC modules, Universal Serial Bus (USB) 2.0, Universal Asynchronous Receiver-Transmitter (UART), Serial Peripheral Interface (SPI), Inter-Integrated Circuit (I2C), Controller Area Network (CAN), and General-Purpose Input/Output (GPIO) controllers. Other custom application-specific peripherals can be implemented in the PL, including complex hardware functions as digital signal processing (DSP), high performance computing (HPC) etc.

Main PL components include Configurable Logic Blocks (CLB), 36 KB of dual-port block RAM, DSP slices, I/O blocks, serial data transceivers and two 12-bit Analog to Digital Converters (ADC). Each CLB contains 16 flip-flops, eight 6-input Lookup Tables (LUT) for programmable logic or distributed memory implementation, and two 4-bit adders for arithmetic functions.

1.1 Boot Devices and Boot Modes

A boot device is a flash memory device, which the system boots from. This is referred as a master boot mode [4]. Depending on selected boot device, there are four possible master boot modes:

- Quad-SPI Boot,
- NAND Boot,
- NOR Boot,
- SD Card Boot.

In QSPI boot mode, the boot device is a non-volatile serial NOR flash memory device, connected to a host system via the SPI. This mode supports multiple input and output (multi I/O) SPI devices with single bit (Single I/O) as well as optional two bit (Dual I/O) and four bit (Quad I/O) data bus width.

1.2 Boot Stages

Zynq-7000 devices use a multi-stage boot process. Both a non-secure and a secure boot is supported. For a secure boot, the PL must be powered on to enable the use of the security block located within the PL, which provides 256-bit Advanced Encryption Standard (AES) and Secure Hash Algorithm (SHA) decryption/authentication.

The PS hardware boot stages include power supply ramping, clocking, resets, pin strap sampling and the Phase-Locked Loop (PLL) initialization. Within 45 clock cycles of the PS reference clock, the hardware samples the seven boot mode strap pins and stores their settings in read-only registers.

1.2.1 Stage 0

A read-only BootROM code, located in the on-chip Read Only Memory (ROM), executes on the primary CPU after a system reset is performed. There are two basic types of reset - the Powe-On Reset (POR) which resets the whole system including all of the registers. All states, except those stored in the eFuse and Battery-Backed RAM (BBRAM) are lost. A non-POR causes the BootROM to execute, but the BootROM retains knowledge about the security level of the previous boot in the `REBOOT_STATUS` register. The non-POR sources include `PS_SRST` pin and internal system resets, also referred to as soft reset (e.g. software controlled reset, watchdog timer, etc.).

The PL power-up and initialization sequences can occur in parallel with or after the PS start-up. If the BootROM needs the PL powered up, then early in the BootROM execution the BootROM

The main tasks of the BootROM are to configure the system, copy the first partition (usually a primary bootloader) from the boot device to the OCM, and then branch the code execution to the OCM. Before the branch, the BootROM disables access to its ROM code.

The BootROM code reads the boot mode register to determine if a master or slave boot mode is used, and if master, also the type of boot device used. Then the BootROM code searches for a valid boot image header (also referred as the BootROM header). The header search continues until a valid header is found or the entire range has been searched. Invalid header is detected by calculating it's checksum.

In the secure boot mode, the BootROM has the ability to authenticate and decrypt the encrypted bootloader partition. Authenticating the bootloader partition the BootROM also validates its data integrity. In the non-secure boot mode, corruption in bootloader partition is *not* recognized [3].

1.2.2 Stage 1

This is generally a first-stage bootloader (FSBL), but it can be any user-controlled code. As the stage 1 code is loaded to the OCM, its size is limited to 192 KB [3]. FSBL reads the partition header table of the boot image (see section 1.3.1) to find a bitstream, second-stage bootloader (SSBL) or bare-metal application partition. Then it is responsible for:

- initialization of selected system features and peripherals (e.g. DDR memory, I/Os, system clock, etc.) with the PS configuration data,
- programming the PL using a bitstream, if provided,
- loading a SSBL or a bare-metal application code into DDR memory,

- handoff the execution to the SSBL or bare-metal application.

The FSBL also supports loading partitions from eMMC flash devices, configured as a secondary boot device [4]. This is useful when there is a small QSPI flash which does not meet the memory requirements to store all the partitions. In this case, the primary boot mode needs to be set to QSPI (through the boot mode pins), but only the FSBL is placed on the QSPI flash. All the other partitions are on the eMMC flash instead. The FSBL ignores the configured primary boot mode and loads the other partitions from eMMC.

If loading of any partition fails for some reason, the FSBL does a fallback and enables the BootROM to load another bootable image that is known to be in a good state, if such an image is present in the boot device. This image is often referred to as the *golden* image [4]. The FSBL updates a `Multiboot Register` and does a non-POR soft system reset, so that the BootROM executes again and loads the image pointed to by the `Multiboot Register`.

The FSBL does a fallback if any of the partitions is corrupted. A corrupted partition is detected either by performing an RSA authentication or calculating a md5 checksum.

1.2.3 Stage 2

During the stage 2 an application software is generally loaded, but it could be also a SSBL [4]. Booting larger systems (e.g. Linux-based operating systems) may require more complex procedures, which are unable to be performed by FSBL due to its own limitations. The SSBL is a special software which is able to properly load such a system and transfer the execution to it.

The Universal Boot Loader (U-boot) is open-source SSBL that is frequently used in embedded devices for booting Linux kernel, device tree, root file system and Linux applications [5]. Unlike the FSBL, U-boot typically runs in DDR, not OCM. Its features include the ability to load, decrypt, authenticate, and execute images from Ethernet, flash memory, SD cards and USB. Users are able to interact with U-boot and control the boot process using the built-in command interpreter.

As the U-boot's commands are fairly low-level, it takes several steps to boot a kernel, but this also makes U-boot more flexible than other bootloaders, since the same commands can be used for more general tasks. Copying data with U-boot requires explicit specification of the physical memory addresses in destination memory [5].

1.2.4 Multiboot

Multiboot is a feature that allows the device to select from multiple boot images stored in a boot device [4]. The boot image which is to be loaded can be user-selectable, or it can be selected automatically by the system upon the user-defined decision criteria. To select an image, the FSBL writes its base address divided by 32 KB into the `MultibootRegister` and then generates a non-POR system reset. When the BootROM executes after the reset, it looks for the boot image header pointed to by the `MultibootRegister`.

1.3 Boot Image Creation

A boot image is a binary file usually stored in a boot device which allows the associated hardware to boot. This may include the operating system, utilities and diagnostics, as well as boot and data recovery information. Once built, the boot image can be distributed to a target device, patched within reasonable limits, and remain disposable in case of a need.

The boot image is created by building the required boot image header, processing input data files and appending tables which describe them. An input file can be software, hardware configuration (bitstream), or general data [4]. All these files are referred to as *images*. A software image is provided either in binary (BIN) or Executable and Linking Format (ELF) file format. Every image can have one or more *partitions*, e.g. an ELF file can have multiple loadable sections, each of which forms a partition in the boot image. There could be performed a chosen type of encryption and authentication on each partition.

Building the boot image in general involves the following steps [6]:

1. Creating a Boot Image Format (BIF) file.
2. Running the Xilinx Bootgen utility to create a bootable binary file.

The BIF file specifies every component of the boot image according to boot order. Several attributes can be applied to each component, e.g. the `bootloader` attribute applied to an ELF file identifies the file as the FSBL. The following code snippet is an example of a simple BIF file:

```
the_ROM_image:
    {
    [init]init\_data.int
    [bootloader]fsbl.elf
    bitstream.bit
    application.elf
    }
```

Bootgen is a Xilinx tool which merges image files together and builds a boot image. Bootgen comes with both a Graphical User Interface (GUI) and a command line option. The command line option can be scripted [6].

1.3.1 Boot Image Layout

Every boot image consists of the following components (see Fig. 1.2) [6]:

- a boot image header,
- a register initialization table,
- an image header table,
- a partition header table,
- a FSBL image,
- optionally other image partitions.

The boot image header (also referred to as the BootROM header) resides in the beginning of a boot image. The boot image header is a structure that contains information related to booting the FSBL. There is only one such structure in the entire boot image. This header is parsed by the BootROM to get determined, where the FSBL is stored in boot device and where it needs to be loaded in OCM. In case of secure boot, some encryption and authentication related parameters are stored in there. Boot image header also provides a User Defined Field (UDF). This 76 bytes of free space has potential to be used for storing boot image version and identification, time stamp, or other user defined data. The UDF is written using the Bootgen. The input user defined data is provided through a text file in the form of a hexadecimal string.

Immediately after the fixed-sized boot image header (BootROM header) there is typically the *image header table*. It is a structure comprising a own header, followed by a linked list of *image headers*. The header of the image header table contains information which is common across all images, e.g. the total number of partitions present in the boot image. Each image header contains information, such as the image name, number of partitions associated with this image and pointer to a first associated *partition header*.

The partition header table is an array of structures containing information and attributes related to each partition, such as partition size, address in boot device, load address in RAM, encryption and authentication related information, etc.

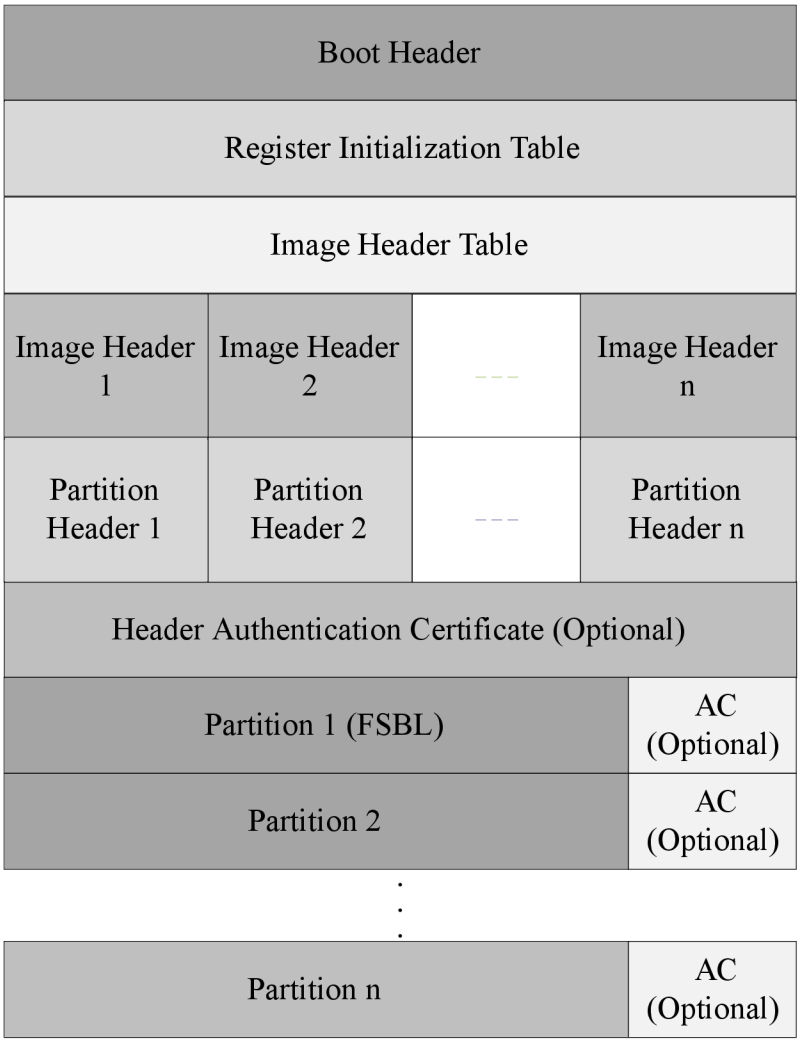

Fig. 1.2: Boot image layout [6].

2 Remote File Transfer

Both the PS and PL of Zynq-7000 are programmed using a conventional device driver, providing support for a user initiated update. In typical system update flow, the central site for developing and distributing software for fielded embedded systems creates a boot image, which needs to be transferred to the system, typically using wired or wireless Ethernet. Initial destination for the new boot image is usually DDR RAM [2].

After the image is transferred to the device, a special software (either a user code or U-boot) copies the image from RAM to another location in RAM, OCM, or NVM (e.g. QSPI flash memory).

2.1 Remote File Transfer Protocols

File transfer is the transmission of a binary file over a telecommunication network from one system to another. Typically, file transfer is mediated by a file transfer communication protocol, which is a convention that describes how to transfer files between two endpoints.

Most of the protocols are designed for the Internet and its Internet protocol suite, commonly known as TCP/IP protocol stack [8]. Internet is a packet-switched network which transmits data divided into units called packets. A packet comprises of a header and a payload. Network applications make use of the services provided by the lower layers, especially the transport layer protocols: Transmission Control Protocol (TCP) or User Datagram Protocol (UDP), which provide reliable or unreliable pipes to other processes.

File transfer protocols are higher-level protocols that operate in the application layer of the TCP/IP stack [8]. File data is encapsulated into transport layer protocol units, such as TCP or UDP segments. The segments transmission is handled by lower layer protocols. Apart from a stream of bits from a file, a file transfer protocol unit may also contain some relevant metadata, such as the file name, file size, file attributes, etc. Some of the most widely used file transfer protocols are:

- File Transfer Protocol (FTP),
- Trivial File Transfer Protocol (TFTP),
- Secure Copy Protocol (SCP),
- Secure File Transfer Protocol (SFTP),
- Hypertext Transfer Protocol (HTTP).

2.1.1 File Transfer Protocol

FTP is a standard network protocol used for the transfer of files between a client and a server on a network. FTP users may authenticate themselves with a clear-text sign-in protocol, normally in the form of a username and password, or connect anonymously if the server is configured to allow it [9].

In both, the active and passive modes of establishing the data connection, the client creates a TCP connection from a random, usually unprivileged port to the FTP server command port 21. The server responds over the control connection with three-digit status code in ASCII with an optional, human-readable explanation of request, e.g. "200 OK".

While transferring data over the network, four data representations can be used, from which most important are [9]:

- ASCII mode - inappropriate for files that contain data other then plain text.
- Image mode (also called binary mode) - the recipient stores the bytestream as the sending machine sends each file byte by byte.

Data transfer itself can be done in three modes [9]:

- Stream mode - the data is sent as a continuous stream, all processing is left to TCP. No terminator character is needed.
- Block mode - FTP breaks the data into several blocks comprising header, byte count, and data field.
- Compressed mode - the data is compressed using a simple algorithm.

FTP does not encrypt its traffic. Usernames, passwords, commands and data can be read by anyone able to perform packet capture on the network. For secure transmission FTP may be secured either with Secure Sockets Layer (SSL) or Transport Layer Security (TLS) protocols or replaced with SFTP.

2.1.2 Trivial File Transfer Protocol

TFTP is a simple derivate of FTP which allows a client to get a file from or put a file onto a remote host [10]. It is mainly used for transferring firmware images and configuration files to network appliances like routers, firewalls, IP phones, etc. [11]. TFTP has been used for this purpose because it is very simple to be implemented by code with a small memory footprint. This is especially useful for low resourced Single-Board Computers (SBC) and SoCs.

TFTP uses UDP as its transport protocol. A transfer request is always initiated targeting port 69, but the data transfer ports are chose independently by the sender and receiver during the transfer initialization. Due to its simplicity, TFTP lacks most of the more advanced features offered by more complex file transfer protocols.

TFTP only reads and writes files from or to a remote server. It cannot list, delete, or rename files or directories and it has no provision for user authentication [10].

2.1.3 Secure Copy Protocol

SCP is based on Secure Shell (SSH) protocol [12]. It was developed for secure transfer of files between a local and a remote host or between two remote hosts. It uses the same mechanism for authentication as SSH, thereby it ensures the authenticity and confidentiality of the data in transit. A client can send files to a server and also request files from a server. Normally, a client initiates an SSH connection to the remote host, and requests an SCP process to be started on the remote server [13]. The remote SCP process can operate in one of two modes:

- source mode, which reads files and sends them back to the client,
- sink mode, which accept the files sent by the client.

2.1.4 Secure File Transfer Protocol

SFTP (also referred to as SSH File Transfer Protocol) is an extension of the SSH protocol [12]. Compared to SCP, which only allows file transfer, SFTP allows for a range of management operations with remote files, including resuming interrupted transfers, directory listings, and remote file removal. While SCP is better designed for one-time file transfers between two networked endpoints, the SFTP does the same and adds the data management [13].

2.1.5 Hypertext Transfer Protocol

Hypertext Transfer Protocol is an application layer protocol designed within the framework of the Internet protocol suite [14]. HTTP usually uses TCP as underlying and reliable transport layer protocol. HTTP was developed to facilitate World Wide Web, where hypertext documents include hyperlinks to other resources that the user can access.

HTTP works as a request-response protocol in a client-server computing model [14]. A web browser may represent the client side, and an application running on a system hosting a website may represent the server side. The client submits a HTTP *request* message to the server by establishing a TCP connection to a particular port on the server, typically port 80 [14]. The server waits for the client's request and returns a *response* comprising of a status line and a message. The body of this message is typically a requested resource. HTTP resource is identified and located on the server by a Uniform Resource Locator (URLs) [14].

The client and server communicate by sending plain-text ASCII messages. The request message consists of the following [14]:

- a request line,
- a request header fields,
- an empty line,
- an optional message body.

The request line and other header fields must all be terminated with Carriage Return (CR) and Line Feed (LF) characters [14].

HTTP defines methods to indicate a desired action to be performed on a identified resource. What this resource represents, whether pre-existing data or data that is generated dynamically, depends on the implementation of the server. The most used methods include [14]:

- GET - the GET method *requests* a representation of the specified resource.
- HEAD - the HEAD method asks for a response identical to that of a GET request, but without the response body.
- POST - the POST method requests that the server *accept* the entity enclosed in the request. The POST data might be for example a block of data that is the result of submitting a web form to a data-handling process, or a file to be uploaded to the server.
- OPTIONS - the OPTIONS method returns the HTTP methods that the server supports for the specified URL.

Format of the response message is similar to the format of the request message. It contains a status line, which comprises a status code and reason message. HTTP response status codes are primarily divided into five groups [14]:

- Informational 1xx,
- Successful 2xx,
- Redirection 3xx,
- Client Error 4xx,
- Server Error 5xx.

HTTP fixes the bugs in FTP that make it inconvenient to use for many small transfers which are typical for web pages. FTP has a stateful control connection which maintains a current working directory, and each transfer requires a secondary connection through which data is transferred. HTTP is stateless and multiplexes control and data over a single connection from client to server on well-known port numbers, which if necessary, can trivially pass through Network Address Translation (NAT) gateways and is simple for firewalls to manage. When FTP is transferring over the data connection, the control connection is idle. If it takes too long to transfer a file, the firewall or NAT may decide that the control connection is dead and stop tracking it. This effectively breaks the connection and confuses the transfer.

On the other hand, a single HTTP connection is only idle between requests and it is normal and expected for such connections to be dropped after a time-out.

3 System Update Implementation

In this chapter we present the proposed system update application for a target device, a system update process flow and a device multiboot functionality. We start from a specification of a target device, for which the application is intended. Next we make an system-level overview, leading to a more detailed description of each component.

3.1 Target Device Description

For the system update application development purpose, we used a TE0720 module, along with TE0703 development board, both from Trenz Electronic. A main component of the TE0720 module is Xilinx Zynq-7000 SoC. Development of the application has to be done with respect to its future usability on TE0803 module based on Xilinx Zynq Ultrascale+ Multi-Processor System-on-Chip (MPSoC). This module will be a main part of the Software-Defined Radio (SDR) emulator, which is being developed by the Department of Radio Electronics of the Faculty of Electrical Engineering and Communication at the Brno University of Technology.

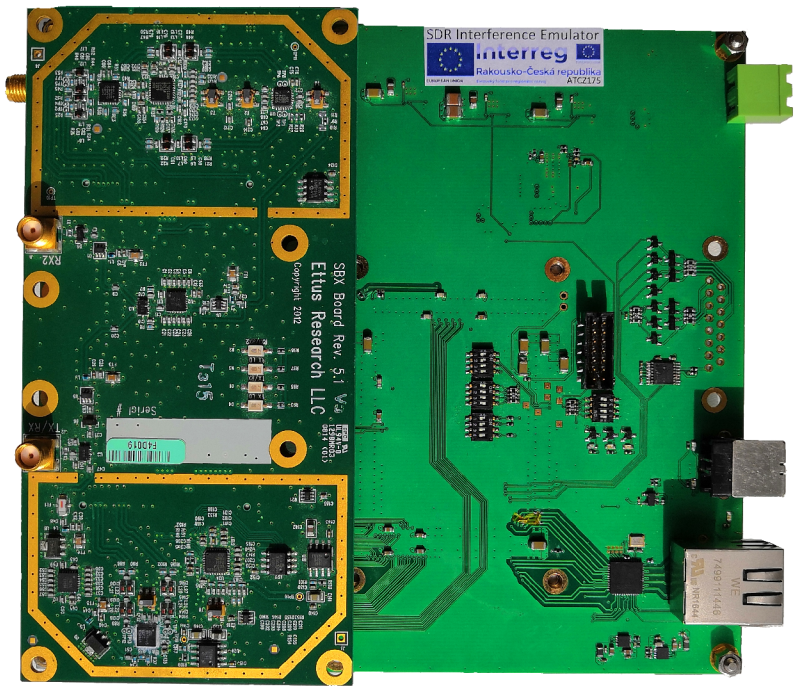

Fig. 3.1: SDR emulator - top view.

The Trenz Electronic TE0720 is an industrial-grade System-on-Module (SoM). Its key features are:

- 1 GB of DDR3 RAM,
- 32 MB of SPI flash memory,
- gigabit Ethernet PHY transciever,
- USB PHY transciever,
- switching-mode power supplies for all on-board voltages,
- configurable I/Os provided via high-speed stacking strips.

3.2 System-Level Design and Considerations

From the system-level point of view, there were three main objectives to be achieved:

1. To enable a target device to automatically select from multiple boot image versions, which are stored in its boot device. Upon a possible previous boot failure, the ability to load a backup configuration (golden boot image) must be ensured.
2. To enable the target device to receive a binary file containing a system update over a network and to decide, which of the currently present boot image versions will be updated.
3. To allow a user to manage the system update process and interact with a target device.

Although the TE0720 module is equipped with a single 32 MB SPI flash memory, the Zynq-7000 QSPI controller is limited to 3-bytes addressing, therefore the boot image can be fetched by the BootROM only if it resides in the first 16 MB of the flash memory. Using the full capacity of the flash for booting purposes would require to store FSBL (also SSBL, if present) of each boot image in the first 16 MB of the memory. The rest of the partitions, possibly residing in a portion of the memory that is beyond 16 MB would be handled by FSBL or SSBL (e.g. U-boot). This could be problematic during the system update - the appliccation would have to be able to accept and write boot images in Intel HEX file format (*.mcs), where each block of data to be written in memory has an address offset specified. Based on this facts we decided to rather reserve the first 16 MB of flash memory only for storing boot images and keep the rest upper 16 MB available for other applications. For a simple and fast localization of boot images in the memory we designed the system to use fixed address mapping, which allows to store one golden image and up to three multiboot images, each up to 4 MB in size (see Fig. 3.2).

For a remote file transfer of a boot image from a user's computer to the target device we decided to use HTTP protocol. This approach brings the user two ways

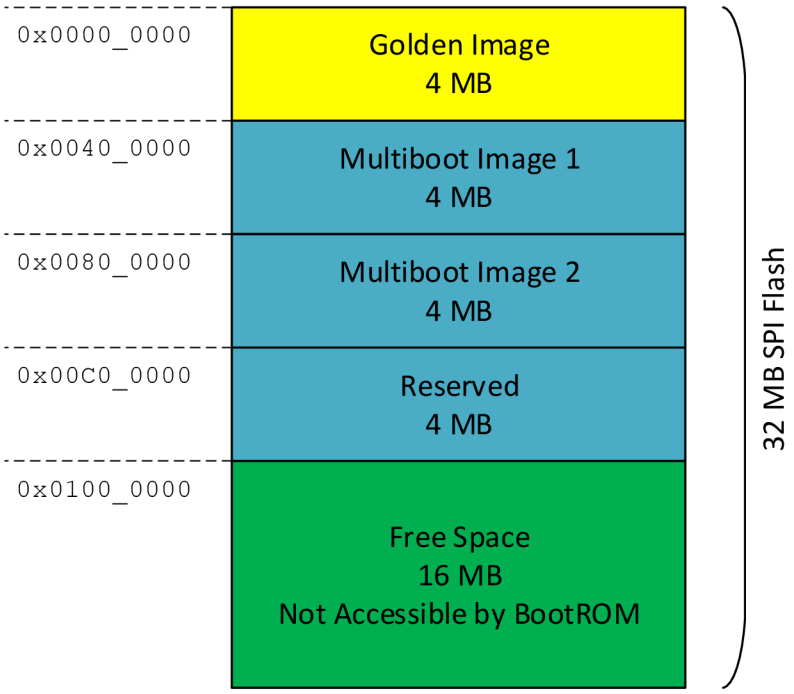

Fig. 3.2: Flash memory address map.

of managing the system update process: the first one is to use a standard command-line application for transferring data using HTTP protocol. The second way is to use the proposed system update webpage, which enables the user to choose and upload the boot image file from the web interface. The webpage code is part of the target device application itself, thus the webpage functionality can be updated along with it.

The target device runs a simple web server, which implements the application-specific subset of HTTP protocol methods. Such a web server can be also used to control or monitor the target device operation.

Upon receiving a request, the server sends a response containing the system update webpage, which is displayed in a web browser of a user's computer. Using this webpage, the user uploads the binary file containing updated boot image to the device. The device then handles the flash memory related system update operations.

To enable the target device to automatically select from multiple image versions upon boot and to decide, which one should be replaced upon system update, we developed two-criteria decision mechanism. Firstly, the device checks the boot image validity. If there are multiple valid boot images present, then the device decides on a boot image version basis - a newest valid image is loaded upon boot, and an oldest

(or invalid) image is overwritten upon system update.

To validate a boot image means to validate its individual partitions. Since the thesis assignment did not put a requirement for a secure boot, we decided to use non-secure boot only. Therefore none of partitions is RSA authenticated and AES encrypted upon boot image creation. Instead of that a md5 checksum of every partition is calculated, except for the FSBL partition. According to section , the BootROM does not support md5-validation of the FSBL partition, neither the Bootgen can calculate and include it in a boot image. For this reason we created a Python script, whose function is to calculate md5 checksum of the FSBL partition, and along with a boot image version (in form of timestamp) to write it to a UDF file. The UDF file then serves as the input for the Bootgen, which writes the UDF data in the user-defined field in the boot image header.

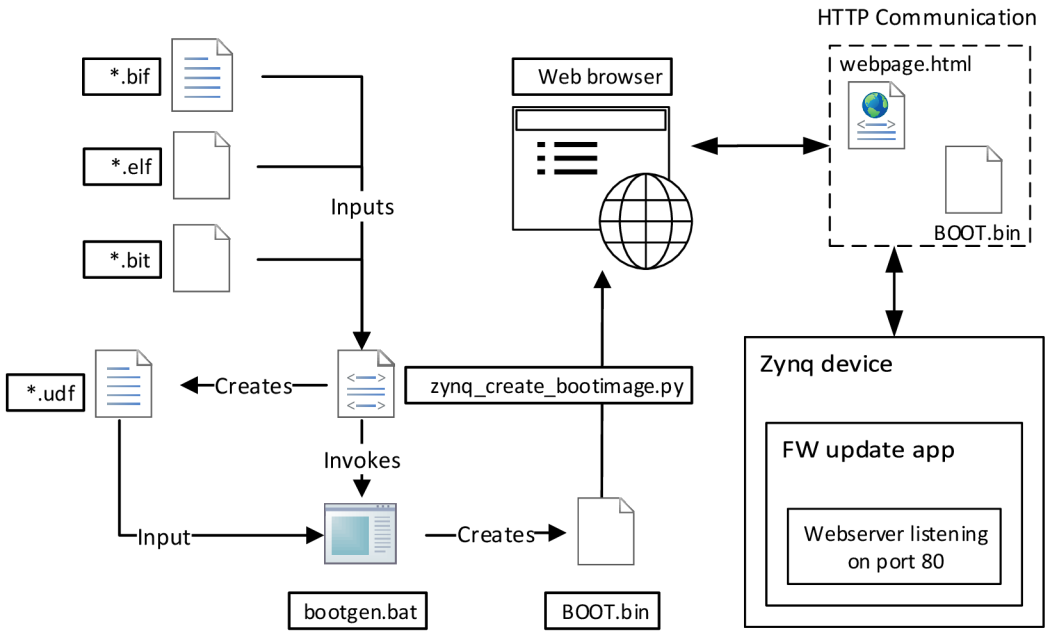

Fig. 3.3: System-level diagram of the proposed system update application.

3.3 Boot Image Building

To create a script for building boot images we had chosen the Python programming language due to its cross-platform feature and great availability of well-documented libraries and packages.

The user firstly creates the BIF file (using Bootgen GUI integrated in Xilinx SDK, or writing it manually) where he specifies all the images (FSBL, application, bitstream, etc.) to be processed into a boot image, along with their attributes and file paths which points to them (see section 1.3). Then the Python script can be executed. The script reads the input BIF file to search for the file path which points to the FSBL image ELF file. Searching is done by using regular expression operations from the package `re` [16].

According to ELF file specification[17], the script reads the ELF file header and program header to obtain the beginning of the FSBL data segment and the segment size. Then it loads the FSBL data from the file and computes its md5 checksum using the package `hashlib` [18]. Next the script gets the current timestamp using the package `datetime` [19].

The checksum, timestamp and the Image Validity Word (a value of 0xFFFFFFFF) are then written to a UDF file as a string of hexadecimal values. The UDF and BIF files, serve as inputs for the Bootgen. The script invokes the Bootgen using the `subprocess` package [20]. The whole Python script listing can be found in Appendix A.2.

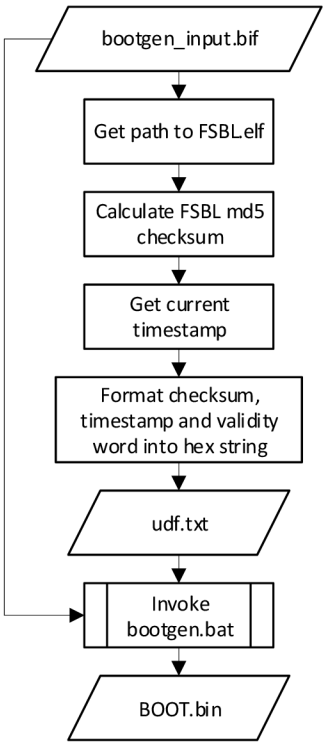

Fig. 3.4: Boot image building script flowchart.

3.4 Multiboot and Fallback Functionality

According to section 1.2.1, upon a system reset, the BootROM starts searching for a valid BootROM header (boot image header), beginning from the base address of the flash memory. Since we had designed the flash memory mapping as it is depicted on Fig. 3.2, the multiboot functionality was only needed to be implemented in FSBL of the golden image . For this purpose we modified a generic FSBL code, which is distributed along with Xilinx Software Development Kit (SDK). Flowchart of the procedures, which the target device takes during multiboot, is presented on Fig. 3.5.

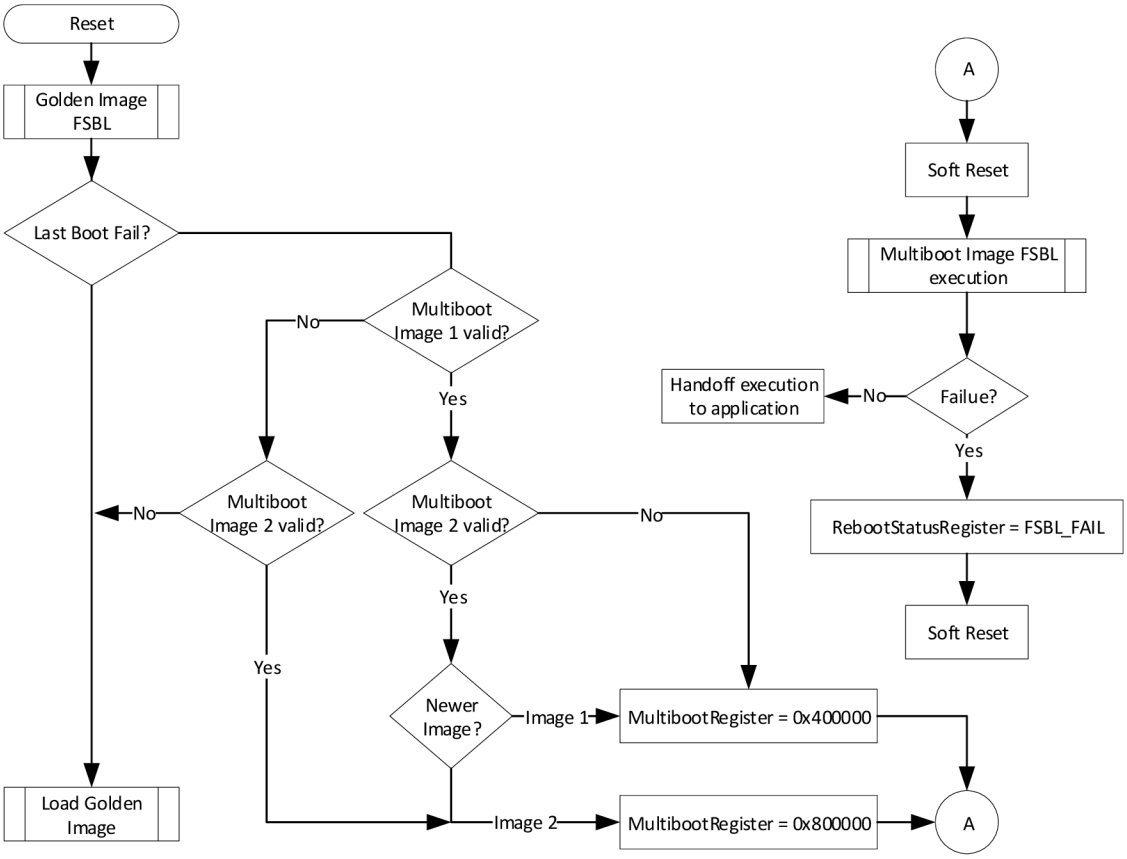

Fig. 3.5: Multiboot functionality flowchart.

After a necessary device initialization, the FSBL reads the `REBOOT_STATUS` register. If no previous boot failure is detected, FSBL continues with searching for boot images in the flash memory at predefined addresses. A valid boot image is recognized by finding:

1. a mandatory value of `0x584C4E58, 'XLNX'`, at `0x24` offset address. The value represents the Image Identification parameter, whose primary use is to allow the BootROM (along with the header checksum) to determine that a valid BootROM *header* is present.

2. a mandatory value of `0xFFFFFFFE` at `0x60` offset address. This value represents the Image Validity Word UDF parameter. Finding this value the FSBL knows, that the *whole* boot image was successfully validated after it had been written into flash memory.

If the golden image FSBL finds multiple valid boot images, it checks their version. A boot image has its version stored in the UDF section of the BootROM header, in form of a hexadecimal-encoded timestamp. Then, the `MultibootRegister` is loaded with the base address of the newest valid boot image found and a soft system reset is triggered.

Upon this reset, the BootROM loads a FSBL from the location, at which points the `MultibootRegister`. Then the FSBL handles loading of the remaining partitions from the flash memory. If the loading fails for some reason, the `FSBL_FAIL` flag in the `REBOOT_STATUS` register is set and a soft system reset is triggered again. The golden image FSBL reboots, but now it load its own partitions immediately, without searching for multiboot images.

3.5 System Update Application

The proposed system update application is intended to run on a target device as an independent task in parallel with other applications/tasks, each having distinct responsibilities and requirements. For this reason its development was based on FreeRTOS platform.

The FreeRTOS is an open source, light-weight, real-time operating system kernel for embedded devices. It provides deterministic time behaviour, priority-based multitasking capability, simple scheduler to switch between task, queues to inter-task communication, semaphores to manage resources sharing between tasks, etc.

From the FreeRTOS features also benefits the Lightweight TCP/IP (lwIP) stack, which is used in the application to enable the networking functionality. The lwIP stack is a small independent implementation if the Internet protocol suite.

Networking in general makes use of a number of timed protocols, which include timeouts, retransmissions, etc. As the FreeRTOS provides multi-threaded environment, we could use an lwIP socket mode Application Programming Interface (API), which blocks on TCP socket reads and writes until they are complete. The necessary timers are handled in the background and does not need to be implemented manually.

3.5.1 HTTP Server

Implementation of the HTTP server on the target device makes use of lwIP stack and its socket mode API. After lwIP initialization, the `network_thread`, which configures the network interface and creates a `http_server` thread. The `http_server` thread creates a new TCP socket, binded to HTTP port 80. The lwIP then starts to listen for incoming connections on this socket. Upon newly accepted connection on port 80, a `process_http_request` thread is created. In this thread, all the main web server's functionality is implemented. Flowchart of the thread is depicted in Fig. 3.6.

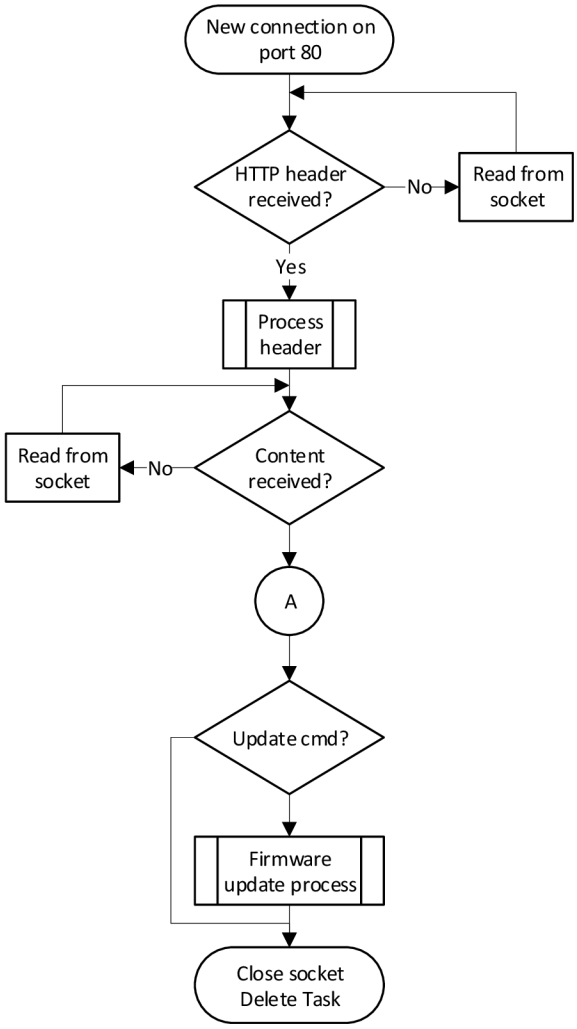

Fig. 3.6: Thread processing HTTP requests flowchart.

On a receiving a new request, a new socket descriptor is created. The server starts to read data from the socket until a whole HTTP header is received. This is done by searching for the header terminator - a sequence of `"CRLFCRLF"` ASCII

characters. Next, the header is processed to obtain the request-related information, namely:

- a HTTP method of the request,
- a client-send command to the server, in form of a URL,
- a length of a payload, if the request have some. The header contains this information in `Content-Length` field.

The server then continues receiving data from the socket until a whole payload is received. Depending on a request method, an appropriate response is generated and sand back to the client. We implemented the server to accept following HTTP methods:

- GET - a client uses this method when it requests a specific resource from the servers. The server currently provides a single resource available to a client. Therefore on every client's GET request, the server responds with the system update webpage. However, if needed in future, the server is prepared to be extended with the functionality to send responses depending on requested resource.
- POST - a client uses this method to upload a boot image file to the server and to send a specific command.
- OPTIONS - a client sends OPTIONS request to describe the communication options for the target resource. The server responds with the allowed option specified in the response header.

The server currently accepts two types of commands contained in a POST request:

- `cmd/update-multiboot`,
- `cmd/update-golden`.

When the server receives the POST request with the `cmd/update-multiboot` command, the system updates one of the multiboot images in the flash memory with the received boot image file. Likewise, if the `cmd/update-multiboot` command is received, the system will update the golden image sector. This feature is assumed to be available only for developing and debugging purposes, as usually it is not desirable to allow the end user to modify the golden image.

Upon receiving a request containing one of the supported commands, the server responds with "200 OK" status code and a message containing size of the received file. If the client sends no or unknown command, the server sends "400 Bad Request" response. Flowchart of the response generation is depicted in Fig. 3.7

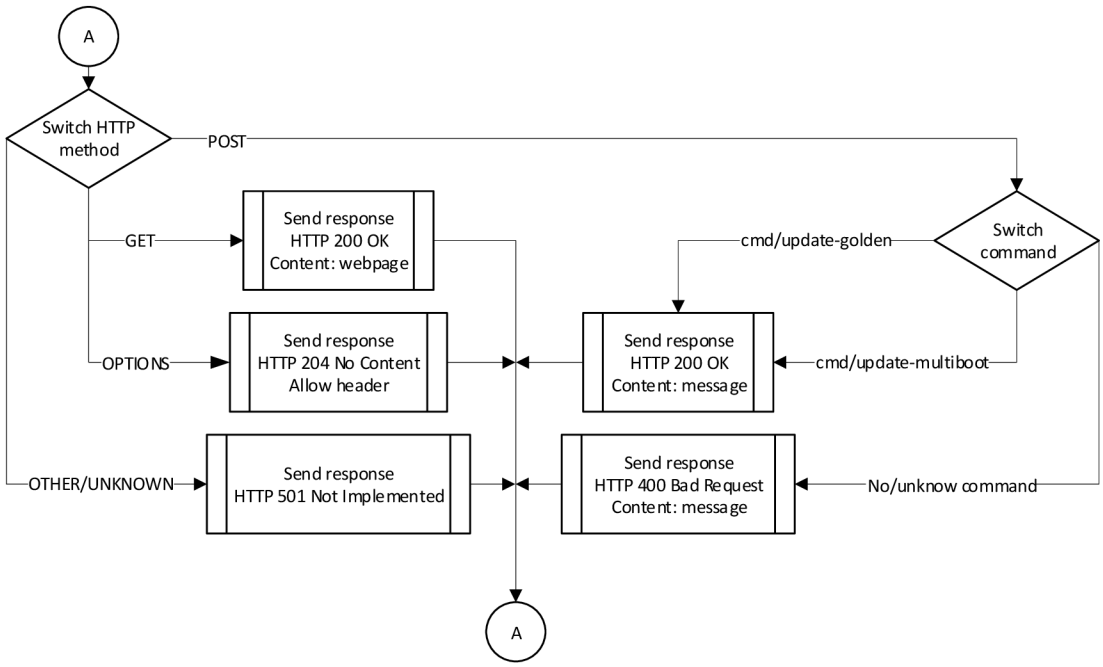

Fig. 3.7: HTTP response generation flowchart.

3.5.2 Client Command-Line Application

Since the target device is running the HTTP server, to upload a boot image file the user can choose one of many available command-line based HTTP clients, e.g. cURL [21], GNU Wget [22], or HTTPie [23]. All of the clients listed above are cross-platform and support file uploads over HTTP. As an example we describe the file upload using cURL on Windows.

cURL tool can be executed from Windows Command Prompt by entering the `curl` command using the following syntax:

```
curl [options / URLs].
```

To upload arbitrary binary data to a server, the command should be used with following options:

```
--header "Content-Type: application/octet-stream"
--data-binary @<filename>
```

By entering a command in the format above cURL builds the POST HTTP request header to which it appends the data file. Request is then send to the specified URL. The full listing of cURL command, including the client request and the server response headers, can be found in Appendix B.1.

3.5.3 Client Web Application

The proposed system update webpage makes use of Asynchronous JavaScript + eXtensible Markup Language (XML), abbreviated as Ajax. Ajax is not a programming language, but rather a set of web development techniques using multiple web technologies on the client side to create asynchronous web applications. Using Ajax, webpages are allowed to change its content dynamically, without the need to reload the entire page [24]. A principal communication model using Ajax can be found in Fig. 3.8. In the proposed web application we used following technologies:

- Hypertext Markup Language (HTML) to describe the structure of a webpage, and provide input forms, user interaction elements, etc.
- The XMLHttpRequest (XHR), which is an API in the form of an object whose methods transfer data between a web browser and a web server.
- JavaScript (JS) to bring these technologies together and to modify the webpage content.

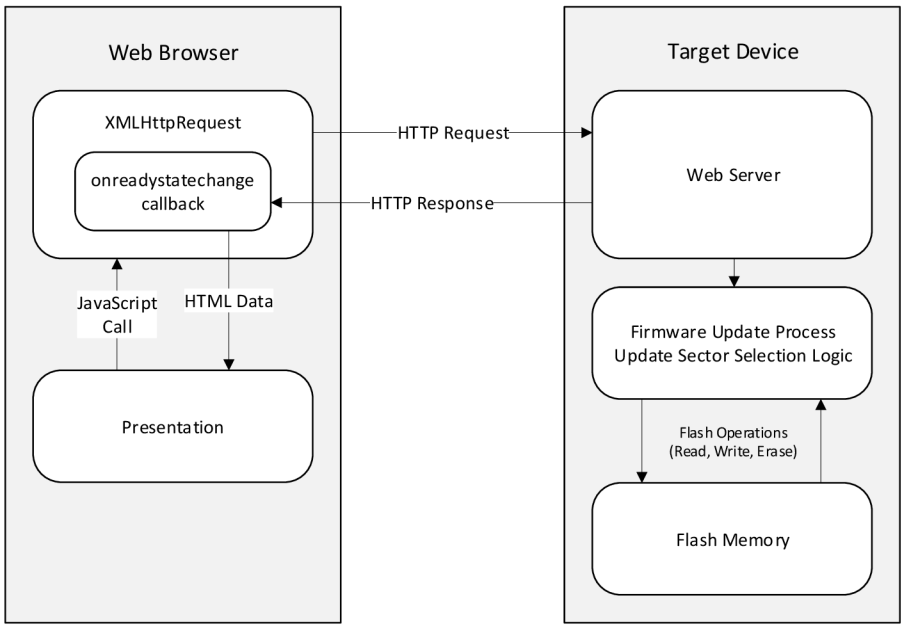

Fig. 3.8: A principal communication model using Ajax.

The webpage contains file input HTML element, which accepts files with .bin file extension, two radio buttons allowing a user to select either golden or multiboot sector in flash memory to be updated, and a submit button to start sending a file. Preview of the webpage, as it is displayed in a browser, is depicted in Fig. 3.9.

Upon pressing the submit button, the `doSubmit` JS function is called. The function obtains the data from all the input elements and then creates new XHR object. The newly-created XHR object is initialized using POST method and with

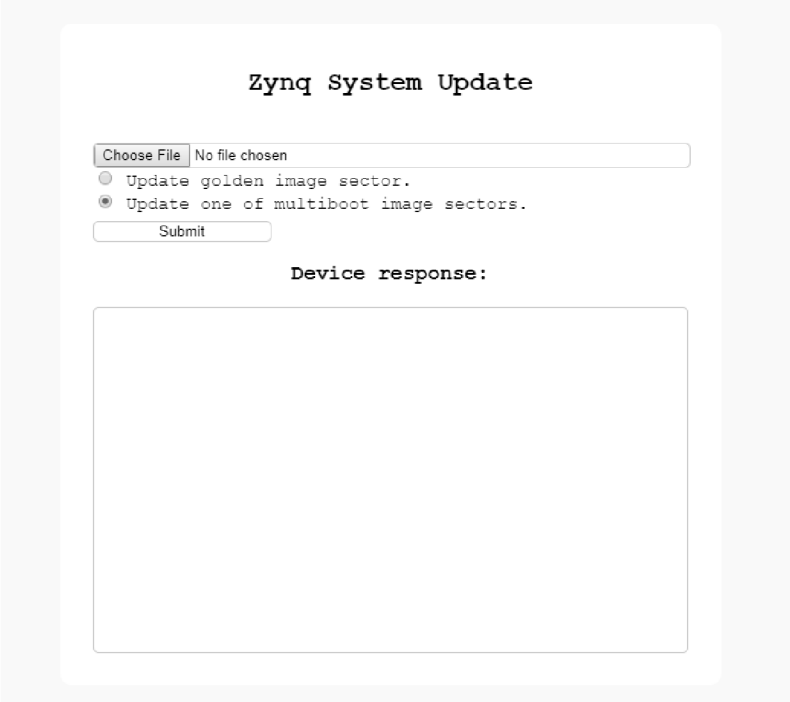

Fig. 3.9: Layout of the system update webpage.

an URL, which represents a command to be send to a server, according to selected update sector. After the XHR's `readyState` attribute indicates that the transfer is complete, yet an empty paragraph is updated with the XHR's response message. The JS code listing of the textttdoSubmit function can be found in Appendix A.1.

3.5.4 Boot Image Validation Process

Before newly-received boot image file is written to a flash memory, the system performs similar operations as it does during multiboot (see Fig. 3.5). It looks for a valid boot images present at predefined fixed addresses in the flash memory and decides which one to update and therefore overwrite. The system primarily updates an invalid image. If there are multiple valid images present in the memory, then the system update the one with a smaller timestamp and thus a lower version.

Technology of the flash memory requires erasing the flash sectors to be overwritten. For this reason and for the process simplicity, the whole 4 MB sector is erased, regardless of the actual image size. The new boot image is then written to the memory using the most effective, 256 byte page programming command [25].

To check a data integrity after the flash write operation, the system reads the data back from the flash memory and validates:

- a boot image header,

- a partition header of every partition present,
- every partition data, including FSBL.

The system therefore does nearly the same operation as the BootROM and FSBL do during boot time. The only difference is in validation of the FSBL partition, which is the BootROM incapable of in non-secure boot mode. To enable the system to validate boot images stored in flash memory we modified a generic FSBL code and created custom library adapted to the system update application needs. An example listing of a C function which wraps other function calls needed to validate the boot image can be found in Appendix A.3.

If the system successfully validates the boot image, it changes a least significant bit of a Image Validity Word in the boot image header from `1` to `0` (programming a flash memory means changing values of ones to zeros), therefore the Validity Word new value will be `0xFFFFFFFE`. Reading that value a golden image FSBL will recognize this boot image a valid so it will be eligible for boot (see section 3.4).

4 Conclusion

The system update application for Zynq-7000 device was designed and implemented. QSPI flash memory device is used as a boot device for storing multiple boot images of user application. A based storing and update mechanism was selected, without the use of a file system. That implies that all boot images are limited in size to maximum of 4 MB.

The system counts on functionality of a golden image, whose bootloader code provides both, an automatic boot image selection upon boot time (multiboot) and recovery loading of backup configuration upon boot failure (fallback). The boot image selection/update decision mechanism is based on calculating checksums and comparing timestamps of the images. The system is designed to always try to load the newest valid boot image and to update the oldest or invalid one.

The system update process is handled by a FreeRTOS and lwIP stack based application, which runs on a target device. The main component of the application is a web server, which implements a subset of HTTP protocol for communication with the application running on a client side. The client side is represented either by one of freely available command line tool (e.g. cURL) which serves as HTTP client, or by a web page which runs a simple web application for transmitting the boot image binary file and displaying the server response.

Writing a newly-received boot image file in flash memory also provides the target device application, which make use of customized library for accessing flash memory data and perform validation operations over them.

For creation the customized boot images files it was developed a Python script which processes the input files and invokes the Bootgen utility.

Our proposal for further development is to:

- Implement a file system-based storing the boot images in the boot device, including flash memories larger than 16 MB. Make a support for other boot device types.
- Extend the functionality of HTTP server with HTML Server-Sent Event technology enabling a client to receive automatic updates from a server (e.g. system update process progress, system status, etc.).
- Consider some of the secure boot mode options, mainly the RSA authentication for boot for better boot image validation.
- Make port of the system for Xilinx Ultrascale+ MPSoC platform.

Bibliography

[1] System Update In: *Yocto Project Wiki* [online]. 20187 [cit. 2019-05-14]. Available: <https://wiki.yoctoproject.org/wiki/Main_Page>

[2] SANDERS, Lester. *Updating a System Securely in the Zynq-7000 AP SoC* [online]. V1.1 Xilinx, 2014, last modified February 12, 2015 [cit. 2019-05-14]. XAPP1224. Available: <https://www.xilinx.com/support/documentation/application_notes/xapp1224-secure-system-update.pdf>.

[3] XILINX. *Zynq-7000 SoC: Technical Reference Manual* [online]. V1.12.2. 2012, last modified July 1, 2018 [cit. 2019-05-14]. UG585. Available: <https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf>.

[4] XILINX. *Zynq-7000 All Programmable SoC Software Developers Guide* [online]. V12.0. 2014, last modified September 30, 2015 [cit. 2019-05-14]. UG821. Available: <https://www.xilinx.com/support/documentation/user_guides/ug821-zynq-7000-swdev.pdf>.

[5] *Das U-Boot – the Universal Boot Loader.* [online]. Gröbenzell (DE): DENX Software Engineering, 2017 [cit. 2019-05-14]. Available: <https://www.denx.de/wiki/U-Boot/>

[6] XILINX. *Bootgen User Guide* [online]. V2018.2. 2018 [cit. 2019-05-14]. UG1283. Available: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1283-bootgen-user-guide.pdf>

[7] SARANGI, Anirudha, Stephen MACMAHON and Upender CHERUKUPALY. *LightWeight IP Application Examples* [online]. v5.1. 2008, last modification November 21, 2014 [cit. 2019-05-14]. XAPP1026. Available: <https://www.xilinx.com/support/documentation/application_notes/xapp1026.pdf>.

[8] Comparison of file transfer protocols. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2019-05-14]. Available: <https://en.wikipedia.org/wiki/Comparison_of_file_transfer_protocols>

[9] BHUSHAN, A.K. BERNERS-LEE. *File Transfer Protocol* [online]. 1971 [cit. 2019-05-14]. RFC 114. Available: <https://www.rfc-editor.org/info/rfc114>.

[10] SOLLINS, K. BERNERS-LEE. *The TFTP Protocol (Revision 2)* [online]. 1992 [cit. 2019-05-14]. RFC 1350. Available: <https://www.rfc-editor.org/info/rfc1350>.

[11] FINLAYSON, R. *Bootstrap loading using TFTP* [online]. 1984 [cit. 2019-05-14]. RFC 906. Available: <https://www.rfc-editor.org/info/rfc906>.

[12] YLONEN, T. a C. LONVICK. *The Secure Shell (SSH) Protocol Architecture* [online]. 2006 [cit. 2019-05-14]. RFC 4251. Available: <http://www.rfc-editor.org/info/rfc4251>.

[13] SCP vs SFTP: Which One Should You Use for File Transfer In: *Make Tech Easier - Computer Tutorials, Tips and Tricks* [online]. Uqnic Network Pte, c2007-2019 [cit. 2019-05-14]. Available: <https://www.maketecheasier.com/scp-vs-sftp/>

[14] FIELDING, R., J. GETTYS, J. MOGUL, H. FRYSTYK, L. MASINTER, P. LEACH a T. BERNERS-LEE. *Hypertext Transfer Protocol – HTTP/1.1* [online]. RFC 2068. 1999 [cit. 2019-05-14]. Available: <https://www.rfc-editor.org/info/rfc2616>.

[15] *TE0720 TRM* [online]. v.85. 2015, last modified November 10, 2017 [cit. 2019-05-14]. Available: <https://www.trenz-electronic.de/fileadmin/docs/Trenz_Electronic/Modules_and_Module_Carriers/4x5/TE0720/REV03/Documents/TRM-TE0720-03.pdf>.

[16] Re - Regular expression operations. In: *Python 3.7.3 documentation* [online]. Wilmington (USA): Python Software Foundation, c2001-2019 [cit. 2019-05-14]. Available: <https://docs.python.org/3/library/re.html>

[17] TIS COMMITTEE. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification* [online]. Version 1.2 1995 [cit. 2019-05-14]. Available: <https://refspecs.linuxfoundation.org/elf/elf.pdf>.

[18] hashlib - Secure hashes and message digests. In: *Python 3.7.3 documentation* [online]. Wilmington (USA): Python Software Foundation, c2001-2019 [cit. 2019-05-14]. Available: <https://docs.python.org/3.4/library/hashlib.htmlto>

[19] datetime - Basic date and time types. In: *Python 3.7.3 documentation* [online]. Wilmington (USA): Python Software Foundation, c2001-2019 [cit. 2019-05-14]. Available: <https://docs.python.org/3/library/datetime.html>

[20] subprocess - Subprocess management. In: *Python 3.7.3 documentation* [online]. Wilmington (USA): Python Software Foundation, c2001-2019 [cit. 2019-05-14]. Available: <`https://docs.python.org/3/library/subprocess.html`>

[21] *Curl: command line tool and library for transferring data with URLs* [online]. Daniel Stenberg and contributors, c1996-2019 [cit. 2019-05-14]. Available: <`https://curl.haxx.se/`>

[22] *GNU Wget* [online]. Boston (USA): Free Software Foundation, 2017 [cit. 2019-05-14]. Available: <`https://www.gnu.org/software/wget/`>

[23] *HTTPie - command line HTTP client* [online]. Jakub Roztočil [cit. 2019-05-14]. Available: <`https://httpie.org/`>

[24] AJAX Introduction In: *W3Schools Online Web Tutorials* [online]. Sandnes (NO): Refsnes Data, c1999-2019 [cit. 2019-05-14]. Available: <`https://www.w3schools.com/xml/ajax_intro.asp`>

[25] CYPRESS SEMICONDUCTOR CORPORATION. In: *128 Mb (16 MB)/256 Mb (32 MB) 3.0V SPI Flash Memory Datasheet* [online]. Rev. *P. San Jose (CA): Cypress Semiconductor Corporation, 2011, Revised August 07, 2018 [cit. 2019-05-16]. Available: <`https://www.cypress.com/file/448601/download`>

List of symbols, physical constants and abbreviations

ADC	Analogue to Digital Converter
AES	Advanced Encryption Standard
Ajax	Asynchronous JavaScript + XML
API	Application Programming Interface
APU	Application Processor Unit
BBRAM	Battery-Backed RAM
CAN	Controller Area Network
CLB	Configurable Logic Block
CPU	Central Processing Unit
CR	Carriage Return
DDR	Double Data Rate
DSP	Digital Signal Processing
ELF	Executable and Linking Format
FPU	Floating-Point Unit
FSBL	First-Stage Bootloader
FTP	File Traansfer Protocol
GPIO	General-Purpose Input/Output
GUI	Graphical User Interface
HPC	High Performance Computing
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
I2C	Inter-Integrated Circuit
I/O	Input/Output
JS	JavaScript
LAN	Local Area Network
LF	Line Feed
LUT	Lookup Table
lwIP	Lightweight TCP/IP
MMU	Memory Management Unit
MPSoC	Multi-Processor System-on-Chip
NAT	Network Address Translation
OCM	On-Chip Memory
PL	Programmable Logic
PLL	Phase-Locked Loop
PS	Proessing System

POR	Power-On Reset
QSPI	Quad Serial Peripheral Interface
RAM	Random Access Memory
SBC	Single-Board Computer
SCP	Secure Copy Protocol
SDK	Software Development Kit
SDR	Software-Defined Radio
SFTP	Secure File Transfer Protocol
SHA	Secure Hash Algorithm
SoC	System-on-Chip
SoM	System-on-Module
SSBL	Second-Stage Bootloader
SSH	Secure Shell
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TFTP	Trivial File Transfer Protocol
U-boot	the Universal Boot Loader
UART	Universal Asynchronous Receiver-Transmitter
UDF	User Defined Field
UDP	Transmission Layer Protocol
URL	Uniform Resource Locator
USB	Universal Serial Bus
XML	eXtensible Markup Language
XHR	XMLHttpRequest

List of appendices

A Code Listings

A.1 JavaScript Function to Transfer a Boot Image

```
<script type="text/javascript">

    function doSubmit() {

    /* Specify server URL */
    const serverUrl = "http://192.168.1.10/";

    var fileSelect = document.getElementById('file-select');
    var updateSector = document.querySelector('input[name='
      + '"sector-radio"]:checked').value;
    var textArea = document.getElementById("server-response");
    var formData = new FormData();

    /* Create new XMLHttpRequest object. */
    var xhttp = new XMLHttpRequest();

    /* Obtain a file from the file input element */
    if(fileSelect.files & fileSelect.files.length == 1) {
      var file = fileSelect.files[0];
      formData.set("file", file , file.name);
    };

   /*
    * An event handler that is called whenever the readyState
    * attribute changes. The callback function updates the
    * content of the textbox element with XHR response text
    * when the data transfer is completed.
    */
    xhttp.onreadystatechange = function() {
      if (this.readyState == 4) {
        textArea.innerHTML = this.responseText + "\n" +
        textArea.innerHTML;
      };
    };

   /*
    * Build the URL according to selected sector to be
    * updated.
    */
    switch (updateSector) {
      case "golden":
        var url = serverUrl.concat("cmd/update-golden");
```

```
        break;
      default:
        var url = serverUrl.concat("cmd/update-multiboot");
        break;
    };

   /*
    * Initialize the XHR. Set the MIME type of the XHR to
    * application/octet-stream. Send the request to a server.
    */
    xhttp.open("POST", url, true);
    xhttp.setRequestHeader("Content-Type",
      "application/octet-stream");
    xhttp.send(file);
    };
</script>
```

A.2 Python Script to Build a Boot Image

```
import re, os, subprocess, datetime, hashlib, struct

# Boot image header fields definitions.
BOOTEOM_HDR_TIMESTAMP_OFFSET = 0x4C
BOOTROM_HDR_FSBL_CHECKSUM_OFFSET = 0x50

# ELF file header fields definitions.
ELF_FHDR_PHDR_PTR_OFFSET = 0x1C
ELF_PHDR_SEG_PTR_OFFSET = 0x04
ELF_PHDR_SEG_SIZE_OFFSET = 0x10

# Regular expression to match the FSBL ELF file path.
pattern = re.compile(r'(?<=\s\[bootloader\])(\w:)'
                     r'([\\|/]\S+[\\|/])(\S+.(elf))')

# Search for FSBL ELF file path in bootgen_input.bif using regex
with open('bootgen_input.bif', 'r') as file:
    fsbl_path = re.search(pattern, file.read()).group()

with open(fsbl_path, 'rb') as file:
    # Read program header table's file offset in bytes.
    file.seek(ELF_FHDR_PHDR_PTR_OFFSET)
    e_phoff = int.from_bytes(file.read(4), byteorder='little')

    # Read the beginning of the data segment offset in bytes.
    file.seek(e_phoff + ELF_PHDR_SEG_PTR_OFFSET)
    p_offset = int.from_bytes(file.read(4), byteorder='little')
```

```
    # Read the number of bytes of the data segment.
    file.seek(e_phoff + ELF_PHDR_SEG_SIZE_OFFSET)
    p_filesz = int.from_bytes(file.read(4), byteorder='little')

    # Read the FSBK data from the ELF.
    file.seek(p_offset)
    fsbl_data = file.read(p_filesz)

# Calculate the md5 checksum of the FSBL data.
fsbl_checksum = hashlib.md5(fsbl_data)

# Get current timestamp.
timestamp = datetime.datetime.now()

# Convert endianness to little-endian and format into hex string.
timestamp_hex_str = format(struct.unpack('<I',
    struct.pack('>I', int(timestamp.timestamp())))[0], '08x')

# Write the FSBL checksum, timestamp, and Image Validity Word
# to UDF file.
with open('udf.txt', 'w') as file:
    file.write(timestamp_hex_str)
    file.write('{:032x}'.format(int.from_bytes(
        fsbl_checksum.digest(), byteorder='big')))
    file.write('ffffffff')
    file.close()

# Specify a file name of the boot image.
filename = 'BOOT4.bin'

# Specify a file path to the bootgen.bat
bootgen_path = r'D:\Xilinx\SDK\2018.2\bin\bootgen.bat'

# Specify arguments to be passed to the bootgen.bat
# -image <filename>[.bif] - specifies the input BIF file.
# -arch [zynq | zynqmp]   - Xilinx architecture
# -o <filename>           - specifies  the output BIN file.
#                           Use no or .bin extension.
# -w [on | off]           - overwrite mode
bootgen_args = r'-image {cwd}\bootgen_input.bif ' \
               r'-arch zynq ' \
               r'-o {cwd}\{filename} ' \
               r'-w on'.format(cwd=os.getcwd(), filename=filename)

# Invoke the bootgen.bat to build a boot image.
subprocess.run([bootgen_path, bootgen_args])
```

A.3 C Function for Boot Image Valiadtion

```
int FlashValidateBootImage (const u32 FlashAddr)
{
  u32 Status;
  u32 PartitionCount;
  u32 PartitionNum;

  PartHeader PartitionHeader[MAX_PARTITION_NUMBER];
  BootRomHeader_t BootRomHeader;
  u32 ImageValidityWord[1] = {IMAGE_VALID};
  u8 Header[0x68];

  /*
   * Get the total number of partitions in a boot image by obtaining
   * the totoal number of images from a image header table.
   * Program then goes through all the images, counting the
   * corresponding partitions.
   */
  PartitionCount = FlashGetPartitionCount(FlashAddr);

  /* Read a boot image header */
  FlashGetBootRomHeader(FlashAddr, &BootRomHeader);

  /*
   * Read a partition header of every partition in the boot
   * image
   */
  FlashGetPartitionHeaderInfo(FlashAddr, PartitionCount,
    PartitionHeader);

  /*
   * Go through all the partition, validate the partition
   * header If the partition header is valid, validate
   * partition data. A special function is used to validate
   * a FSBL partition because a FSBL partition checksum
   * is stored in UDF of the boot image header, unlike
   * a common partitions checksums, which are stored at
   * the end of the boot image file.
   */
    for (PartitionNum = 0; PartitionNum < PartitionCount;
    PartitionNum++) {

    Status = ValidatePartitionHeader(&PartitionHeader[PartitionNum]);
    if (Status != XST_SUCCESS)
      return XST_FAILURE;
```

```
        if (PartitionNum == 0) {
          Status = FlashValidateFsblPartition(FlashAddr, &PartitionHeader[
        PartitionNum]);
        } else {
          Status = FlashValidatePartition(FlashAddr, &PartitionHeader[
        PartitionNum]);
        }
        if (Status != XST_SUCCESS)
          return XST_FAILURE;

        /*
         * Write 0xfffffffe at 0x60 offset address in flash to mark the
         * boot image valid.
         */
        FlashWriteWord(FlashAddr + IMAGE_VALIDITY_WORD_OFFSET, IMAGE_VALID)
        ;
}
        return XST_SUCCESS;
```

B Command Prompt Listings

B.1 cURL Client Command and Server Response

```
C:\update\>curl --verbose --header "Content-Type:application/octet-
    stream" --data-binary @BOOT.bin http://192.168.1.10/cmd/update-
    multiboot

* Trying 192.168.1.10...
* TCP_NODELAY set
* Connected to 192.168.1.10 (192.168.1.10) port 80 (#0)
> POST /cmd/update-multiboot HTTP/1.1
> Host: 192.168.1.10
> User-Agent: curl/7.55.1
> Accept: */*
> Content-Type:application/octet-stream
> Content-Length: 153488
> Expect: 100-continue
>
* Done waiting for 100-continue
* We are completely uploaded and fine
< HTTP/1.1 200 OK
< Access-Control-Allow-Origin: null
< Cache-Control: no-cache
< Content-Type: text/plain
< Content-length: 122
< Connection: close
<
[Web Server] 200 OK: Received boot image of size 153488 bytes. One
    of multiboot firmware sectors in flash will be updated.
* Closing connection 0>
```