In-Application Programming

BOOTLOADER USING ETHERNET AND TCP

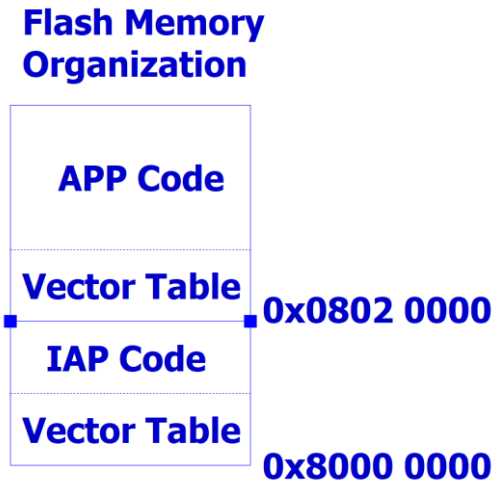
Ataberk ÖKLÜ
KUARTIS

Table of Contents

Motivation

There are various methods to re-program the STM32 MCUs [1]. The most common way is to use internal bootloader, which is located in System Memory (probably before 0x20000000). Internal bootloader allows using different interfaces like I2C, USART, CAN, DFU, SPI to handling re-programming process. If the desired peripheral is not compatible with the internal bootloader, the other method is to use In-Application Programming (IAP) [2].

The In-Application Programming method is requiring new bootloader software in flash memory to handle re-programming. The work principle of this method is to reserve a portion of the Flash Memory to use there for IAP, then after handling request, is jumping to where the main application is located in the Flash Memory. A minimalist and simply approach to divide a flash memory can be represented as:

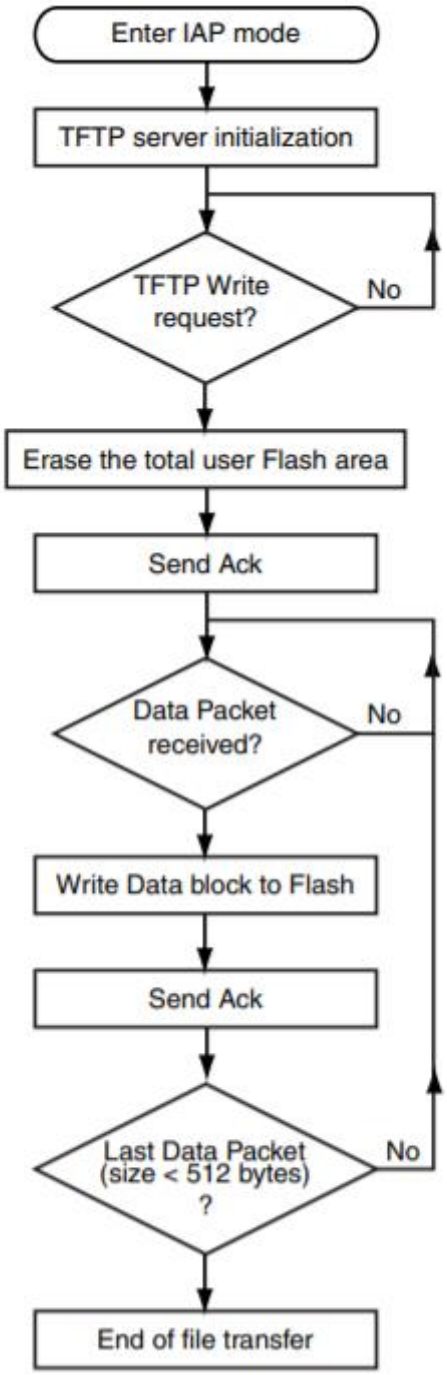


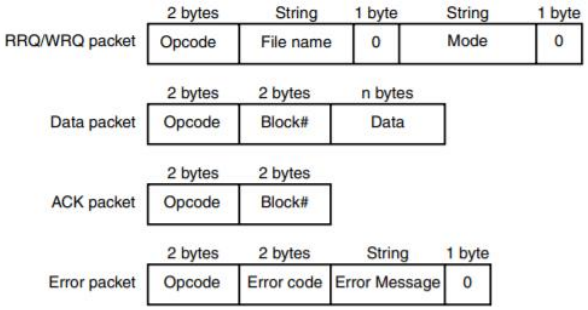
The very beginning of the flash memory is reserved for IAP Code and its interruption vector table. The 0x8000 0000 address is important, since the Reset_handler inside the startup assembly file, refers there as main user program. The beginning of the APP Code is determined by whom writes the bootloader, considering the AIP Code size. In this configuration, backup option is not available. Hence, in the case of AIP Mode trigger is only depended to APP Code, any wrong or corrupted code write attempt can turn the MCU a brick, requiring full memory erase and re-programming using internal bootloader.

IAP using Ethernet allows using both TFTP (Trivial File Transfer Protocol) and HTTP (Hyper Text Transfer Protocol), FTP (File Transfer Protocol). TFTP is the most common approach among them. STM32F4 series support the LwIP, which is light-weighted implementation of the TCP/IP protocol. TFTP lies on the UDP transport layer whereas the HTTP method works on TCP [3]. For the project, a new method which is on top of raw TCP server is going to be implemented.

TFTP Implementation

An example of TFTP Client algorithm and TFTP data package format are shown below. The triggering method for IAP checked.





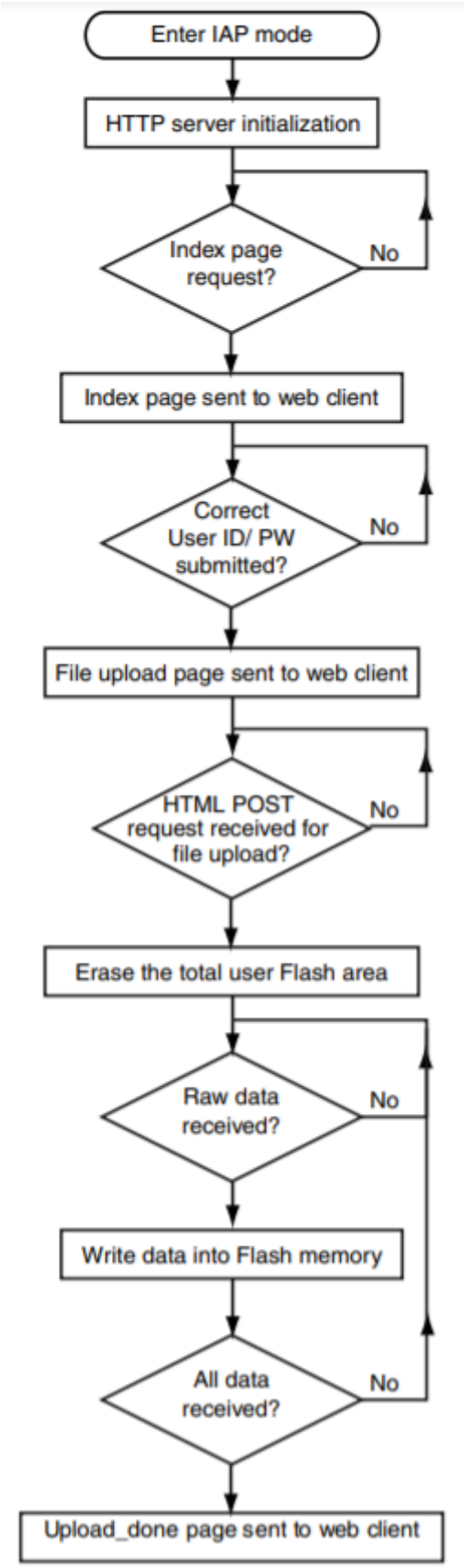
TFTP packages begins with opcode referring to the operation:

Opcode	Operation
0x1	Read request (RRQ)
0x2	Write request (WRQ)
0x3	Data
0x4	Acknowledgment (ACK)
0x5	Error

"User Flash Area" refers to where the APP Code is located in Flash Memory.

HTTP Implementation

HTTP method requires more memory then TFTP method, because of using more complicated protocol. In addition, Web Pages should be included in the code as raw byte blocks.



A simple algorithm for HTTP implementation is shown on the left. Server initializations make the MCU publish web pages to the network it is belong to. Either static IP address or DHCP can be used for IP assignment.

Introduction

The main motivation of this project is to manage and reprogram partitions in the Flash memory. In this project, the beginning of the flash memory is allocated for the partition table, containing boots' addresses related to the corresponding boot section. This is followed by In-Application Programming* – bootloader code to handle programming requests. The remaining sectors in the flash memory* are allocated for factory reset code and different application programs.

The intended working environment is STM32F746ZG Nucleo board*, using Raw LwIP API* without RTOS*, communication over ethernet, and bootloader application at TCP layer*. The preferred software IDE is STM32CubeIDE, and STM32CubeProgrammer to monitor flash memory in a convenient way. The TCP communication is listened through WireShark software for validations and debugging. Bootloader application is configured to support both static IP and DHCP server.

Challenge

In this project, only LwIP Raw API [4] and TCP are used. HTTP, TFTP protocols and FreeRTOS middleware are excluded from the project scope.

STM32F746 Configurations

The default settings are based on predetermined settings of STM32F746ZG [5] Nucleo board. The following configurations are the most plain setting to operate as bootloader application. The settings that are changed from default is marked with (*).

System Core - Cortex M7 Configuration

In this project, Flash interface is preferred as AXI, with all other features are off. MPU is not used.

System Core – GPIO

To trigger bootloader session, built-in button is used.

System Core – NVIC

Make sure that Time Base SysTick Interrupt has higher priority than Ethernet Interrupts, if activated. (Default: OFF).

System Core – SYS

Serial Wire debug is activated.

Connectivity – Ethernet Configurations

The mode is RMII with 100 Mbits/s Full Dublex configuration. The external PHY IC is LAN8742A.

Middleware – LwIP *

LwIP is light-weighted IP API to handle various types of communication protocols in embedded systems. TCP, UDP, HTTPD, MDNS/TFTP, SNTP/SMTP, SNMP, PPP are supported protocols.

General Settings

DHCP, TCP, UDP, and RTOS settings are located in this portion. DHCP, TCP modules are enabled. MEMP_NUM_TCP_PCB is the number of simultaneous open TCP connections, leaved as default (5).

Key Options

This portion contains various parameters having effects on the connection behavior and capabilities. The important parameters are elaborated.

MEM_SIZE: 1600 Bytes

Heap Memory Size. Related with TCP Window, TCP_WND.

MEMP_NUM_PBUF: 16

PBUF_POOL_SIZE: 16

The maximum number of Pbufs chained in Pbuf structure.

PBUF_POOL_BUFSIZE: 592 Bytes

The size of each pbuf.

LWIP_ARP: Enabled

Allows handling ARP packages.

TCP_WND: 2144 Bytes

The maximum receive window.

TCP_MSS: 536 Bytes

Maximum Segment size which is upper limit for receiving and transmitting packages. This limit is acknowledged by both sides while opening a connection.

*TCP_SND_BUF: TCP_WND / 2 : 1072 Bytes > 2 * TCP_MSS*

TCP send buffer size.

Clock Configurations

HSE 8 MHz bypass clock source fed to PLL to operate SYSCLCK at 192 MHz.

Bootloader

The main bootloader program is responsible for four major tasks, namely, Ethernet connection handler, TCP server to handle requests, Flash management, and Partition Table Handler. Each major task has their own c / header file pair, app_ethernet.c/.h, tcp_server.c/.h, flash_if.c/.h, and partition_table_handler.c/.h, respectively. The characteristic applications configurations are located in main.h file. Usage of DHCP, server port, flash partitions, and static IP settings are configurable.

Code Construction 1: App Ethernet

App_ethernet.c/.h files are based on example codes located in SMT32F7 Cube Repository. This task has user informing routine and necessary DHCP client functions. Also requires BSP_Nucleo_144 library to function basic LED operations. In main initiation process, ethernetif_set_link and User_notification functions are called. And to handle DHCP timing, in while routine, DHCP_Periodic_Handle is called.

The MX_LWIP_INIT function does not support static IP connection after selecting DHCP, therefore, it should be replaced by custom function Netif_Config, defined at the bottom of the main.c.

Code Construction 2: TCP Server

TCP server is required to handle incoming requests, in this case they are TCP packages carrying the application code to be written to the flash memory. TCP server consists of initiation of the server, accepting connections, receiving packages and necessary callbacks.

The server initiation – tcp_server_init, should be called in main after ethernet_set_link to establish proper server. In main.h, there is SERVER_PORT parameter to set communication port.

Code Construction 3: Flash Memory Management

In order to write application code to the flash memory, it is required to use HAL functions and necessary configurations to operate this process properly. Flash_if.c/.h files contains initiation, erase and write processes. An important setting to be manually set is the sector and number of sectors to be erased according to the flash partition management.

Code Construction 4: Partition Table Handler

Management of the partition table and boot configurations are gathered into partition_table_handler.c /.h file. One OTA boot mask and partition table created to allow boot programming process use them for shaping new boot configurations and updating the partition table. In the case of any error occurs in updating this empty boot mask will not cause any change in partition table. In the scope of this project, a partition table formation of holding only two boot configurations is created. Since this application more suitable for page-organized flash memories, partition table is placed in a way that no conflict happens in erasing process, however for further purposes, one prototype function is reserved for the sector-organized flash memory case implementation.

OTA Structure

Each Boot has its own boot configurations for identification.

- BootID := Either index or member id of related Boot.
- Address := The address of the actual boot data is located in Flash Memory.
- Size := The total number of bytes the boot has.
- MD5 := The MD5 checksum value obtained from the client. It may be used to secondary check in controller side.

```
typedef struct OTA {
	uint8_t		BootID;
	uint32_t	Address;
	uint32_t	Size;
	uint8_t		MD5[16];
} OTA ;
```

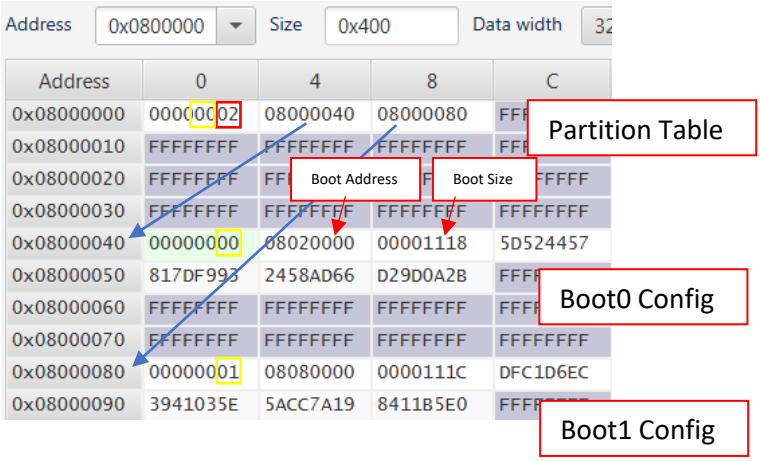
Partition Table Structure

In this proof-of-concept project, partition table is fixed to have two boots in memory other than bootloader itself. This structure is an only simple reflection of the boot management idea. Either linked or OTA* array (OTA**) structures can be used. On the other hand, following structure is preferred to allow the reader to cooperate and understand better.

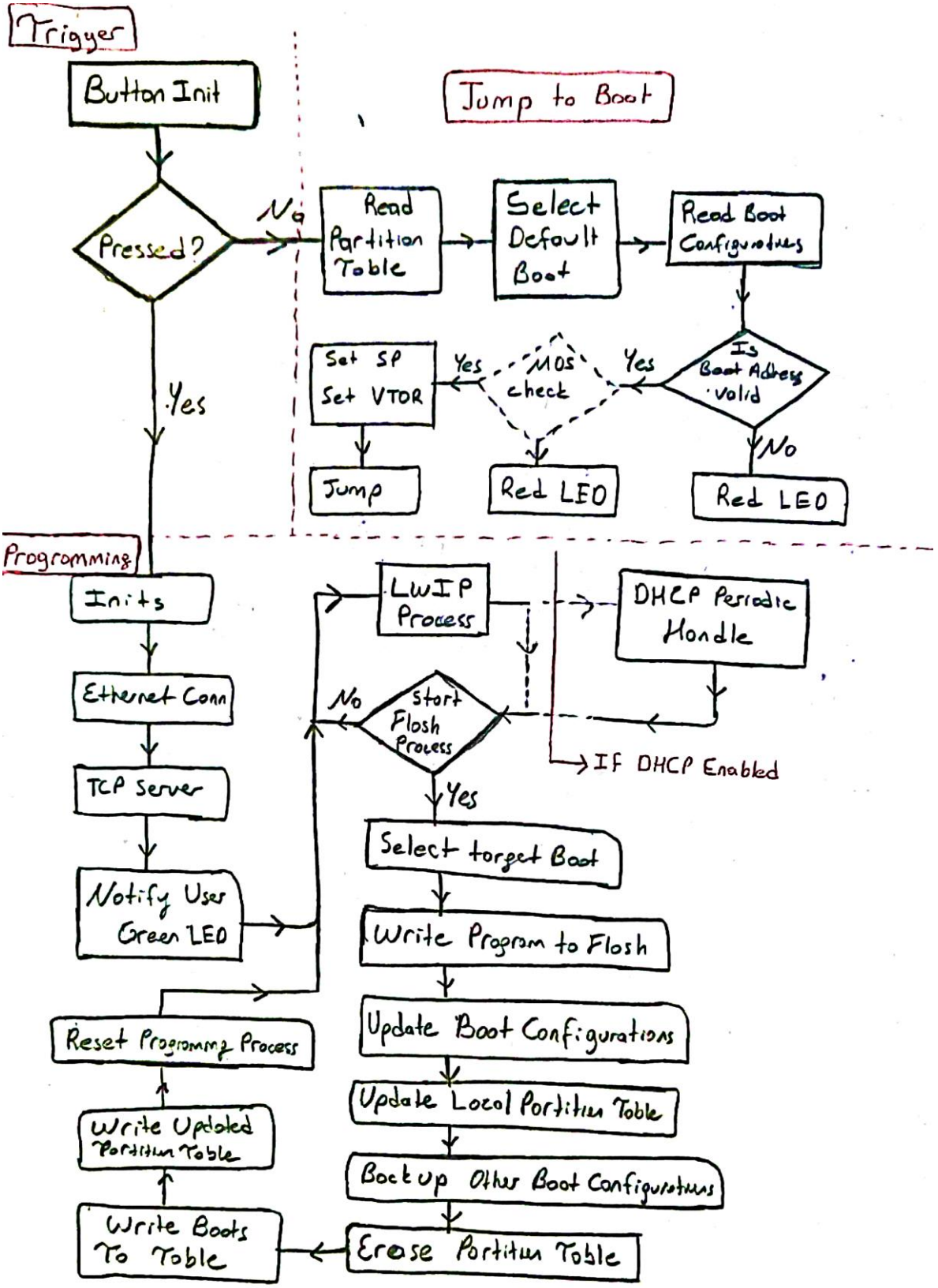
```
typedef struct PartitionTable{
	uint8_t		NumOfBoots;
	int8_t		selectedOTA;
	OTA*		boot0;
	OTA*		boot1;
} PartitionTable;
```

- NumOfBoots := 2 (Fixed)
- selectedOTA := The last updated boot's ID
- boot0 and boot1 := The locations of boot configurations in Flash Memory.

Partition Table in Flash Memory



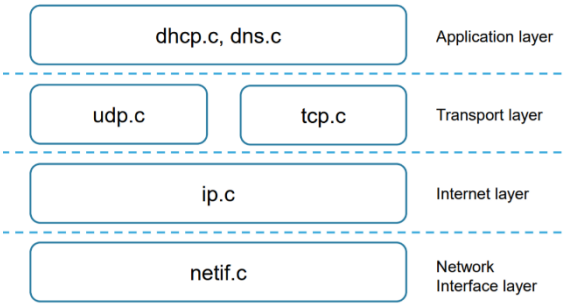
Main Bootloader Flowchart



LwIP API

The reference used for this section is attached along the report.

LwIP Architecture



LwIP provides useful structures along many layers. The bootloader project is top on the TCP Layer.

LwIP APIs

	RAW API	Netconn / Socket API
RTOS	No need	Need
Control based on	Pcb	socket
Calling methods	Callback	Close to the windows or Linux socket APIs
Structure	Core APIs	Higher level APIs
Application	➢ Lower memory devices ➢ Application without RTOS ➢ Developers has more control	➢ Higher memory devices ➢ Porting of protocols or application coming from Linux/windows
complexity	++	+
Memory	--	+

In this project, Raw API is preferred since RTOS is out of the scope of the project. The main advantage is the requirement of much less memory compared to Netconn and socket API. However, the construction and implementation of the Raw API is more complex than Netconn / Socket API.

LwIP Server Setup

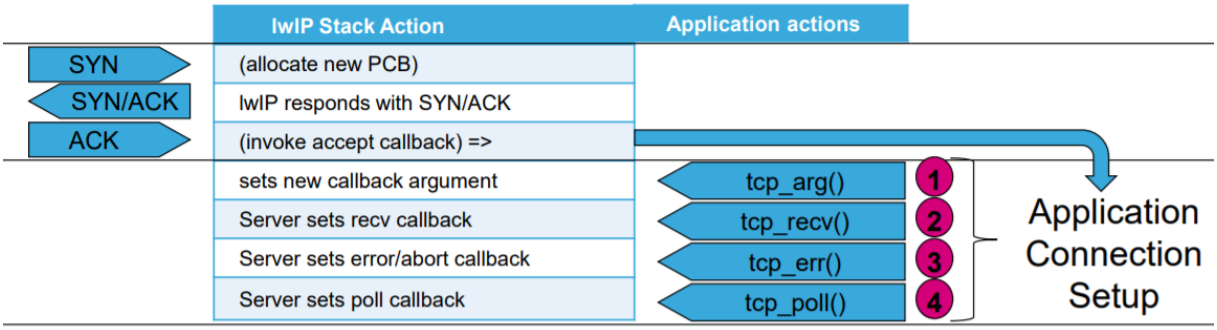
lwIP Stack Action	Application actions
Create TCP PCB	tcp_new() 1
Bind port number	tcp_bind() 2
Create listening endpoint (new PCB allocated)	tcp_listen() 3
Set accept callback	tcp_accept() 4

Application Listener Setup (server)

Server setup process (tcp_server_init@tcp_server.c) contains four basic functions. Accept callback function (tcp_server_accept@tcp_server.c) handles the incoming traffic. To keep track of the connection a tcp_server_structure cs variable is created. This control structure allows the program to track the packages and the connection status. In accept callback function connection status is converted to `ES_ACCEPTED` state to handle connection in receive callback function (tcp_server_recv@tcp_server.c).

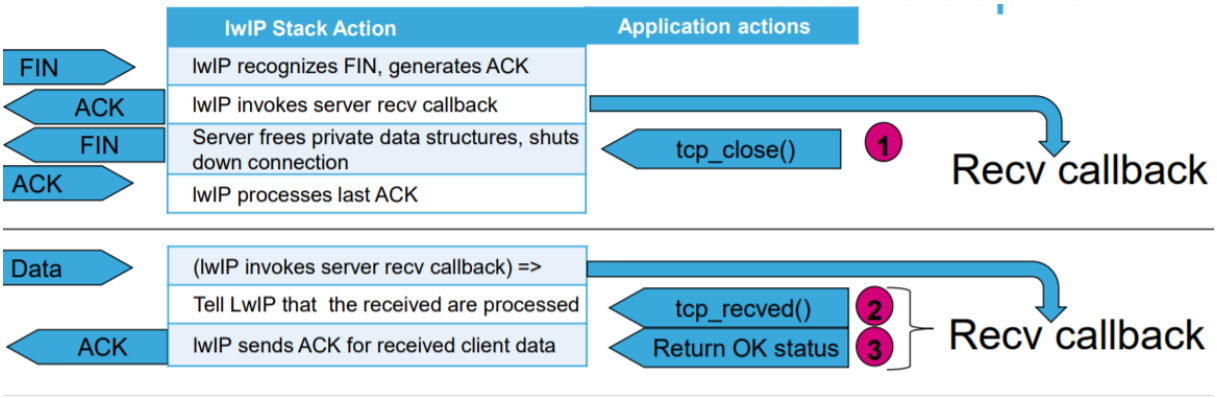
Accept Callback Function

The tcp_server_accept@tcp_server.c function is assigned as accept callback function in tcp_Server_init@tcp_server.c.



Accepting a connection requires four definition, which are one server structure assignment and three callback function assignments. Receive callback function handles the incoming packages considering the connection status tracked by cs variable.

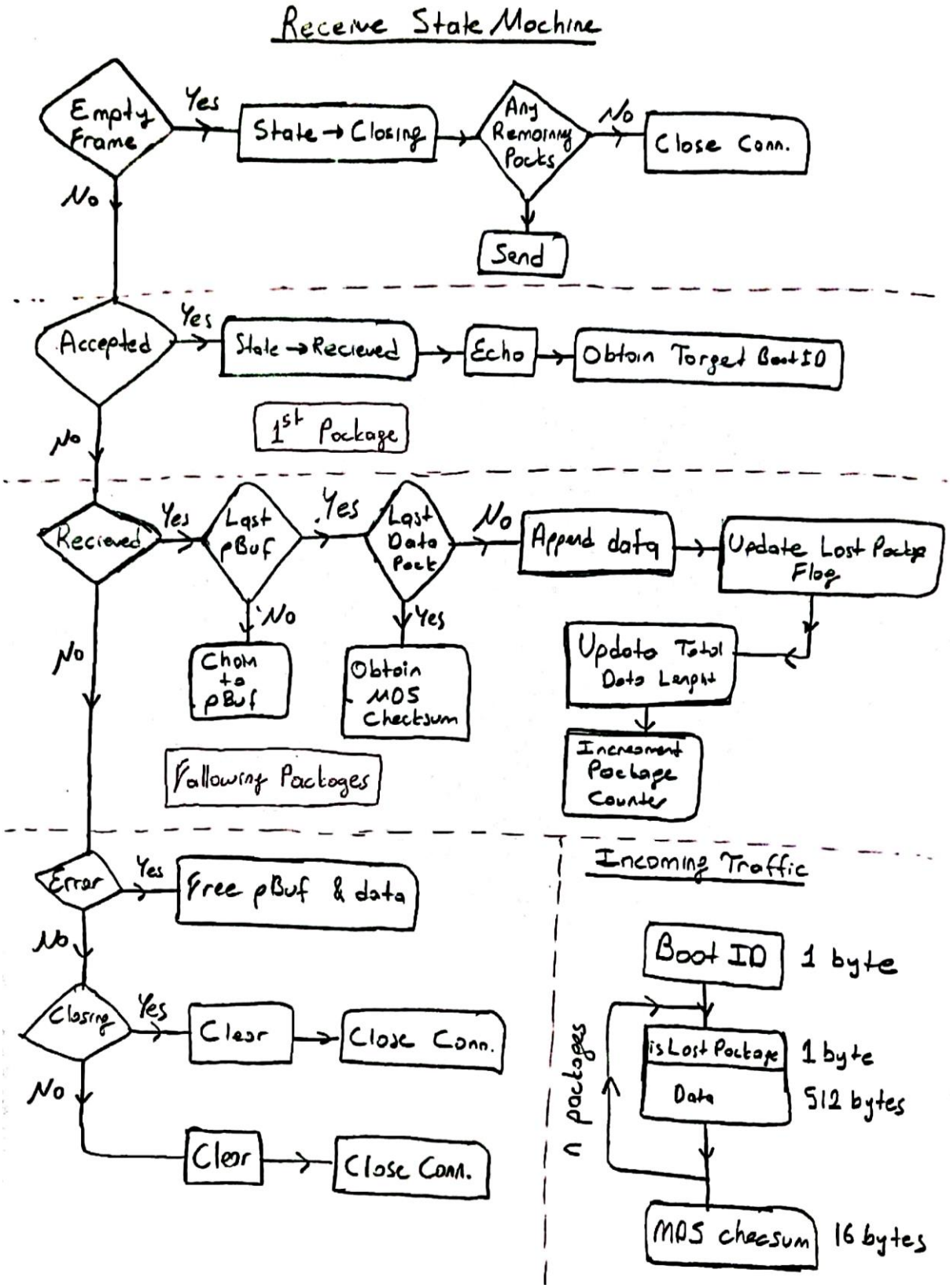
Receive Callback Function



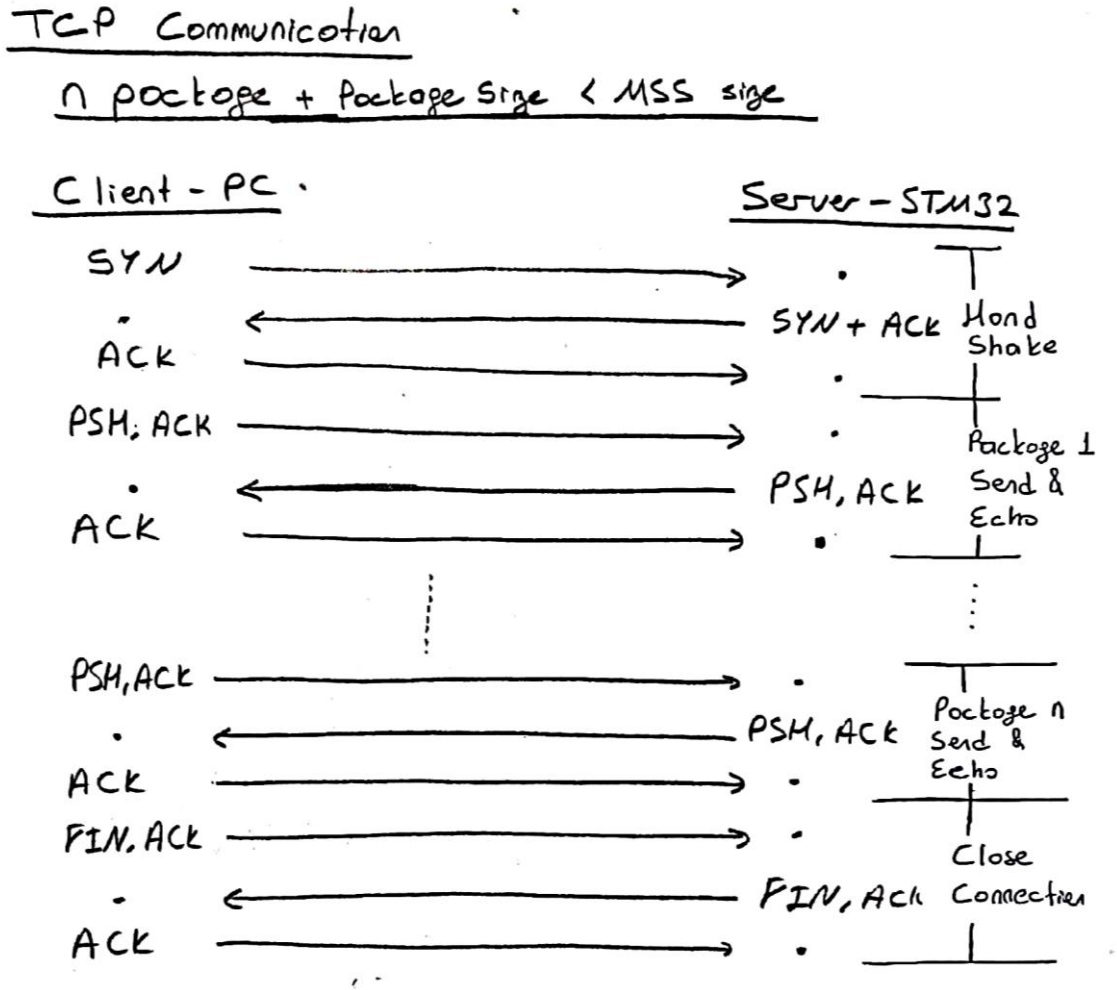
Data reception follows the procedures described in the figure above. Connection status is examined in the receive state machine to determine the action required. Receive Callback function should both the FIN flag and PSH flag separately. If the incoming package carrying a payload, then, receive state machine also should inform the LwIP Raw API that payload is received and handled by tcp_recved function.

The figures explaining the package and flag traffics between client and server can be observed clearly in the TCP Connection Tests section below.

TCP Package Receive State Machine



TCP Communication Test: Server Window Size Behavior



This experiment is conducted under these conditions:

- TCP Windows size: 2144 Bytes
- TCP Maximum Segment Size: 536 Bytes
- TCP Heap Memory Size: 1600 Bytes
- Sent TCP Package Size: 2 Bytes
- Number of sent packages: 300
- Total Bytes Sent: 600 Bytes

This experiment is to examine the handling process of TCP Window Size. In conclusion, windows size of the microcontroller is refreshed automatically, when it declines to 1610 Bytes.

817 28.869068	192.168.1.5	192.168.1.15	TCP	56 49237 → 80 [PSH, ACK] Seq=533 Ack=533 Win=64860 Len=2 [TCP segment of a reassembled PDU]
818 28.869332	192.168.1.15	192.168.1.5	TCP	60 80 → 49237 [PSH, ACK] Seq=533 Ack=535 Win=1610 Len=2 [TCP segment of a reassembled PDU]
819 28.914817	192.168.1.5	192.168.1.15	TCP	54 49237 → 80 [ACK] Seq=535 Ack=535 Win=64858 Len=0
820 28.976584	192.168.1.5	192.168.1.15	TCP	56 49237 → 80 [PSH, ACK] Seq=535 Ack=535 Win=64858 Len=2 [TCP segment of a reassembled PDU]
821 28.976840	192.168.1.15	192.168.1.5	TCP	60 80 → 49237 [PSH, ACK] Seq=535 Ack=537 Win=2144 Len=2 [TCP segment of a reassembled PDU]
822 29.023215	192.168.1.5	192.168.1.15	TCP	54 49237 → 80 [ACK] Seq=537 Ack=537 Win=65392 Len=0

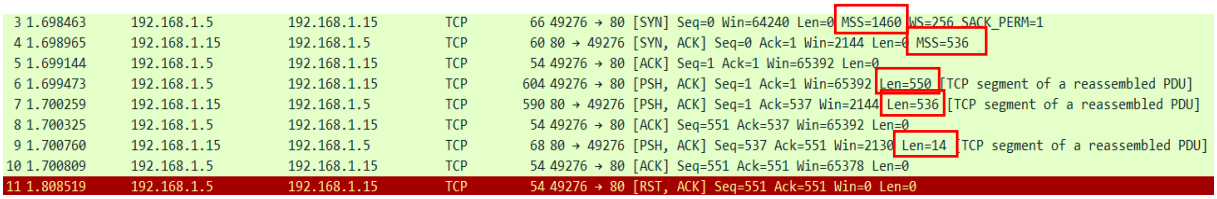
- Client – PC: 192.168.1.5
- Server – STM32: 192.168.1.15:80

TCP Communication Test: MSS Behavior

The Maximum Segment Size (TCP_MSS:536 Bytes) is broadcasted by both sides of the connection. The limiting size for MSS is 536 Bytes, hence the server – STM32 – is limited by this value on configuration stage on purpose. The main focus of following experiment is to investigate what happens when client wants to send package that is bigger than the maximum segment size of the server.

This experiment is conducted under these conditions:

- TCP Windows size: 2144 Bytes
- TCP Maximum Segment Size: 536 Bytes
- TCP Heap Memory Size: 1600 Bytes
- Sent TCP Package Size: 550 Bytes
- Number of sent packages: 1
- Total Bytes Sent: 550 Bytes



3	1.698463	192.168.1.5	192.168.1.15	TCP	66	49276 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
4	1.698965	192.168.1.15	192.168.1.5	TCP	60	80 → 49276 [SYN, ACK] Seq=0 Ack=1 Win=2144 Len=0 MSS=536
5	1.699144	192.168.1.5	192.168.1.15	TCP	54	49276 → 80 [ACK] Seq=1 Ack=1 Win=65392 Len=0
6	1.699473	192.168.1.5	192.168.1.15	TCP	604	49276 → 80 [PSH, ACK] Seq=1 Ack=1 Win=65392 Len=550 [TCP segment of a reassembled PDU]
7	1.700259	192.168.1.15	192.168.1.5	TCP	590	80 → 49276 [PSH, ACK] Seq=1 Ack=537 Win=2144 Len=536 [TCP segment of a reassembled PDU]
8	1.700325	192.168.1.5	192.168.1.15	TCP	54	49276 → 80 [ACK] Seq=551 Ack=537 Win=65392 Len=0
9	1.700760	192.168.1.15	192.168.1.5	TCP	68	80 → 49276 [PSH, ACK] Seq=537 Ack=551 Win=2130 Len=14 [TCP segment of a reassembled PDU]
10	1.700809	192.168.1.5	192.168.1.15	TCP	54	49276 → 80 [ACK] Seq=551 Ack=551 Win=65378 Len=0
11	1.808519	192.168.1.5	192.168.1.15	TCP	54	49276 → 80 [RST, ACK] Seq=551 Ack=551 Win=0 Len=0

- Client – PC: 192.168.1.5
- Server – STM32: 192.168.1.15:80

At the three handshake stage, MSS limits are broadcasted by both the client (1460 Bytes) and the server (536 Bytes). In the sending process, client informs the server that it wants to send 550-Byte long data (Len = 550). In return, the server can only echo 536-byte long data since it has MSS = 536 Bytes. Then, the server appends the remaining 14 bytes. However, hence the client is expecting a full echo, the communication is evaluated as “corrupted”, and responded by reset flag from the client.

Although, the echo feature is used only for debug purposes in this project, the maximum package size is limited by 512 data bytes + 1 flag within In-Application Programming (IAP) structure.

TCP Communication Test: Programming Example

This experiment is conducted under these conditions:

- TCP Windows size: 2144 Bytes
- TCP Maximum Segment Size: 536 Bytes
- TCP Heap Memory Size: 1600 Bytes
- Total Data -Bytes Sent: 4376 Bytes
- Max Package Size: 512 Bytes (Data) + 1 Byte (Flag) = 513 Bytes
- Total Data Buffer Size: 32 kB

6	4.083586	192.168.1.5	192.168.1.10	TCP	66	50280 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
7	4.095908	192.168.1.10	192.168.1.5	TCP	60	80 → 50280 [SYN, ACK] Seq=0 Ack=1 Win=2144 Len=0 MSS=536
8	4.096090	192.168.1.5	192.168.1.10	TCP	54	50280 → 80 [ACK] Seq=1 Ack=1 Win=65392 Len=0
9	4.096246	192.168.1.5	192.168.1.10	TCP	55	50280 → 80 [PSH, ACK] Seq=1 Ack=1 Win=65392 Len=1
10	4.097318	192.168.1.10	192.168.1.5	TCP	60	80 → 50280 [PSH, ACK] Seq=1 Ack=2 Win=2143 Len=1
11	4.097358	192.168.1.5	192.168.1.10	TCP	567	50280 → 80 [PSH, ACK] Seq=2 Ack=2 Win=65391 Len=513
12	4.100134	192.168.1.10	192.168.1.5	TCP	567	80 → 50280 [PSH, ACK] Seq=2 Ack=515 Win=1630 Len=513
13	4.143818	192.168.1.5	192.168.1.10	TCP	54	50280 → 80 [ACK] Seq=515 Ack=515 Win=64878 Len=0
14	4.205714	192.168.1.5	192.168.1.10	TCP	567	50280 → 80 [PSH, ACK] Seq=515 Ack=515 Win=64878 Len=513
15	4.206591	192.168.1.10	192.168.1.5	TCP	567	80 → 50280 [PSH, ACK] Seq=515 Ack=1028 Win=2144 Len=513
16	4.252377	192.168.1.5	192.168.1.10	TCP	54	50280 → 80 [ACK] Seq=1028 Ack=1028 Win=65392 Len=0
17	4.314832	192.168.1.5	192.168.1.10	TCP	567	50280 → 80 [PSH, ACK] Seq=1028 Ack=1028 Win=65392 Len=513
18	4.315716	192.168.1.10	192.168.1.5	TCP	567	80 → 50280 [PSH, ACK] Seq=1028 Ack=1541 Win=1631 Len=513
19	4.361491	192.168.1.5	192.168.1.10	TCP	54	50280 → 80 [ACK] Seq=1541 Ack=1541 Win=64879 Len=0
20	4.423198	192.168.1.5	192.168.1.10	TCP	567	50280 → 80 [PSH, ACK] Seq=1541 Ack=1541 Win=64879 Len=513
21	4.424248	192.168.1.10	192.168.1.5	TCP	567	80 → 50280 [PSH, ACK] Seq=1541 Ack=2054 Win=2144 Len=513
22	4.469518	192.168.1.5	192.168.1.10	TCP	54	50280 → 80 [ACK] Seq=2054 Ack=2054 Win=65392 Len=0
23	4.530791	192.168.1.5	192.168.1.10	TCP	567	50280 → 80 [PSH, ACK] Seq=2054 Ack=2054 Win=65392 Len=513
24	4.531783	192.168.1.10	192.168.1.5	TCP	567	80 → 50280 [PSH, ACK] Seq=2054 Ack=2567 Win=1631 Len=513
25	4.577885	192.168.1.5	192.168.1.10	TCP	54	50280 → 80 [ACK] Seq=2567 Ack=2567 Win=64879 Len=0
26	4.639546	192.168.1.5	192.168.1.10	TCP	567	50280 → 80 [PSH, ACK] Seq=2567 Ack=2567 Win=64879 Len=513
27	4.640537	192.168.1.10	192.168.1.5	TCP	567	80 → 50280 [PSH, ACK] Seq=2567 Ack=3080 Win=2144 Len=513
28	4.685454	192.168.1.5	192.168.1.10	TCP	54	50280 → 80 [ACK] Seq=3080 Ack=3080 Win=65392 Len=0
29	4.746993	192.168.1.5	192.168.1.10	TCP	567	50280 → 80 [PSH, ACK] Seq=3080 Ack=3080 Win=65392 Len=513
30	4.747837	192.168.1.10	192.168.1.5	TCP	567	80 → 50280 [PSH, ACK] Seq=3080 Ack=3593 Win=1631 Len=513
31	4.793415	192.168.1.5	192.168.1.10	TCP	54	50280 → 80 [ACK] Seq=3593 Ack=3593 Win=64879 Len=0
32	4.855398	192.168.1.5	192.168.1.10	TCP	567	50280 → 80 [PSH, ACK] Seq=3593 Ack=3593 Win=64879 Len=513
33	4.856290	192.168.1.10	192.168.1.5	TCP	567	80 → 50280 [PSH, ACK] Seq=3593 Ack=4106 Win=2144 Len=513
34	4.901959	192.168.1.5	192.168.1.10	TCP	54	50280 → 80 [ACK] Seq=4106 Ack=4106 Win=65392 Len=0
35	4.964607	192.168.1.5	192.168.1.10	TCP	335	50280 → 80 [PSH, ACK] Seq=4106 Ack=4106 Win=65392 Len=281
36	4.965840	192.168.1.10	192.168.1.5	TCP	335	80 → 50280 [PSH, ACK] Seq=4106 Ack=4387 Win=1863 Len=281
37	4.965951	192.168.1.5	192.168.1.10	TCP	70	50280 → 80 [PSH, ACK] Seq=4387 Ack=4387 Win=65111 Len=16
38	4.967319	192.168.1.10	192.168.1.5	TCP	70	80 → 50280 [PSH, ACK] Seq=4387 Ack=4403 Win=1847 Len=16
40	5.010511	192.168.1.5	192.168.1.10	TCP	54	50280 → 80 [ACK] Seq=4403 Ack=4403 Win=65095 Len=0
41	5.071227	192.168.1.5	192.168.1.10	TCP	54	50280 → 80 [FIN, ACK] Seq=4403 Ack=4403 Win=65095 Len=0
42	5.071668	192.168.1.10	192.168.1.5	TCP	60	80 → 50280 [FIN, ACK] Seq=4403 Ack=4404 Win=1846 Len=0
43	5.071748	192.168.1.5	192.168.1.10	TCP	54	50280 → 80 [ACK] Seq=4404 Ack=4404 Win=65095 Len=0

BootID

Data

MD5

- Client – PC: 192.168.1.5
- Server – STM32: 192.168.1.10:80

In this example, a blinky LED application code is written to Flash memory by using python script. The important parameters like origin address in linker script and vector offset parameter is set before compilation.

The first byte sent is the target Boot ID. The following 9 large packages are carrying actual boot data, each starting with last package flag byte. The last 16-byte long package is the MD5 checksum of total boot data to allow further control implementations on controller side. Python script output can be used detailed debugging purposes.

```
1st Chunk (HEX): [0x0, 0x0, 0x5, 0x20, 0xc1, 0xe, 0x2, 0x8, 0x55, 0x10, 0x2, 0x8, 0x57, 0x10, 0x2]

Last Chunk (HEX): [0x0, 0x8, 0x2, 0x40, 0x0, 0x24, 0xf4, 0x0, 0x1, 0x0, 0x0, 0x0, 0x10, 0x0, 0x0, 0x0]

General Code Information:
Total # of Bytes:        4376
Buffer Size:             512
Total # of Chunks:       9
# of Last Chunk bytes:   280
```

Boot Upload Code – Python Script

The script takes HEX file and separates predefined sized chunks to send. The script takes four parameters, namely, Chunk Size, Boot id, server IP and port. The attention must be given to the case that the input HEX file should be created upon two necessary settings:

- Target Application should be set correct Flash Origin address corresponding Boot ID
- The vector table offset value also should be updated

For example, an application which is intended to be uploaded to the Boot 0 Partition.

- Boot 0 Address is set in main.h file of Ethernet Bootloader Project. (0x08020000 – Sector 5)

Then, Linker and System_stm32xxx.c file must be updated, respectively.

- `  FLASH    (rx)    : ORIGIN = 0x08020000,   LENGTH = 384K /* 128Kb+256Kb */`
- `#define VECT_TAB_OFFSET  0x20000 /* Flash Base + offset */`

These settings should be changed only once before the HEX file generation. After obtaining and storing the HEX file, these settings can be changed to default values. In the case of any change in application code, a new HEX file must be generated following the necessary settings.

Python script provides following data for debug purposes:

- The first 16 bytes of 1st chunk
- The last 16 bytes of last chunk
- Number of Total Data Bytes
- Number of Chunks
- The Size of Last Chunk
- Max Chunk Size
- MD5 Checksum in HEX format

Known issues and limitations

- After CubeMX Code Generation performed, MX_LWIP_INIT function call should be manually deleted, since it is not compatible with static IP assignment. Instead, custom Netif_Config function is declared in main.c file to handle both DHCP server and Static IP options.
- The controller used in this project – STM32F746 (Nucleo Board)- has no hardware support for MD5 hash generation, unlike the same-family STM32F756 controller. Hence in order to add checksum controller mechanism, MbedTLS middleware might be used, however, there are not convincing documentation for implementing MbedTLS on lwIP Raw API without RTOS. (See Suggestions)
- Any external target application should be compiled and linked after these two settings: flash origin in linker script and vector table offset value (VECT_TAB_OFFSET) in system c file.
- The maximum size allocated for boot data buffer is 32 Kb (tcp_server.h), which limits the programming of large-scaled applications, however, this buffer size can be extended under the condition of remaining RAM size. The incoming boot data length is checked, however, in the case of excess, the program will halt in a while loop, since error handling procedure is not specified. Or instead of writing the boot code as a whole by dataBuffer, it can be separated to different write routines to be able to handle much larger data sizes than 32Kb.
- The Maximum Segment Size (MSS) is set to 536 Bytes. Therefore, Maximum chunk size in python script should not exceed 535 bytes for optimal operation. In the case of need of larger chunks, MSS must be extended before Max Chunk size and PACKAGE_BUFFER_SIZE in tcp_server.h file. Max Chunk Size and PACKAGE_BUFFER_SIZE must be equal. (See Suggestions)
- Since the beginning of the flash memory is allocated to the partition table, default boot address must be set correctly in option bytes [6]. To configure option bytes STM32Cube Programmer Software might be used. In this project Bootloader is moved to the beginning of the Sector 1 (@0x08008000) since it is multiple of 16kb to set boot address and the sector 0 assigned to partition table for convenience and page-organized flash memory implementations.
- The default boot setting in partition table is updated each new boot uploaded. Switching between boots is not implemented.
- There is no control or backup process, in a case of power lose or reset during boot programming and partition table update. If controller fails (resets or shutdowns) due to any external reason during programming boot to flash, followings can happen:
    - Flash sectors erased but no boot data is written: Bootloader lights RED LED to indicate that there is no meaningful boot on this boot address. (_estack check)
    - A portion of the boot data is written: Unpredictable Probably Hard Fault
        - This can be solved by MD5 checksum control if partition table is updated.
    - Boot data is written but partition table or boot configuration is not updated: Not Default boot, MD5 hash is not updated.
- The first validation test for a boot is done by checking whether the first byte of the boot is (_estack) end of the RAM = Origin(RAM) + Size(RAM). However, for different families that will run this bootloader, this first validation parameter should be set according to host controller. (_estack) is defined in linker script, and the first bytes of boot code is this specified RAM address.

Suggestions

- Since the controller used in this project has sector-organized flash memory, partition table is placed in the first sector in order to prevent unintended erase of bootloader code and to ease the implementations on page-organized flash memories. Page-organized flash memories is more suitable for this project [7].
- MD5 hash generation utility on STM32F746 is depended on both MbedTLS and FreeRTOS. Since it is not supported on hardware, unlike STM32F756, MD5 checksum check is not implemented. However, an implementation prototype is casted. For given calculatedMD5, there is checkBootMD5 function in partition_table_handler.c file [8] [9].
- Echo feature can be disabled since it is only used for debug purposes. However, it is kept enabled to allow further control implementations in client side. It also can be converted any response in case of need.
- In partition_table_handler.h file, the locations of the boot configurations are offseted from the table information by 0x40. However, for more compact table, consecutive table boot configuration locations can be translated sizeof(PartitionTable) and sizeof(OTA) further instead of 0x40.
- In the case of change in ethernet connection in idle mode in bootloader stage, link_callback function (ethernetif_update_config) is called. If any specific operation related to link up / down conditions, refer to ethernetif_update_config@ethernetif.c function.
- The selectedOTA byte described in PartitionTable structure is configured as int8_t to allow the user to implement a control mechanism like booting the bootloader when it has the value -1.
- For more robust package receiving, instead of using fixed $package\_counter \times PACKAGE\_BUFFER\_SIZE$ value, dynamic package buffer length (pbuf->len) and a tracking pointer might be used. This might help for a case in which, MSS value could not extended anymore and the need for using PACKAGE_BUFFER_SIZE greater than MSS value. Hence chunks are divided larger than the controller can handle (MSS), packages are divided for maximum value of MSS by TCP, automatically. But the suggested usage is to have: $MSS = 536\,Bytes > PACKAGE\_BUFFER\_SIZE + 1 = 512 + 1.$

Dynamic Sized Packages Implementation

TCP server is expecting packages which have maximum PACKAGE_BUFFER_SIZE + 1 bytes. If for any reason, dynamic package size is preferred, refer to here. For example, to achieve robustness in the case of $MSS < PACKAGE\_BUFFER\_SIZE + 1 = MAX\_CHUNK\_SIZE + 1$, this might be a starting point, although it is not recommended.

TCP Server

In the tcp_server_recv @tcp_server.c, the location where the boot data is going to be written is tracked by simple calculation of $package\_counter * PACKAGE\_BUFFER\_SIZE$. Instead, we can track the target location by a pointer initialized by the dataBuffer pointer and incremented by $pbuf -> len - 1$.

Standard Implementation

TCP Server receive callback:

```
…
if(!isLastPackage) {
	// Not Last Package
	memcpy((dataBuffer+package_count*PACKAGE_BUFFER_SIZE), (p->payload+1), (p->len-1));

	…
}
```

New Implementation

The target location to write data package can be tracked by dataPtr:

```
uint8_t* dataPtr = dataBuffer;
…
if(!isLastPackage) {
	// Not Last Package
	memcpy((dataPtr), (p->payload+1), (p->len-1));
	dataPtr += (p->len-1);
	…
}
```

MD5 Checksum Control Implementation

Partition Table Handler

A function prototype is assigned for the checking operation in partition_table_handler.h file:

```
/* MD5 Checksum Control */
uint8_t checkBootMD5(OTA* boot, uint8_t* calculatedMD5);
```

In partition_table_handler.c file, this prototype and a variable are defined:

```
uint8_t calculatedMD5[16] = {0};


/*
 *     This function is prototype for MD5 checksum control of boot
 */


uint8_t checkBootMD5(OTA* boot, uint8_t* calculatedMD5)
{
	UNUSED(boot);
	UNUSED(calculatedMD5);
	/*
	 *     return !(memcmp ( boot->MD5, calculatedMD5, 16 ));
	 *
	 *     memcmp returns 0, when there is no difference
	 *
	 */

	return 1;
}
```

This function is set in a way that it returns true always since calculatedMD5 variable is not updated.

Main file

In main.c file calculatedMD5 is exported. The only necessary implementation is to calculate MD5 from the address "$boot -> Address$" with "$boot -> Size$" many bytes, and assign it to exported variable calculatedMD5.

```
extern uint8_t calculatedMD5[16];
…
…

if(checkBootMD5(&boot, calculatedMD5)) {

…	// Jump to boot Routine

}
```

Partition Table on Sector vs. Page Implementation

In this project, partition table is located at the first sector of the STM32F746ZG Flash Memory. Since this bootloader project more suitable for page-organized flash memories, partition table is left as a single entity in the first sector to ease further page implementations and prevent unintended erase of codes. On the other hand, if there is no other option but sector-organized flash memory and do not want to allocate entire sector to partition table, you will need to implement an erase / write process to keep data erased with partition table.

Partition Table Handler

In partition_table_handler.h file, there is updatePartitionTableInFlash function prototype to allow further implementations. An example of a simple method steps:

- Read the entire sector in which the partition table located and store it in RAM or another Flash Memory Sector
- Backup the partition table and boot configurations in variables to update necessary values in next steps
- Erase the target sector
- Clear beginning of the sector which the partition table is allocated
- Write the non-partition-table data back to erased sector
- Use the same flowchart used to update the partition table and boot configurations (main.c)
    - Retrieve the partition table and boot configurations from backup variables or flash
        - readPartitionTableFromFlash
        - readBootFromTable
    - Update the target boot configurations and write to the flash
        - updateBootConfiguration
        - writeBoot2Table
    - Write the unchanged boot(s) back to the flash
        - writeBoot2Table
    - Update partition table and write to the flash
        - updatePartitionTableConfigurations
        - writePartitionTable2Flash

Instead of erasePartitionTable function, this implemented updatePartitionTableInFlash function should be used afterwards.

Default Definitions in the Project

Main Header File

Partition Table and Boot Configurations

- `#define USER_FLASH_BOOT0_ADDRESS 0x08020000 // @ 128 KB -> Sector 4`
- `#define USER_FLASH_BOOT0_LAST_PAGE_ADDRESS 0x0807FFFF // @ End of Sector 5 (256 kB)`
- `#define USER_FLASH_BOOT1_ADDRESS 0x08080000 // @ 256 KB -> Sector 6`
- `#define USER_FLASH_BOOT1_LAST_PAGE_ADDRESS 0x080FFFFF // @ End of Sector 5 (256 kB)`
- `#define USER_FLASH_END_ADDRESS 0x080FFFFF // @ End of Sector 7 (256 kB)`

General

- `//#define USE_DHCP`
- `#define SERVER_PORT 80`

Static IP Settings

/*Static IP ADDRESS: IP_ADDR0.IP_ADDR1.IP_ADDR2.IP_ADDR3 */

- `#define IP_ADDR0 (uint8_t) 192`
- `#define IP_ADDR1 (uint8_t) 168`
- `#define IP_ADDR2 (uint8_t) 1`
- `#define IP_ADDR3 (uint8_t) 10`

/*NETMASK*/

- `#define NETMASK_ADDR0 (uint8_t) 255`
- `#define NETMASK_ADDR1 (uint8_t) 255`
- `#define NETMASK_ADDR2 (uint8_t) 255`
- `#define NETMASK_ADDR3 (uint8_t) 0`

/*Gateway Address*/

- `#define GW_ADDR0 (uint8_t) 192`
- `#define GW_ADDR1 (uint8_t) 168`
- `#define GW_ADDR2 (uint8_t) 1`
- `#define GW_ADDR3 (uint8_t) 1`

TCP Server Header File

- `#define DATA_BUFFER_SIZE (32*1024) // 32 Kb`
- `#define PACKAGE_BUFFER_SIZE 512`

Partition Table Handler

- `#define PARTITION_TABLE_LOC 0x08000000`
- `#define TABLE_BOOT0_LOC (PARTITION_TABLE_LOC + 0x40)`
- `#define TABLE_BOOT1_LOC (TABLE_BOOT0_LOC + 0x40)`

PARTITION_TABLE_LOC := Beginning of the flash memory

Consecutive table boot locations can be translated sizeof(PartitionTable) and sizeof(OTA) further instead of 0x40.

References

[1] ST, "STM32 microcontroller system memory boot mode - Application note," [Online]. Available: https://www.st.com/resource/en/application_note/cd00167594-stm32-microcontroller-system-memory-boot-mode-stmicroelectronics.pdf.

[2] ST, "STM32F407/STM32F417 in-application programming (IAP) over Ethernet," [Online]. Available: https://www.st.com/resource/en/application_note/dm00036062-stm32f407stm32f417-inapplication-programming-iap-over-ethernet-stmicroelectronics.pdf.

[3] ST, "AN3968," [Online]. Available: https://www.st.com/resource/en/application_note/dm00036062-stm32f407stm32f417-inapplication-programming-iap-over-ethernet-stmicroelectronics.pdf.

[4] ST, "LwIP TCP/IP stack demonstration for STM32F4x7 microcontrollers," [Online]. Available: https://www.st.com/resource/en/application_note/dm00036052-lwip-tcpip-stack-demonstration-for-stm32f4x7-microcontrollers-stmicroelectronics.pdf.

[5] "STM32F75xxx and STM32F74xxx Referrence Manuel," [Online]. Available: https://www.st.com/resource/en/reference_manual/dm00124865-stm32f75xxx-and-stm32f74xxx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf.

[6] "STM32F74xxx / STM32F75xxx Reference Manuel - Boot address option bytes," [Online]. Available: https://www.st.com/resource/en/reference_manual/dm00124865-stm32f75xxx-and-stm32f74xxx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf#page=86.

[7] Dim, "Flash page size and Sectors," [Online]. Available: https://electronics.stackexchange.com/questions/278437/stm32f74x-flash-page-size-and-sectors/278439#278439.

[8] ST, "STM32 cryptographic library," [Online]. Available: https://www.st.com/resource/en/user_manual/dm00215061-stm32-crypto-library-stmicroelectronics.pdf#page=28.

[9] "Using LwIP stack with raw API on MBED TLS without RTOS," [Online]. Available: https://community.st.com/s/question/0D50X0000AlepRB/using-lwip-stack-with-raw-api-on-mbed-tls-without-rtos-using-cubemx.