Assignment 02: Design and Implement Boot Loader Program for DUOS

Course Name: Operating System
Instructor: Dr. Mosaddek Hossain Kamal
Submitted By: Ahmed Nesar Tahsin Choudhury (Roll: 02), Mehrajul Abadin Miraj (Roll: 20)
Submission Date: Nov 20, 2024

Detailed Documentation for Server and Client Files

Overview The project is a system for managing firmware updates and interactions between a server and a client microcontroller (STM32). The system uses serial communication to exchange data packets that include commands for operations such as debugging, initialization, data sending, and version checking. Here's how to use and understand these files:

1. How to Use the System

Server Setup

- Ensure Python3 is installed on your system.
- Connect the microcontroller to your computer using the appropriate serial connection, such as a USB to UART converter. We are using UART2, which is connected to the USB through the ST-LINK debugger.
- Before running the server, set the current OS version in the server from the `version.txt` file. If an updated version of the OS is available, place the binary file of the updated OS in the `os_files` directory. Additionally, update the `version_names.json` file so that it can locate the corresponding file name from the version major and version minor.
- Run the Python script (`server.py`) using the command `python3 server.py`. The script will automatically handle incoming packets from the microcontroller and respond based on the command type and data contained within these packets.

Client (Microcontroller) Setup

- The bootloader code should be compiled and flashed onto an STM32 microcontroller.
- The microcontroller should be configured to match the communication parameters specified in the Python script (e.g., baud rate, serial port).
- The microcontroller will initiate communications based on its firmware logic, responding to commands from the server or sending requests to the server.

2. Packet Protocol Details

Packet Structure

- **Command Field (1 byte)**: Determines the type of action to be performed.
- **Length Field (1 byte)**: Indicates the length of the data segment in the packet.
- **Data Field (up to 128 bytes)**: Contains the data to be processed or transmitted.
- **CRC Field (4 bytes)**: Contains a CRC checksum for error checking.

Packet Max Length

- The total length of a packet is fixed at 134 bytes.

3. Commands and Their Functions

- **Command 0 (DEBUG)**: Used for debugging purposes, allowing the server to receive and display debug messages from the client.
- **Command 1 (INITIALIZATION)**: Initializes communication and prepares the system for data transfer. Upon receiving this command, the server will send the number of chunks it will transmit for updating the OS. The server sends the number of chunks as a 32-bit integer, so the corresponding length will always be 32 bytes.
- **Command 2 (SEND)**: Manages the sending of data packets from the server to the client, typically used for firmware updates or command execution. This command sends the corresponding chunk as received from the client as an acknowledgment.
- **Command 3 (VERSION CHECK)**: Used to verify or check the firmware version between the client and server to ensure compatibility or identify the need for updates. This is the first command the client performs. In this command, the length field is 64 bits. The first 4 bytes in the data section represent the current version major, and the following 4 bytes represent the current version minor. Based on version matching, this command will either call 'test_init' or 'jump_to_os'.
- **Command 4 (OS CONSOLE)**: Transforms the server into an operating system console, allowing direct command input and responses from the connected microcontroller. After completing the OS update or version check, the bootloader will jump to the OS. In this mode, there is no protocol for console printing, so this command will disable the packet protocol on the server. Consequently, the `kprintf` function will serve as the console for the OS. NOTE: In the bootloader section, we use `UART_READ` with the packet protocol.
- **Command 5 (RETRANSMIT)**: Requests the retransmission of a packet, typically used when a CRC check fails.

4. Code Execution Flow

Server Side

- The server initializes by setting up serial communication and reading necessary version files.
- It enters a loop where it continuously listens for packets.
- Upon receiving a packet, the server converts it to a packet from (initially it is just a bytes array). Then, checks its integrity using a CRC check.
- Depending on the command in the packet, the server executes different handlers (e.g., send data, initialize communication).
- It can handle errors, retransmit requests, and debug information output.

Client Side

- The client initializes system peripherals and sets up the serial interface.
- Initially, it goes to 'check_for_updates'; from there, it has two options.
- It will request the server for the current updated OS version. Upon detecting a version mismatch, it will request an OS update. Otherwise, it will jump to the OS.
- For updating the OS, the client initially sends a `INITIALIZATION` request, the response of which provides it with the number of chunks to receive. After that, it sends ACK messages asking for the next chunk from the server. The server reads the request, and responds with the requested chunk.
- After the completion of file transfer, the version numbers are saved in the flash memory of the MCU.
- Later, the bootloader commands the server to act as a console for the OS. The server stops responding and changes it's state to `OS Console`. In this state, it can only print messages sent from the device.
- In every packet transfer, CRC validation is performed and the packets are retransmitted in case any error is detected.

5. Functions Input, Output, and Description

Server Function Documentation

1. `read_version()`

- **Input:** None
- **Output:** A tuple `(VERSION_MAJOR, VERSION_MINOR)` containing the major and minor version numbers, both integers.
- **Description:** Reads the version information from the `version.txt` file and returns the major and minor version numbers.

2. `get_version_name(major, minor)`

- **Input:**

  - `major` (int): The major version number.
  - `minor` (int): The minor version number.
- **Output:** A string representing the version name.
- **Description:** Returns the version name corresponding to the given `major` and `minor` version numbers from the `version_names.json` file.

3. `read_file_in_chunks(filename, chunk_size=PACKET_DATA_MAX_LENGTH)`

- **Input:**
  - `filename` (str): The name of the file to read.
  - `chunk_size` (int): The size of each chunk to read (default is `PACKET_DATA_MAX_LENGTH`).
- **Output:** None (Modifies a global `file_chunks` variable to hold the chunks).
- **Description:** Reads the specified file in chunks and stores them in the global `file_chunks` list.

4. `print_chunks()`

- **Input:** None
- **Output:** None
- **Description:** Prints the first two chunks from the `file_chunks` list in hexadecimal format. This was implemented for debugging.

5. `to_list_of_integers(packet: Packet)`

- **Input:**
  - `packet` (Packet): The packet object to convert.
- **Output:** A list of integers representing the packet's command, length, and data.
- **Description:** Converts the packet into a list of integers, including the command, length, and data fields.

6. `bytearray_to_packet(byte_array: bytearray)`

- **Input:**
  - `byte_array` (bytearray): A bytearray representing a packet.
- **Output:** A `Packet` object.
- **Description:** Converts a bytearray into a `Packet` object by extracting the command, length, data, and CRC fields.

7. `calculate_crc(packet: Packet)`

- **Input:**
  - `packet` (Packet): The packet to calculate the CRC for.
- **Output:** A placeholder value `0` (CRC calculation not yet implemented).

- **Description:** A placeholder function to calculate the CRC (currently returns 0).

8. `crc_validate(packet)`

- **Input:**
  - `packet` (Packet): The packet to validate.
- **Output:** Boolean value `True` (CRC validation always succeeds).
- **Description:** Validates the CRC of the given packet (always returns `True` in the current implementation).

9. `bytes_to_int(byte)`

- **Input:**
  - `byte` (bytes): A single byte to convert.
- **Output:** An integer representing the byte.
- **Description:** Converts a byte to an integer.

10. `int_to_byte(integer: int)`

- **Input:**
  - `integer` (int): An integer to convert.
- **Output:** A byte representing the integer.
- **Description:** Converts an integer to a byte.

11. `file_size(file_path)`

- **Input:**
  - `file_path` (str): The path to the file.
- **Output:** An integer representing the size of the file in bytes.
- **Description:** Returns the size of the specified file.

12. `handle_debug(packet: Packet)`

- **Input:**
  - `packet` (Packet): The packet to handle.
- **Output:** None
- **Description:** Handles the "debug" command by printing the data from the packet.

13. `handle_initialization(ser: serial.Serial, packet: Packet)`

- **Input:**
  - `ser` (serial.Serial): The serial connection.
  - `packet` (Packet): The packet containing initialization data.
- **Output:** None
- **Description:** Handles the "initialization" command by sending an initialization packet over the serial connection.

14. `handle_send(ser: serial.Serial, packet: Packet)`

- **Input:**
  - `ser` (serial.Serial): The serial connection.
  - `packet` (Packet): The packet to handle.
- **Output:** None
- **Description:** Handles the "send" command by sending a chunk of data over the serial connection.

15. `receive_packet_bytes(ser: serial.Serial)`

- **Input:**
  - `ser` (serial.Serial): The serial connection.
- **Output:** A `bytearray` containing the received packet bytes.
- **Description:** Receives a packet over the serial connection and returns it as a bytearray.

16. `handle_version_check(ser: serial.Serial, packet: Packet)`

- **Input:**
  - `ser` (serial.Serial): The serial connection.
  - `packet` (Packet): The packet to handle.
- **Output:** None
- **Description:** Handles the "version check" command by sending the version information over the serial connection.

17. `handle_crc_error(ser: serial.Serial)`

- **Input:**
  - `ser` (serial.Serial): The serial connection.
- **Output:** None
- **Description:** Sends a retransmission request packet in case of a CRC error.

18. `handle_os_print(ser, packet)`

- **Input:**
  - `ser` (serial.Serial): The serial connection.
  - `packet` (Packet): The packet to handle.
- **Output:** None
- **Description:** Handles the "OS console" command by printing the received data to the console.

19. `operate(ser: serial.Serial, packet: Packet)`

- **Input:**
  - `ser` (serial.Serial): The serial connection.
  - `packet` (Packet): The packet to handle.

- **Output:** None
- **Description:** Operates based on the packet's command by delegating to the appropriate handling function.

20. `test_write(ser: serial.Serial)`

- **Input:**
  - `ser` (serial.Serial): The serial connection.
- **Output:** None
- **Description:** Sends test data over the serial connection.

21. `crc32_stm32_32bit(data: list[int]) -> int`

- **Input:**
  - `data` (list[int]): A list of 32-bit integers to calculate the CRC for.
- **Output:** An integer representing the calculated CRC.
- **Description:** Calculates the CRC32 checksum using the STM32F446RE's polynomial (Ethernet CRC-32).

22. `validate_crc(packet: Packet)`

- **Input:**
  - `packet` (Packet): The packet to validate.
- **Output:** Boolean `True` if the packet's CRC is valid, otherwise `False`.
- **Description:** Validates the CRC of the given packet by comparing the calculated CRC with the packet's CRC.

23. `test_crc(packet: Packet)`

- **Input:**
  - `packet` (Packet): The packet to test the CRC for.
- **Output:** A list of integers representing the packet's data for CRC calculation.
- **Description:** Tests the CRC calculation by comparing the packet's CRC with the calculated CRC.

24. `os_print(ser: serial.Serial)`

- **Input:**
  - `ser` (serial.Serial): The serial connection.
- **Output:** None
- **Description:** Prints any available data from the serial connection to the console.

STM32 Function Documentation

`CRC_Init`

- **Input:** None.
- **Output:** None.
- **Description:** Initializes the CRC peripheral by enabling the clock and resetting the CRC control register.

`CRC_Calculate(uint32_t *data, uint32_t length)`

- **Input:**
  - `data`: Pointer to an array of 32-bit integers to calculate the CRC for.
  - `length`: Number of 32-bit integers in the `data` array.
- **Output:**
  - Returns a 32-bit CRC value.
- **Description:** Computes the CRC of a given data array using the CRC hardware.

`flash_read_byte(uint32_t* address)`

- **Input:**
  - `address`: Address from which to read a byte.
- **Output:**
  - Returns an 8-bit value read from the given address.
- **Description:** Reads a byte of data from the specified memory address.

`flash_read_register(uint32_t* address)`

- **Input:**
  - `address`: Address of the register to read.
- **Output:**
  - Returns a 32-bit value read from the given address.
- **Description:** Reads a 32-bit register value from the specified address.

`flash_unlock`

- **Input:** None.
- **Output:** None.
- **Description:** Unlocks the flash memory by writing a specific sequence to the FLASH key register.

`flash_lock`

- **Input:** None.
- **Output:** None.
- **Description:** Locks the flash memory by setting the lock bit in the FLASH control register.

`flash_set_program_size(uint32_t psize)`

- **Input:**
  - `psize`: Desired program size (0 for byte, 1 for half-word, etc.).
- **Output:** None.
- **Description:** Configures the flash programming size in the control register.

`flash_wait_for_last_operation`

- **Input:** None.
- **Output:** None.
- **Description:** Waits until the last flash operation completes by polling the busy flag in the status register.

`flash_program_byte(uint32_t address, uint8_t data)`

- **Input:**
  - `address`: Memory address to program.
  - `data`: 8-bit value to write at the specified address.
- **Output:** None.
- **Description:** Programs a single byte into flash memory at the specified address.

`flash_program_4_bytes(uint32_t address, uint32_t data)`

- **Input:**
  - `address`: Memory address to program.
  - `data`: 32-bit value to write at the specified address.
- **Output:** None.
- **Description:** Programs four bytes (a word) into flash memory at the specified address.

`flash_program(uint32_t address, const uint8_t *data, uint32_t len)`

- **Input:**
  - `address`: Memory address to start programming.
  - `data`: Pointer to a buffer containing the data to write.
  - `len`: Number of bytes to write.
- **Output:** None.
- **Description:** Writes a sequence of bytes to flash memory.

`flash_erase_sector(uint8_t sector, uint32_t program_size)`

- **Input:**
  - `sector`: Sector number to erase.
  - `program_size`: Flash program size configuration.
- **Output:** None.

- **Description:** Erases a specific sector of flash memory.

`test_flash`

- **Input:** None.
- **Output:** None.
- **Description:** Tests flash memory operations including unlocking, erasing, programming, and reading values.

`calculate_packet_CRC(struct Packet* packet)`

- **Input:**
  - `packet`: Pointer to a `Packet` structure containing data for CRC calculation.
- **Output:**
  - Returns a 32-bit CRC value.
- **Description:** Calculates the CRC for a given packet based on its fields.

`populate_CRC(struct Packet* packet)`

- **Input:**
  - `packet`: Pointer to a `Packet` structure where CRC will be calculated and stored.
- **Output:**
  - Returns the CRC as an integer for confirmation.
- **Description:** Computes and populates the `crc` field of the given packet.

`send_packet(struct Packet* packet)`

- **Input:**
  - `packet`: Pointer to a `Packet` structure to be sent.
- **Output:** None.
- **Description:** Sends a packet over the UART interface, including its command, length, data, and CRC.

`debug(const uint8_t* statement)`

- **Input:**
  - `statement`: Pointer to a null-terminated string to debug.
- **Output:** None.
- **Description:** Prepares a debug packet with the given statement and sends it.

`bits32_from_4_different_bytes(uint32_t a, uint32_t b, uint32_t c, uint32_t d)`

- **Input:**
  - `a, b, c, d`: Four individual bytes (as 32-bit integers).

- **Output:**
  - Returns a 32-bit value formed by concatenating the inputs.
- **Description:** Combines four bytes into a single 32-bit value.

`bits32_from_4_bytes(uint8_t *bytes)`

- **Input:**
  - `bytes`: Pointer to an array of four bytes.
- **Output:**
  - Returns a 32-bit value formed by concatenating the input bytes.
- **Description:** Combines an array of four bytes into a single 32-bit value.

`init`

- **Input:** None.
- **Output:** None.
- **Description:** Initializes and sends a default packet.

`packet_from_bytes(uint8_t* data_bytes, struct Packet* packet)`

- **Input:**
  - `data_bytes`: Array of bytes representing a serialized packet.
  - `packet`: Pointer to a `Packet` structure to populate.
- **Output:** None.
- **Description:** Deserializes a byte array into a `Packet` structure.

`receive_packet(struct Packet* packet)`

- **Input:**
  - `packet`: Pointer to a `Packet` structure to populate.
- **Output:**
  - Returns `1` on successful reception, `0` on error.
- **Description:** Receives and deserializes a packet from the UART interface.

`test_read`

- **Input:** None.
- **Output:**
  - Returns an 8-bit value read from the UART interface.
- **Description:** Waits for and returns a valid byte from the UART interface.

`test_ring_buffer`

- **Input:** None.
- **Output:** None.
- **Description:** Tests UART ring buffer functionality by reading data and writing it to flash.

`test_flash_write_4_bytes`

- **Input:** None.
- **Output:** None.
- **Description:** Tests writing and verifying a 32-bit value in flash memory.

`write_init`

- **Input:** None.
- **Output:** None.
- **Description:** Prepares flash memory for writing by erasing the required sector.

`write_end`

- **Input:** None.
- **Output:** None.
- **Description:** Locks the flash memory after writing.

`write(struct Packet* packet, uint32_t chunk_index)`

- **Input:**
  - `packet`: Pointer to a `Packet` structure containing data to write.
  - `chunk_index`: Index of the chunk to write within the flash memory.
- **Output:** None.
- **Description:** Writes a specific packet chunk to the flash memory at the appropriate address.

6. Overall Outcome

- The system enables robust communication between a server and a microcontroller for tasks such as firmware updates, debugging, and command execution.
- It uses a structured packet protocol to ensure reliable data transmission with CRC checks for integrity.
- The system's design allows for easy expansion or modification to include more commands or handle different types of data transmissions.

This detailed documentation provides a comprehensive understanding of the system's functionality and implementation, allowing for effective use, troubleshooting, and further development.

7. Cautions

- **UART Ring Buffer**: Due to probable issues with the UART ring buffer, it is necessary to flush the UART buffer after each reception to ensure accurate and reliable data handling.
- **CRC Implementation**:

  - There are existing sections of code involving CRC that have been commented out due to conflicts with our implemented macro definition named CRC. These sections should be reviewed.
  - The CRC_DR register expects 32 bits of data. So, we had to combine four uint8_t variables into one uint32_t variable by bitwise shifting before we modified the register. Writing uint8_t variables directly won't produce the correct CRC value.

- **Header File Access in Bootloader**: In the bootloader section, Visual Studio Code may encounter difficulties in locating the correct header files. To resolve this, one must manually navigate to and edit the header files as needed. In the future, we may look into the issue and fix it.
- **Data Writing Protocol**: We faced issues while trying to write individual bytes to the flash memory. To address this problem, data writing is performed in 32-bit segments, and the memory address is incremented by 4 bytes after each write to ensure data integrity and alignment. The issue is probably due to 32-bit alignment requirement in the MCU.
- **CRC Calculation**: Prior to performing CRC calculations, the CRC register must be reset. This step is crucial to maintain the accuracy of the CRC values computed during data transmission.
- **Serial Read**: Initially, we encountered issues while reading bytes one at a time using `ser.read(<number of bytes>)`. To address this, we adjusted our approach by waiting for `ser.in_waiting` to match the expected number of bytes before reading the exact amount. Further investigation is needed to identify the root cause of the issue.