Glavni Plan Projekta: Migracija "Hotel Controller" na ESP32 (WT32-ETH01)

1. Cilj Projekta

Cilj ovog projekta je potpuna migracija postojeće STM32F4 "Hotel Controller" aplikacije na ESP32 (WT32-ETH01) platformu. Projekat zahtijeva zadržavanje apsolutne kompatibilnosti sa postojećim RS485 protokolom [cite: 236-239, 292-323], HTTP CGI komandama i logikom blokirajućih upita (`HTTP2RS485`) , kako bi se osigurala integracija u postojeći sistem.

Zbog nedostatka naprednih *in-circuit* debagera (npr. Keil MDK), klasična monolitna `HC_Service` [cite: 236-239] arhitektura se napušta i zamjenjuje modernom, modularnom arhitekturom zasnovanom na nezavisnim "Menadžerima" (po uzoru na priloženi `update_manager.c` [cite: 326-361]).

2. Generalna Arhitektura (FreeRTOS)

Sistem će biti baziran na FreeRTOS-u. Umjesto jedne "superloop" petlje, logiku ćemo podijeliti na nezavisne zadatke (Tasks) i module.

2.1. Centralni Dispečer (Rs485Service)

Centralni dio sistema je `Rs485Service` zadatak. Njegov jedini posao **NIJE** da izvršava logiku, već da djeluje kao "dispečer" ili "dirigent" koji upravlja pristupom RS485 magistrali.

U svojoj glavnoj petlji, on će provjeravati prioritete:

1. **Najviši Prioritet:** `HttpQueryManager` (blokirajući upit od korisnika).
2. **Visok Prioritet:** `UpdateManager` (ažuriranje firmvera u toku).
3. **Normalan Prioritet:** `LogPullManager` (redovni "pull" mehanizam).
4. **Nizak Prioritet:** `TimeSyncManager` (periodični broadcast vremena).

Kada `Rs485Service` primi paket, on ga validira (SOH, CRC, EOT) i prosljeđuje odgovarajućem Menadžeru na obradu.

2.2. Logički Tok Podataka

Blokirajući HTTP Upit (npr. `cst=101`): `HTTP Klijent` -> `HttpServer` -> `HttpQueryManager::ExecuteBlockingQuery()` -> (Preuzima magistralu od `Rs485Service`) -> `RS485 Magistrala` -> (Odgovor) -> `Rs485Service::ProcessResponse()` -> `HttpQueryManager` (oslobađa blokadu) -> `HttpServer` -> `HTTP Klijent`

Redovni Polling: `Rs485Service` (daje dozvolu) -> `LogPullManager::Service()` -> `RS485 Magistrala` -> (Odgovor) -> `Rs485Service::ProcessResponse()` -> `LogPullManager::ProcessResponse()` -> `EepromStorage::WriteLog()`

3. Nomenklatura Projekta

Koristit ćemo striktnu nomenklaturu za svu kodu:

Kategorija	Stil (Format)	Primjer
Fajlovi (Moduli)	PascalCase	`Rs485Service.h` , `EepromStorage.cpp`
Tipovi (Klase, Strukture)	PascalCase	`class EepromStorage` , `struct LogEntry`
Funkcije / Metode	PascalCase	`void Initialize()` , `bool WriteLog(LogEntry* entry)`
Globalne Varijable	g_snake_case	`g_system_status` , `g_rs485_task_handle`
Članske Varijable (Klase)	m_snake_case	`uint16_t m_write_index;`
Lokalne Varijable	camelCase	`int localCounter` , `bool isFound`
Konstante / Makroi	ALL_CAPS_SNAKE_CASE	`RS485_DE_PIN` , `MAX_LOG_ENTRIES`

4. Hardverska Arhitektura (Mapa Pinova v9)

Ovo je zaključana mapa pinova za WT32-ETH01 modul:

Funkcija	Periferija	Pin na ploči	GPIO
Kritični Sat	EMAC CLK	IO0	GPIO0
RS485	`Serial2` RX	RXD	GPIO5
RS485	`Serial2` TX	TXD	GPIO17
RS485	Driver Enable	485_EN	GPIO33
I2C (EEPROM)	I2C SDA	IO14	GPIO14
I2C (EEPROM)	I2C SCL	IO15	GPIO15
DWIN Displej	`Serial0` TX	TX0	GPIO1
DWIN Displej	`Serial0` RX	RX0	GPIO3
Ext. Flash	SPI SCK (Sat)	IO12	GPIO12
Ext. Flash	SPI MOSI (Izlaz)	IO4	GPIO4
Ext. Flash	SPI MISO (Ulaz)	IO36	GPIO36
Ext. Flash	SPI CS	CFG	GPIO32

Status LED	On-board LED	IO2	GPIO2
WLAN Reset	Dugme	IO39	GPIO39
Rasvjeta	Relej	(Nema)	**VIRTUALNI PIN**
Debug Port	`Serial0`	TX0/RX0	GPIO1/3

5. Plan Razvoja (Faze)

1. **FAZA 1 (Razvoj Jezgra):**
   - Uređaj je spojen USB kablom. `Serial0` (`GPIO1/3`) se koristi za `Serial.println()` u VS Code. DWIN Displej je isključen.
   - Razvijamo i testiramo sve ključne module (`NetworkManager` , `EepromStorage` , `Rs485Service` , `LogPullManager` , `HttpQueryManager` , `UpdateManager`).
   - Sav debug ide na `Serial0` .
2. **FAZA 2 (Integracija Grafike):**
   - USB kabl je isključen. DWIN Displej se fizički spaja na `TX0` / `RX0` pinove.
   - Sav `Serial.println()` ispis sada automatski ide na DWIN.
   - Dodajemo kod za čitanje sa `Serial0` (`Serial.read()`) da bismo primali komande *sa* DWIN-a.

6. Detaljna Definicija Projektnih Zadataka (Modula)

Ovo je plan implementacije za svaki modul.

Zadatak 1: Modul `ProjectConfig` (`ProjectConfig.h`)

- **Cilj:** Centralno mjesto za svu konfiguraciju.
- **Ključne Funkcionalnosti:**
  - Definiše Mapu Pinova (v9) koristeći `#define` .
  - Definiše EEPROM memorijsku mapu (adrese za Konfiguraciju, Listu Adresa i Logove).
  - Definiše konstante RS485 protokola (SOH, EOT, `RESP_TOUT` ...).
  - Definiše našu dogovorenu Nomenklaturu u komentaru.

Zadatak 2: Modul `EepromStorage` (`EepromStorage.cpp / .h`)

- **Cilj:** Upravlja I2C EEPROM (24C1024) memorijom za svu trajnu pohranu podataka.
- **Ključne Funkcionalnosti:**
  - Inicijalizuje `Wire` (I2C) na pinovima `GPIO14` i `GPIO15` .
  - Implementira **Logger** baziran na `logger.c` (head/tail, `STATUS_BYTE_VALID`).
  - Čita i piše Konfiguracioni blok (IP, System ID...).
  - Čita i piše Listu Adresa (do 400 adresa).

- **Javni Interfejs (Prijedlog):**
  - `void Initialize();`
  - `bool ReadConfig(AppConfig* config);`
  - `bool WriteConfig(AppConfig* config);`
  - `bool ReadAddressList(uint16_t* listBuffer, uint16_t maxCount);`
  - `bool WriteLog(LogEntry* entry);`
  - `bool GetOldestLog(LogEntry* entry);`
  - `bool DeleteOldestLog();`
  - `void ClearAllLogs();`
- **Koraci Implementacije:**
  1. Implementirati `Initialize()` sa `Wire.begin()`.
  2. Portovati `logger.c` logiku (`LOGGER_Init`, `LOGGER_WriteEvent` ...) u C++ klasu, koristeći `Wire.write/read` umjesto STM32 HAL funkcija.
  3. Napisati helper funkcije za čitanje i pisanje bloka konfiguracije.
  4. Napisati helper funkcije za čitanje liste adresa (koja će biti učitana u RAM pri startu).

Zadatak 3: Modul `SpiFlashStorage` (`SpiFlashStorage.cpp / .h`)

- **Cilj:** Upravlja eksternim SPI Flash čipom za skladištenje Firmware (FW) fajlova.
- **Ključne Funkcionalnosti:**
  - Inicijalizuje SPI magistralu na pinovima iz v9 mape.
  - Pruža funkcije za brisanje (erase), pisanje (write) i čitanje (read) blokova podataka.
  - Ovaj modul će koristiti `UpdateManager`.
- **Javni Interfejs (Prijedlog):**
  - `void Initialize();`
  - `bool EraseChip();`
  - `bool BeginWrite(uint32_t address);`
  - `bool WriteChunk(uint8_t* data, uint16_t length);`
  - `bool EndWrite();`
  - `bool ReadChunk(uint32_t address, uint8_t* buffer, uint16_t length);`
  - `uint32_t GetJEDEC_ID();`

Zadatak 4: Modul `NetworkManager` (`NetworkManager.cpp / .h`)

- **Cilj:** Upravlja mrežnom konekcijom (Ethernet/Wi-Fi) i vremenom.
- **Ključne Funkcionalnosti:**
  - Inicijalizuje Ethernet (ETH) kao primarni interfejs.
  - Inicijalizuje Wi-Fi kao backup (koristeći `WiFiManager` ako ETH ne uspije).
  - Pokreće NTP klijenta za sinhronizaciju vremena (`configTzTime`).

- Pokreće Ping Watchdog za provjeru konekcije.
- Upravlja `WLAN_RST_BTN` (`GPIO39`) za reset Wi-Fi postavki.

Zadatak 5: Modul `Rs485Service` (Dispečer) (`Rs485Service.cpp / .h`)

- **Cilj:** Upravlja fizičkim pristupom RS485 magistrali i djeluje kao dispečer za druge module.
- **Arhitektura:** FreeRTOS zadatak (`rs485_task`).
- **Ključne Funkcionalnosti:**
  - Inicijalizuje `Serial2` (pinovi 5, 17) i `DE` pin (GPIO33).
  - Implementira `Run()` petlju koja je **Prioritetni Dispečer**.
  - Petlja `Run()` provjerava:
    1. Da li `HttpQueryManager` traži magistralu? (preko FreeRTOS `event` ili `flag` -a).
    2. Ako da: Daje mu ekskluzivni pristup i čeka da završi.
    3. Ako ne: Da li `UpdateManager` traži magistralu?
    4. Ako da: Poziva `UpdateManager::Service()` i prosljeđuje mu kontrolu za jedan ciklus.
    5. Ako ne: Da li `LogPullManager` traži magistralu?
    6. Ako da: Poziva `LogPullManager::Service()`.
    7. Ako ne: Da li `TimeSyncManager` traži magistralu?
    8. Ako da: Poziva `TimeSyncManager::Service()`.
  - Prima podatke sa `Serial2.read()`.
  - Validira SOH/CRC/EOT (ovo je njegova ključna uloga).
  - Prosljeđuje validan paket (`ProcessResponse`) onom menadžeru koji je trenutno aktivan.
- **Koraci Implementacije:**
  1. Kreirati FreeRTOS zadatak (`xTaskCreate`).
  2. Implementirati `Initialize()` sa `Serial2.begin()`.
  3. Implementirati `Run()` petlju sa logikom dispečera (za sada samo za `LogPullManager`).
  4. Implementirati `SendPacket(uint8_t* data, uint16_t length)` (upravlja `DE` pinom).
  5. Implementirati `ReceivePacket()` (ne-blokirajuće čitanje sa `Serial2`).

Zadatak 6: Modul `LogPullManager` (`LogPullManager.cpp / .h`)

- **Cilj:** Replicira logiku standardnog polling-a (`UPD_RC_STAT`) i skupljanja logova (`UPD_LOG`) iz `HC_Service`.
- **Arhitektura:** Vođen od strane `Rs485Service` dispečera.
- **Ključne Funkcionalnosti:**
  - Drži `m_current_address_index` za listu adresa (učitana iz `EepromStorage`).
  - `Service()`:
    1. Uzima `HC_GetNextAddr()` (našu verziju).
    2. Traži od `Rs485Service` da pošalje `GET_SYS_STAT` paket na tu adresu.

3. Postavlja interno stanje na `WAITING_FOR_STATUS_ACK`.

- `ProcessResponse(uint8_t* packet)`:
  1. Prima odgovor na `GET_SYS_STAT`.
  2. Ako odgovor kaže "imam log" (`rx_buff[7] == '1'`), mijenja stanje u `WAITING_FOR_LOG`.
  3. Traži od `Rs485Service` da pošalje `GET_LOG_LIST` paket.
  4. Kada primi log, prosljeđuje ga `EepromStorage::WriteLog()`.

Zadatak 7: Modul `HttpQueryManager` (`HttpQueryManager.cpp / .h`)

- **Cilj:** Replicira logiku blokirajućih upita (`HTTP2RS485` [cite: 236-239] i `HC2RT_Link` [cite: 236-239, 292-323]).
- **Arhitektura:** Koristi FreeRTOS semafor/queue za blokiranje pozivajućeg `HttpServer` zadatka.
- **Javni Interfejs (Prijedlog):**
  - `bool ExecuteBlockingQuery(HttpCommand* cmd, uint8_t* responseBuffer);`
- **Koraci Implementacije:**
  1. Kreirati semafor (`xSemaphoreCreateBinary`).
  2. Implementirati `ExecuteBlockingQuery`:
     1. Formatirati RS485 paket na osnovu `cmd` (uključujući `owa=` logiku za bridging).
     2. Postaviti flag `Rs485Service` dispečeru ("Trebam magistralu ODMAH!").
     3. Čekati da `Rs485Service` da dozvolu.
     4. Poslati paket preko `Rs485Service::SendPacket()`.
     5. Čekati na semafor (`xSemaphoreTake`) sa timeoutom od 5-10 sekundi.
     6. Ako je odgovor stigao (vidi korak 3), kopirati ga u `responseBuffer` i vratiti `true`.
     7. Ako je timeout, vratiti `false`.
  3. Implementirati `ProcessResponse(uint8_t* packet)`:
     1. Kopirati primljeni paket u interni bafer.
     2. Osloboditi semafor (`xSemaphoreGive`) da odblokira `ExecuteBlockingQuery`.

Zadatak 8: Modul `UpdateManager` (`UpdateManager.cpp / .h`)

- **Cilj:** Upravlja procesom ažuriranja firmvera, bazirano na `update_manager.c` [cite: 326-361].
- **Arhitektura:** Vođen od strane `Rs485Service` dispečera.
- **Koraci Implementacije:**
  1. Adaptirati `update_manager.c` [cite: 326-361] u C++ klasu.
  2. Zamijeniti `f_read` pozive sa pozivima našem `SpiFlashStorage` modulu.
  3. Zamijeniti `TF_SendSimple` sa `Rs485Service::SendPacket()`.
  4. `HttpServer` će pozivati `UpdateManager::StartSession()`.
  5. `Rs485Service` će pozivati `UpdateManager::Service()` i `UpdateManager::ProcessResponse()`.

Zadatak 9: Modul `HttpServer` (`HttpServer.cpp / .h`)

- **Cilj:** Obrada svih HTTP GET/POST zahtjeva.
- **Ključne Funkcionalnosti:**
  1. Inicijalizovati `AsyncWebServer` da sluša na oba interfejsa (ETH i Wi-Fi).
  2. Implementirati **sve** CGI handlere iz `Procitaj !!!.txt` [cite: 292-323].
  3. Za **ne-blokirajuće** komande (npr. `fwu=hc`): Pozvati `UpdateManager::StartSession()` i odmah vratiti "OK".
  4. Za **blokirajuće** komande (npr. `cst=101`): Pozvati `HttpQueryManager::ExecuteBlockingQuery()` i sačekati odgovor prije slanja klijentu.
  5. Implementirati handler za `POST /upload-firmware` koji podatke prosljeđuje `SpiFlashStorage::WriteChunk()` .
  6. Implementirati makro-zamjenu (npr. `RCgra` u `26486`) [cite: 292-323].

Zadatak 10: Ostali Moduli (`TimeSync` , `VirtualGpio`)

- **TimeSyncManager:** Jednostavan modul koji periodično (preko `Rs485Service`) šalje `SET RTC DATE TIME` broadcast.