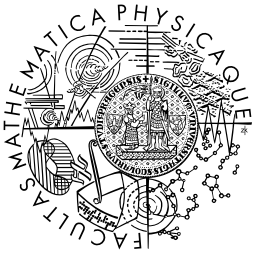


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Pavel Marek

Chytrý termostat pro platformu STM32

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce:	Doc. RNDr. Tomáš Bureš, Ph.D.
Studijní program:	Informatika
Studijní obor:	Programování a softwarové systémy

Praha 2018

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Rád bych poděkoval svému vedoucímu panu doc. Burešovi za jeho četné rady a připomínky.

Název práce: Chytrý termostat pro platformu STM32

Autor: Pavel Marek

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Doc. RNDr. Tomáš Bureš, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Cílem práce je vytvořit systém pro regulaci domácího vytápění. Součástí celého systému je embedded zařízení z platformy STM32, které je umístěno u uživatele doma a pravidelně měří a udržuje teplotu tak, jak ji uživatel přednastavil. Dále centrální webový server, se kterým zařízení komunikuje, a pomocí kterého uživatel může na dálku měnit nastavení a sledovat aktuální stav zařízení. Součástí práce je návrh a implementace komunikace mezi a zařízením a webovým serverem, v rámci které zařízení periodicky posílá na server naměřenou teplotu a synchronizuje s ním nastavení.

Klíčová slova: Internet věcí embedded programování

Title: Smart thermostat on STM32

Author: Pavel Marek

Department: Department of distributed and dependable systems

Supervisor: Doc. RNDr. Tomáš Bureš, Ph.D., Department of distributed and dependable systems

Abstract: Aim of this thesis is to develop a home thermoregulation system. There is an embedded device from the STM32 platform that measures actual temperature and regulates the heating based on user's preferences. This device communicates with a central web server through which user can monitor all his devices and change heating preferences on them. Implementation of the communication protocol between the embedded device and the web server periodically sends measured temperature from the embedded device and synchronizes heating preferences on both sides.

Keywords: Internet of things embedded programming

Obsah

Úvod

Termín *embedded* zařízení označuje zařízení, které má typicky relativně omezené výpočetní schopnosti, zejména ve smyslu výkonnosti procesoru a velikosti paměti. Funkcionalita, kterou mohou embedded zařízení mít, sahá od té nejjednodušší jako je třeba mikrokontrolér v pračce, až po komplexní, jako jsou třeba chytré hodinky. S tím přímo souvisí i architektura jejich softwaru. Na těch nejjednodušších, do nekonečna vykonávajících jedinou úlohu, běží v cyklu jediný software nazývaný v takovém případě firmware. Na komplexnějších, typicky hardwarově výkonnějších zařízeních už může běžet nějaký vyšší operační systém typu Linux nebo Windows, a tedy od klasických desktopových počítačů se příliš neliší.

Další důležitý termín pro tuto práci je *IoT* (Internet of Things), což je označení pro síť, která tyto embedded zařízení propojuje.

Tato práce se zabývá vývojem firmware pro embedded zařízení spadající do kategorie IoT s tím, že toto zařízení komunikuje s centrálním serverem.

0.1 Cíle práce

Cílem práce je vytvořit firmware pro embedded zařízení chovající se jako termostat tj. umožňující uživateli nastavit teplotu po libovolné časové intervaly přes den a zobrazující aktuálně naměřenou teplotu, a software pro webový server umožňující správu tohoto zařízení na dálku. Kromě firmware pro embedded zařízení a software pro webový server specifikuje tato práce také komunikaci mezi těmito entitami a jejich synchronizaci, při které musíme vzít v úvahu to, že uživatel může nastavovat různé hodnoty na embedded zařízení a na webovém serveru ve stejnou dobu.

0.2 Struktura práce

První kapitola nazvaná „Technologie“ představuje všechny technologie, které jsou v rámci celé práce použity. Patří sem mimo jiné volba konkrétního embedded zařízení.

Dále následují dvě hlavní kapitoly analýza a architektura, kde každá z nich má tři podsekce: „STM“, „Webový server“ a „Komunikace STM se serverem“. Podsekce „STM“ rozebírá embedded část práce, „Webový server“ se zabývá návrhem a implementací webového serveru a „Komunikace STM se serverem“ se zabývá propojením STM a webového serveru. Kapitola analýza nabízí několik možností řešení každé podsekce a vybírá z nich ty nejvhodnější. Kapitola architektura se zabývá konkrétní implementací pro každou podsekci.

Jako další je kapitola „Práce se systémem“, která popisuje použití v reálném prostředí, po ní kapitola „Vyhodnocení“, ve které je popsáno jakým způsobem se testuje funkcionalita celého systému.

1. Technologie

Tato kapitola se zabývá technologiemi, které máme v rámci naší práce k dispozici.

1.1 Koncové zařízení

Koncovým zařízením máme na mysli samotný termostat, který měří teplotu u uživatele doma. Zařízení má poměrně jednoduchou úlohu - má pouze měřit teplotu, umožnit uživateli nastavit teplotu po různé časové intervly a to všechno periodicky posílat na server. K vykonání takto jednoduché úlohy nepotřebujeme příliš paměti, ani výkonný procesor. Bohatě nám stačí zhruba 256 KB paměti pro kód a 64 KB RAM paměti. Po zařízení také požadujeme aby nebylo příliš drahé a tím pádem neobsahovalo zbytečně výkonný hardware, který nevyužijeme. To rovnou vyřazuje z výběru zařízení typu Raspberry Pi. Tím, že ušetříme na hardware si zase ztížíme práci na software. Kdybychom totiž použili zařízení typu Raspberry Pi, mohli bychom do něj, díky relativně výkonnému hardwaru, nahrát operační systém Linux a tím pádem bychom měli k dispozici nejen abstrakci nad hardwarem, ale také v podstatě libovolnou technologii, kterou bychom mohli použít při programování našeho softwaru. Jediné co by mohlo být problematické s použitím Linuxu je komunikace s externími komponentami tj. teplotním sensorem a relé.

1.1.1 Vstupně-výstupní požadavky

Vzhledem k tomu, že chceme aby zařízení zobrazovalo aktuální údaje a navíc aby uživatel mohl různé údaje na zařízení nastavovat, potřebujeme, aby zařízení mělo alespoň malý displej. Pro vstup od uživatele bychom mohli použít buď několik tlačítek nebo joystick což se dá považovat za téměř ekvivalentní řešení. Dále by se dal použít dotykový displej, ten je ale pro naše účely zbytečně nákladný.

V našem případě použijeme zařízení STM3210C-Eval board [1] se zabudovaným 3,2 palcovým displejem, joystickem, GPIO konektory a ethernetovým konektorem. Povaha našeho firmware umožňuje použít jiné zařízení z rodiny STM32 pouze s drobnými modifikacemi.

STM3210C-Eval board nepatří mezi nejlevnější zařízení, na druhou stranu ale obsahuje obrovské množství periferií které nevyužijeme - například CAN, Motor control, SD kartu, apod.

Poznámka: ve zbytku textu budeme koncové zařízení označovat pouze jako *STM*.

1.1.2 Externí komponenty

Teplotní senzor Dále potřebujeme dostatečně přesný a zároveň levný teplotní senzor. K tomu účelu nám vystačí DS18B20 [2], který můžeme k téměř jakémukoli zařízení připojit pomocí tří GPIO konektorů - napájení, uzemnění a data. Jedna z výhod teplotního senzoru DS18B20 je ta, že sám provádí převod z analo-

gové naměřené teploty na digitální data. Při maximální přesnosti měření na čtyři desetinná místa stupňů celsia trvá tento převod zhruba 750 ms.

Relé Jako poslední potřebujeme relé, které nám umožní spouštět připojený kotel.

1.2 Webový server

Na webový server máme následující požadavky:

- Podpora více STM zařízení pro jednoho uživatele.
- Zobrazování naměřené teploty a nastavených intervalů každého zařízení.
- Možnost přenastavit intervaly.

Požadavek na podporu zobrazování hodnot z více zařízení najednou téměř znemožňuje použití koncového zařízení jako webového serveru. A to jednak malou výkonností koncového zařízení, ale také přílišnou složitostí řešení - museli bychom se totiž rozhodovat, které koncové zařízení jednoho uživatele použijeme jako webový server, a nebo bychom museli uživateli dodat úplně jiné zařízení, na kterém webový server poběží.

V našem případě bude lepší použít právě jeden centrální webový server, který poběží na hardwaru typického serveru. Mimo jiné můžeme na tomto centrálním webovém serveru s výhodou využít architektury LAMP (Linux, Apache, MySQL, and PHP), která nám výrazně usnadní vývoj webu.

2. Analýza

V této kapitole rozebereme možnosti návrhu celého systému.

2.1 STM

2.1.1 Firmware

Tato kapitola se zabývá tím z jakých komponent můžeme poskládat výsledný firmware na STM. Součástí toho je krátký přehled knihoven, které můžeme použít k vývoji, a jejich porovnání.

STM32CubeF1

Výrobce STM poskytuje pro rodinu procesorů F1 [1] balíček STM32CubeF1 [3] obsahující různé knihovny pro abstrakci hardwarem na deskách obsahujících procesory F1. Zde je důležité zmínit výhodu použití nezměněné desky STM3210C-Eval board od STM oproti poskládání vlastní desky z různých hardwarových komponent. Kromě knihovny HAL (Hardware abstraction layer), která poskytuje abstrakci nad všemi periferiemi na desce, obsahuje totiž STM32CubeF1 i knihovny BSP (Board support package), které poskytují abstrakce pro rozšíření specifická pro jednotlivé desky. V našem případě je to například displej a joystick.

RTOS

Existuje více variant RTOS (real-time operating system). Pro naše účely bychom mohli použít například FreeRTOS [4]. FreeRTOS funguje v podstatě pouze jako plánovač úloh - s jeho použitím bychom v naší aplikaci definovali entry point úlohy, její prioritu apod., a pak bychom ji spustili. Největší výhoda FreeRTOSu je při použití TCP/IP komunikace, resp. LwIP knihovny. Umožňuje nám totiž programovat v tzv. LwIP sekvenčním API, ve kterém vytvoření nového TCP spojení a následné poslání dat můžeme naprogramovat v jednom bloku kódu. LwIP při použití s RTOS totiž pro příjem a odesílání paketů vytvoří nový task. Na druhou stranu při použití LwIP bez RTOS musíme programovat v tzv. callback API. V tomto případě reaguje LwIP na události jako je například přijetí paketu, navázání TCP spojení, atd., tak, že zavolá uživatelem specifikovanou callback funkci. Což znamená, že na zdánlivě velice jednoduchou úlohu typu: „vytvoř TCP spojení a pošli 20 bajtů dat“ potřebujeme alespoň tři odlišné funkce.

Nevýhoda RTOS nastává v momentě, kdy chceme, aby víc různých tasků přistupovalo ke stejným datům a musíme tasky synchronizovat. To může být zdrojem špatně replikovatelných a špatně laditelných chyb.

Použití RTOS pro naši aplikaci je zbytečné. Přestože bychom mohli několik „tasků“, které v rámci naší aplikace vykonáváme, pohodlně vytvořit a spustit s použitím RTOS, bohatě nám postačí tyto „tasky“ spouštět jako interrupt handlery některých hardwarových časovačů.

[1] Rodinou procesorů F1 máme na mysli procesory typu STM32F1xx

GUI

Pro vývoj uživatelského rozhraní můžeme použít buď knihovnu typu STemWin [5], nebo můžeme vytvořit vlastní jednoduchou knihovnu. Vzhledem k možných komplikacím při integraci tak veliké knihovny jako je STemWin a tomu, že předpokládáme velice jednoduché GUI na STM, je lepší implementovat vlastní knihovnu.

TCP/IP

Požadavky na TCP/IP:

- Pro jednoduchost předpokládejme, že server má statickou, veřejnou IPv4 adresu a nám tedy postačí podpora pro IPv4.
- TCP. To potřebujeme kvůli protokolu HTTP.
- Bylo by vhodné aby knihovna uměla i DNS, protože zakódovat IP adresu serveru do firmwaru STM by bylo jednak velice nepraktické a jednak by to neumožňovalo přesunout server na jinou IP adresu.
- DHCP - určitě totiž nechceme kódovat IP adresu STM do jeho firmwaru, mohlo by totiž dojít ke kolizi s jiným STM nebo s úplně jiným zařízením například tiskárnou nebo osobním počítačem v rámci sítě, kde se STM vyskytuje.

Na výběr máme z většího množství knihoven určených primárně pro embedded zařízení. Vzhledem k tomu, že součástí STM32CubeF1 je integrační vrstva pro knihovnu LwIP [6], která podporuje všechny výše zmíněné požadavky a ještě spoustu dalších, nemá smysl se zabývat porovnáváním LwIP s jinými TCP/IP knihovnami. Navíc ani nepožadujeme aby TCP/IP knihovna byla příliš efektivní. V našem případě je tedy nejlepší použít knihovnu LwIP, s tím, že její integrace nám zabere minimálně času.

2.1.2 Popis GUI

V této kapitole specifikujeme výstup STM tj. popíšeme co všechno a za jakých okolností bude STM zobrazovat na displej.

Nejprve specifikujeme v jakých stavech se STM může nacházet. Toto jsou pouze stavy, které nás zajímají v kontextu GUI, rozhodně se nejedná o souhrn všech možný stavů.

Stavy STM

- Ethernetová periferie je korektně inicializována a kabel je připojen. Zkráceně budeme značit `ETH-up`.
- Ethernetový kabel je buď odpojen nebo ethernetová periferie se z nějakého důvodu neinicializovala. Zkráceně budeme značit jako `ETH-down`.
- STM je připojeno k serveru. Zkráceně značíme `CONNECTED`.

- STM je odpojeno od serveru. Zkráceně značíme `DISCONNECTED`.
- STM se připojuje k serveru. Zkráceně značíme `CONNECTING`.
- V EEPROM nejsou uložena žádná konfigurační data intervalů. Tento stav nastává pouze v případě, kdy uživatel zapne STM poprvé.
- V EEPROM jsou uložena konfigurační data intervalů.
- Uživatel už zadal klíč do STM. Tento klíč se ukládá do EEPROM, aby ho uživatel nemusel zadávat opakovaně.
- Uživatel ještě klíč nezadal. To znamená, že se ještě nepokoušel připojit k serveru.

Invarianty

- Odpojení resp. zapojení ethernetového kabelu když je STM zapnuté by nemělo způsobit žádnou chybu. To znamená, že stavy `ETH-up` a `ETH-down` se mohou libovolně střídat.
- Pokud se STM dostane do stavu `ETH-up`, dá uživateli možnost připojit se k serveru.
- Pokud je STM ve stavu `CONNECTED`, nemůže se od serveru odpojit. Jediná možnost jak STM odpojit od serveru je buď ho resetovat nebo odpojit ethernetový kabel.
- Pokud STM vůbec není připojeno k internetu a EEPROM je prázdná, umožní STM uživateli nastavit intervaly.

Popis jednotlivých obrazovek

V rámci celého GUI nám stačí tyto grafické prvky:

- Tlačítka, pomocí kterých uživatel buď potvrdí zadaná data, nebo je zahodí. Kromě toho umožňují ještě tlačítka přepínání mezi jednotlivými obrazovkami.
- Okna s nastavitelnou hodnotou kde hodnota může být například čas v minutách, nebo teplota. Uživatel může hodnotu nastavit pomocí joysticku.
- Okna, která slouží pouze pro zobrazování nějaké hodnoty - například aktuálního času, nebo naměřené teploty.
- Různé nápisy resp. nadpisy pro lepší přehlednost.

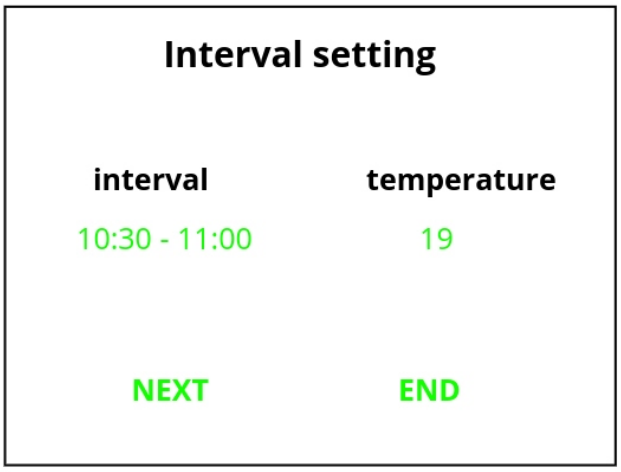
Nejprve uvedeme nákresy jednotlivých obrazovek a popíšeme co znamenají. Potom uvedeme tzv. frame diagram tj. stavový diagram obrazovek, kde popíšeme jak se mezi sebou jednotlivé obrazovky přepínají. Zeleně zabarvený text v obrázcích představuje buď nakliknutelné tlačítko nebo měnitelný prvek - například čas. Jak jsme již zmiňovali, STM se ovládá pomocí joysticku pod displejem. Pohybem joysticku do stran přepínáme mezi jednotlivými nakliknutelnými prvky. Pohybem

joysticku nahoru a dolu měníme hodnoty právě nakliknutých prvků, pokud to jde. Nakliknutý prvek na displeji STM poznáme podle červeného zabarvení.

Jednotlivé obrazovky jsou pojmenovány podle názvů tříd (v C++), které je reprezentují a jsou zmíněny v kapitole architektura.

Obrázek 2.1: ClkFrame - nastavení času

ClkFrame Obrazovka 2.1 je zobrazena po startu STM v momentě, kdy ethernetová periferie není inicializována a STM se tedy nemůže připojit k serveru aby mohlo rovnou synchronizovat čas se serverem. Uživatel musí nastavit nějaký čas (není nutné aby odpovídal reálnému času), aby STM vědělo, který interval je aktuální a tedy jakou teplotu má udržovat.



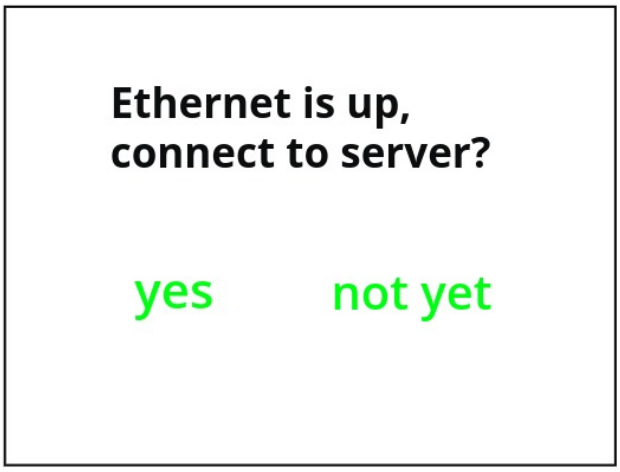
Obrázek 2.2: SetIntervalFrame - nastavení intervalů

SetIntervalFrame Obrazovka 2.2 umožňuje uživateli nastavit časové intervaly s teplotou. Obrazovka zobrazuje nastavení pro právě jeden interval, pokud uživatel stiskne tlačítko „Next“, zobrazí se nastavení dalšího intervalu, pokud stiskne tlačítko „End“, nastavování intervalů se ukončí a všechny nastavené intervaly jsou uloženy do EEPROM.

Zobrazena je v těchto případech:

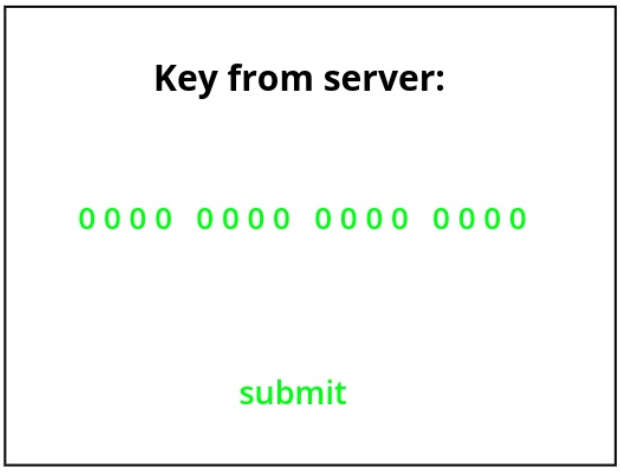
- Když EEPROM neobsahuje žádné nastavení intervalů. To nastává v případě kdy uživatel ještě žádné intervaly nenastavoval - tedy po prvním spuštění STM.
- Když uživatel zmáčkne v obrazovce `MainFrame` tlačítko „reset intervals“.

OverviewIntervalFrame `OverviewIntervalFrame` neboli „přehled všech nastavených intervalů“ vypadá stejně jako `SetIntervalFrame` až na to, že intervaly nejdou nastavovat, pouze prohlížet. Tlačítkem „Next“ se uživatel prokliká postupně přes všechny intervaly a tlačítkem „End“ zobrazování ukončí.



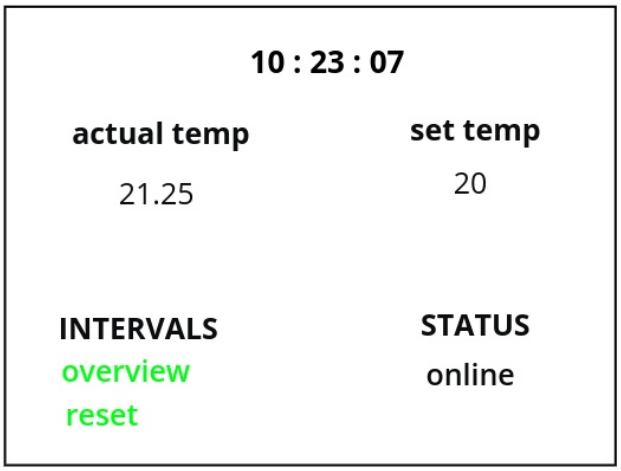
Obrázek 2.3: ConnectFrame - připojení k serveru

ConnectFrame Obrazovka 2.3 je zobrazena ihned po spuštění STM v momentě, kdy ethernetová periferie je inicializována a klíč ještě není uložen v EEPROM. V takovém případě se STM může ihned připojit k serveru a nemusíme uživatele zdržovat s nastavováním času.



Obrázek 2.4: KeyFrame - zadání klíče vygenerovaného serverem

KeyFrame Do obrazovky 2.4 uživatel zadá osmi bajtový DES klíč [7] vygenerovaný na serveru. Po stisknutí tlačítka „Submit“ se STM pokusí přihlásit na server.

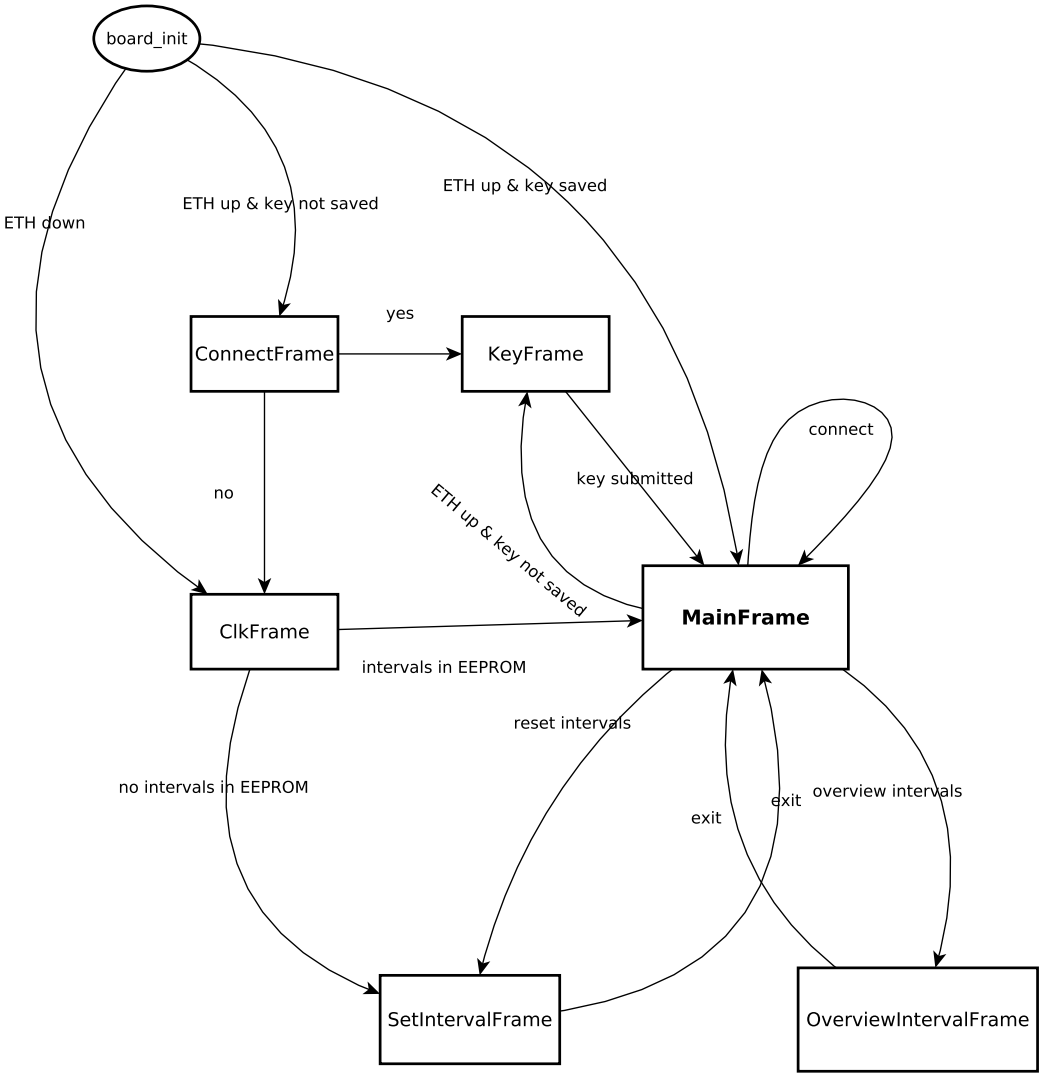


Obrázek 2.5: MainFrame - hlavní obrazovka

MainFrame MainFrame (2.5) reprezentuje hlavní obrazovku. Jsou zde zobrazeny zásadní hodnoty - aktuálně naměřená teplota (actual temp), přednastavená teplota (set temp) a stav připojení k serveru. Pokud je STM ve stavu `DISCONNECTED` a zároveň `ETH-up`, zobrazí v `MainFrame` tlačítko „connect“, pomocí kterého se uživatel dostane na obrazovku `KeyFrame`, do které zadá klíč vygenerovaný serverem a připojí se tak k serveru. V případě kdy je klíč uložen v EEPROM, tak se STM po stisku tlačítka „connect“ rovnou připojí k serveru a pouze zobrazí stav `CONNECTING`.

Uživatel se z `MainFrame` také může dostat do `SetIntervalFrame` pomocí „reset intervals“ tlačítka a do `OverviewIntervalFrame` pomocí „overview intervals“ tlačítka.

Vztahy mezi obrazovkami



Obrázek 2.6: Diagram přepínání obrazovek

Diagram 2.6 je v podstatě orientovaný graf s počátečním uzlem `board_init` a koncovým uzlem `MainFrame`. `ClkFrame` a `SetIntervalFrame` představují konfigurační obrazovky, které se postupně zobrazí v případě kdy je STM offline

nebo je spuštěno poprvé. Po zadání konfigurace, je-li to nutné, se uživatel nakonec dostane do `MainFrame`, ze kterého už může prohlížet jednotlivé intervaly v `OverviewIntervalFrame`, případně všechny intervaly přenastavit v `SetIntervalFrame`. Dále se z `MainFrame` může uživatel ještě dostat do `KeyFrame`, pokud se chce připojit k serveru a klíč ještě není uložen v EEPROM.

2.1.3 Použité periferie

V této části textu vypíšeme periferie na STM (tedy na desce STM3210C-Eval board), které jsou pro naši aplikaci nejdůležitější a stručně popíšeme jejich fungování. Nemá smysl zde opisovat technické detaily z referenčního manuálu [8].

EEPROM

EEPROM je 64 KB persistentní uložiště dat na I2C sběrnici. Uložit do něj můžeme v podstatě cokoli a kdykoli v libovolném formátu. V našem případě do této paměti budeme ukládat nastavení teplotních intervalů a privátní klíč určený k šifrování komunikace se serverem.

Časovače

Na STM je několik časovačů různých typů lišících se podle toho co všechno podporují. Velice zjednodušeně se na časovač dá dívat jako na komponentu, která má tři základní 16-bitové registry: prescaler, counter a auto-reload. Prescaler dělí frekvenci časovače a tím ovlivňuje rychlost zvyšování counteru. Counter registr se zvyšuje nebo snižuje o jedna každý tik časovače. Perioda časovače vyprší tehdy, když counter registr přeteče resp. podteče hodnotu z auto-reload registru - podle toho, jestli je časovač nastavený tak, aby counter zvyšoval nebo snižoval. Časovač se dá nastavit tak, aby při vypršení své periody vygeneroval *Update event* interrupt, který se nám bude hodit při spouštění různých tasků (tasky deklarujeme jako handlery tohoto přerušení).

RTC

RTC (real-time clock) je nezávislý časovač. Od ostatních časovačů se odlišuje tím, že se na něm dá snadno nastavit perioda s délkou jedné sekundy a že je schopen provozu i pokud je napájen pouze z baterie.

RTC má 32-bitový counter registr, do kterého můžeme zapsat libovolný čas a později z něj číst aktuální čas.

Hardwarová přerušení RTC dále lze nastavit tak, aby při každém zvýšení čítače generovalo hardwarové přerušení (second interrupt). V naší aplikaci je sekundové přerušení žádoucí.

Použití V naší aplikaci používáme pro synchronizaci času se serverem Unix timestamp [9]. Vzhledem k tomu, že do RTC counter registru můžeme kdykoli zapsat, můžeme do něj zapsat tento timestamp, který získáme ze serveru a RTC bude counter registr každou sekundu zvyšovat. Tímto jednoduchým způsobem umožníme synchronizaci času serveru a STM.

GPIO

Na STM je celkem 100 GPIO pinů. Každý z těchto pinů se dá nastavit jako vstupní nebo výstupní, navíc má k sobě interně připojený rezistor. Na teplotní senzor využijeme 3 GPIO piny - napájení, uzemnění a datový pin. Komunikace s teplotním senzorem je specifikována 1-Wire protokolem [2] a vyžaduje, aby na datovém pinu byl připojený pull-up rezistor. Tento datový pin stačí přepínat z režimu „input pull-up“ tj. vstupní režim s připojeným pull-up rezistorem, do režimu „output push-pull“ tj. výstupní režim bez připojeného rezistoru.

Ethernet

Vzhledem k veliké složitosti této periferie ji zde nebudeme vůbec popisovat a pouze řekneme, že ji budeme používat.

2.2 Webový server

2.2.1 Funkcionalita

Požadavky na webový server jsou:

- Podpora více STM pro jednoho uživatele.
- Připojení více STM najednou.
- Zobrazení stavu všech uživatelových STM.
- Možnost nastavit různou teplotu po různé časové intervaly.
- Autentikace STM.
- Autorizace uživatele.

Webový server slouží jako dálkové rozhraní mezi uživatelem a zařízením. Uživatel může odkudkoli, kde má připojení k internetu, zkontrolovat činnost zařízení a případně ho přenastavit.

Server podporuje pouze jeden druh zařízení (STM), nicméně je naprogramován tak, že pro případné rozšíření na více druhů zařízení nemusíme v jeho softwaru dělat veliké změny.

Udržování spojení s STM Server potřebuje s STM udržovat spojení, resp. potřebuje vědět, jestli je STM k serveru připojeno, nebo jestli je offline. Bylo by také vhodné, aby server tento stav dostatečně často aktualizoval a mohl ho tím pádem s malou odezvou zobrazit uživateli. Kapitola 2.3 detailně specifikuje komunikaci mezi serverem a STM. Pro teď nám vystačí informace, že veškerá komunikace bude probíhat pomocí HTTP zpráv. Ta je důležitá pro to, abychom se mohli rozhodnout jakým způsobem na serveru rozhodnout, zda je dané STM zařízení offline nebo je k němu připojené.

Možnosti jak udržovat spojení s STM jsou:

[2] 1-Wire protokol je popsán v [2]

1. Využít vlastnosti persistent connection u HTTP/1.1 [10]. S využitím této vlastnosti by server mohl STM deklarovat jako offline v momentě, kdy by uzavřel TCP spojení.

2. S ohledem na jednoduchost klienta (STM), udržet komunikaci co nejjednodušší tj. na HTTP/1.0 s co nejmenším počtem header options ze strany klienta. V takovém případě je na serveru nutnost zavést něco jako mechanismus časovače, který se obnoví po každé, když přijde libovolná zpráva od daného STM, a v momentě kdy vyprší, označí STM jako offline.

3. STM a server budou udržovat separátní TCP spojení na úplně jiném portu, které by po dobu své životnosti značilo pouze to, že STM je připojeno k serveru. Tato možnost je příliš komplikovaná, zejména kvůli tomu, že každé STM by si se serverem muselo domluvit unikátní port přes který toto nové spojení budou udržovat.

V rámci dodržení strategie co nejjednoduššího firmwaru STM je řešení číslo 2 nejvhodnější.

2.2.2 Použité technologie

Backend

Přestože očekáváme, že konečná webová aplikace bude poměrně malá a to jak požadovanou funkcionalitou, tak svým vzhledem, použití některého z webových backend frameworků nám může velice usnadnit práci, například v těchto aspektech:

- Zabudovaná podpora uživatelů - nám jen stačí přidat uživatelům odkazy na jejich zařízení.

- Zabudovaná podpora databázových systémů - většina backend frameworků poskytuje vrstvu abstrakce nad databázovými systémy. V praxi to znamená, že nikde v kódu nemusíme přímo vytvářet SQL dotazy, stačí používat funkční API pro databáze v programovacím jazyce, ve kterém je daný framework napsaný.

Existuje mnoho různých backend frameworků. Jsou to například NodeJS napsaný v Javascriptu, Ruby on Rails napsaný v Ruby, Django [11] napsaný v Pythonu a Laravel napsaný v PHP. Všechny tyto frameworky nám mohou usnadnit práci podobným způsobem, pro naši aplikaci se liší především tím, jaký programovací jazyk používají.

Vzhledem k tomu, že pro naši aplikaci je pro výběr konkretního frameworku relevantní pouze to, jaký programovací jazyk používá, a vzhledem k tomu, že autor má předchozí zkušenosti s Pythonem, vybereme Django.

Frontend

Vytváření vzhledu webu si usnadníme použitím knihovny Bootstrap [12]. Ta nám umožňuje snadno vytvořit moderně vypadající responsive webovou aplikaci.

Pro větší přehlednost požadujeme, aby se uživateli všechny jeho STM zobrazily na jednu stránku, která bude alespoň trochu interaktivní. Bylo by proto

vhodné použít nějaký frontend framework, například ReactJS nebo AngularJS. Integrace některého z těchto frontend frameworků do Djanga je ale příliš komplikovaná a náš web příliš malý pro to, aby se jejich použití vyplatilo. Proto budeme frontend vyvíjet pouze s pomocí Bootstrap a jQuerry [13].

2.3 Komunikace STM a web serveru

Po komunikačním systému vyžadujeme následující funkcionalitu:

- Periodicky posílat aktuálně naměřenou teplotu z STM na server.
- Synchronizovat čas mezi serverem a STM
- Synchronizovat nastavení intervalů na STM a na serveru. Pro tento bod potřebujeme mít funkční synchronizaci času.

Předpokládáme, že webový server běží na statické, veřejné IP adrese. Pokud STM připojíme ethernetovým kabelem do internetové sítě, bude mu přidělena dynamická IP adresa. Nemá smysl, aby si uživatel pořizoval statickou IP adresu kvůli STM. Vzhledem k tomu, že IP adresa STM je dynamická, server ji nemůže dopředu znát. Což znamená, že iniciátor komunikace musí být STM.

Pro komunikaci mezi webovým serverem a STM můžeme vybrat buď HTTP nebo vlastní protokol. Implementace a otestování vlastního protokolu by bylo příliš pracné, využijeme tedy existujícího HTTP. Navíc nám v takovém případě stačí implementovat HTTP pouze na straně STM.

Device ID Vzhledem k tomu, že chceme aby server podporoval připojení více STM najednou, potřebujeme tato STM nějakým způsobem identifikovat. Otázka je kdy a kde tento identifikátor vzít. Máme dvě možnosti:

- Identifikátor je zakódovaný do firmwaru STM.
- Identifikátor se dynamicky vytvoří na serveru při prvním přístupu konkrétního STM.

Po serveru požadujeme, aby autentizoval STM tj. aby ověřil, že příchozí zprávy chodí skutečně od nějakého STM. Pokud bychom vytvářeli device ID dynamicky na serveru, nemohli bychom ověřit, že příchozí zpráva přišla od STM. V našem případě bude lepší device ID zakódovat do firmwaru každého STM a rovnou ho uložit do databáze serveru. Na rozdíl od privátního klíče nehrozí při uložení device ID přímo do firmware STM žádné bezpečnostní riziko.

2.3.1 Synchronizace

Uživateli dovolíme nastavovat intervaly naráz na serveru i na STM, platit bude ale pouze nastavení, které bylo uloženo později (vzhledem k reálnému času). Jiná, uživatelsky pravděpodobně příjemnější, možnost by byla nedovolit uživateli současně nastavovat intervaly na serveru i na STM. Tato možnost je ovšem zbytečně složitá na implementaci.

Synchronizace času

Pro synchronizaci intervalů je potřeba, aby server i STM měli nastavený společný čas. Vzhledem k tomu, že implementace NTP [14] ale i jednoduššího a podobného protokolu SNTP [15] je na STM pracná, zvolíme jiný, jednodušší způsob synchronizace času mezi STM a serverem, který bude vycházet z TP [16].

V rámci synchronizace času postačí sekundové rozlišení. Vyjdeme z TP, který má sekundové rozlišení, a ještě ho zjednodušíme pro naše účely. Podobně jako je tomu u TP, budeme ze serveru posílat počet sekund, které uběhly od nějakého pevně stanoveného data. Na rozdíl od TP to ale ještě zjednodušíme tak, že tento počet sekund bude posílat server v rámci první HTTP odpovědi. Další rozdíl od TP bute ten, že naše počáteční datum nebude `1.1.1900`, ale `1.1.1970`. *Timestampem* budeme dále uvažovat číslo reprezentující počet sekund od `1.1.1970 00:00`, které se vejde do 4-bytové proměnné bez znamenénka. [3]

V souvislosti se vzdáleností STM od serveru a se zpožděním síťové komunikace se může stát, že čas na STM bude až o pár sekund opožděný. Kdyby se uživateli například podařilo na serveru uložit nové nastavení intervalů a pár mikro sekund poté nastavit nové intervaly na STM, mělo by STM správně nahrát intervaly na server, namísto toho ale intervaly stáhne ze serveru a svoje nastavení přepíše. Předpokládejme, že uživatel toto dělat nebude.

Z definice timestamp vyplývá, že STM nerozlišuje jednotlivé časové zóny. Pro jednoduchost budeme čas na STM interpretovat jako čas v zóně UTC. Pro správnou interpretaci času na STM v různých časových zónách bychom museli implementovat sofistikovanější časový protokol typu NTP.

Synchronizace intervalů

Synchronizace intervalů v situaci, kdy je STM připojeno k serveru a tím pádem má synchronizovaný čas, je přímočará. Následující text popisuje jak lze intervaly synchronizovat když byly na STM uloženy v momentě, kdy bylo STM offline.

Kromě samotných intervalů a jejich timestampu ukládá STM do své EEPROM ještě příznak značící jestli toto uložení proběhlo v momentě, kdy byl čas synchronizován se serverem.

Po synchronizaci času se serverem si STM opraví timestamp intervalů v EEPROM, není-li nastaven výše zmíněný příznak. Tato oprava probíhá tak, že k němu přičte rozdíl mezi timestampem přijatým ze serveru a timestampem reprezentující aktuální, realitě pravděpodobně neodpovídající, čas na STM. [4]

Tímto mechanismem jsme docílili toho, že z pohledu serveru má STM vždy nastavený reálný čas.

2.3.2 Zabezpečení

Požadavky

- Autentikace zpráv. Potřebujeme zajistit, aby server přijímal pouze zprávy od STM.

[3] Tato definice je totožná s Unix timestamp [9], který je nějakým způsobem podporován ve většině programovacích jazyků, což nám usnadní programování serveru.

[4] implicitně je na STM totiž nastaveno datum 1.1.1970

- Šifrování zpráv. Z hlediska zabezpečení by bylo nejlepší kdybychom šifrovali jak těla HTTP zpráv, tak jejich hlavičky. Pro jednoduchost implementace budeme šifrovat pouze těla zpráv.

HTTPS Komunikace webového serveru s STM by mohla být kompletně vedena v HTTPS. Pro HTTPS potřebujeme na STM TLS. To je poskytováno například knihovnou Cyclone-SSL [17], která je závislá na knihovně Cyclone-TCP/IP [18]. My už jsme ovšem pro STM použili knihovnu LwIP, protože STM poskytuje ethernetový ovladač přímo pro tuto knihovnu a tím pádem můžeme LwIP rovnou použít. Kdybychom ale chtěli použít Cyclone-TCP/IP, tak bychom museli ještě implementovat ovladač pro tuto knihovnu. Závěr je tedy takový, že HTTPS používat nebudeme.

Symetrické nebo asymetrické šifrování Kdybychom chtěli použít pouze asymetrické šifrování, server by měl privátní klíč a každé STM by šifrovalo komunikaci veřejným klíčem. Problém tohoto řešení je nedostatečná autentizace - veřejný klíč může získat každý, nejen STM. Pro naše účely je tedy lepší použít symetrické šifrování. Dodejme ještě že střídání asymetrického a symetrického šifrování, podobně jako je tomu u výše zmíněného HTTPS, jsme zavrhli zejména kvůli složitosti. Potřebujeme, aby uživatel mohl pohodlně zadat klíč do STM. Bude se nám tedy hodit šifrování s krátkým klíčem, což je například DES, které má klíč délky 8 bytů. Pro naše využití to je dostačující zabezpečení.

Pro symetrické šifrování potřebujeme, aby existovalo tolik párů klíčů, jako je STM zařízení. Jeden z páru má server a druhý STM. Následující text se zabývá problémem kde a kdy vzít tyto klíče.
Máme dvě možnosti:

- STM má zabudovaný klíč ve svém firmwaru a server má databázi všech těchto klíčů.
- Server generuje klíč na žádost uživatele.

Mít klíč zabudovaný do firmwaru STM není příliš vhodné pro případy, kdy by si uživatel pořídil STM „z druhé ruky“. Zejména je zde nižší zabezpečení nového vlastníka STM - bývalý vlastník by totiž teoreticky mohl z firmware dostat podobu klíče a podvrhovat komunikaci novému vlastníkovi.

Proto zvolíme řešení s generováním nových klíčů na žádost uživatele.

Generování klíčů

Server si ke každému STM potřebuje přiřadit klíč, což může udělat nejdřív v momentě, kdy mu od daného STM přijde *connect* zpráva (což je první zpráva, kterou STM pošle - viz 3.3.1). I kdyby dopředu věděl, který klíč patří kterému STM, nemůže vědět od kterého STM mu přišla *connect* zpráva, protože dopředu nezná IP adresy jednotlivých STM.

Postup generování klíčů

1. Server vygeneruje klíč s omezenou životností na žádost uživatele.

2. Uživatel zadá klíč do STM.

3. STM pošle šifrovanou *connect* zprávu serveru.

4. Server se pokouší dešifrovat zprávu všemi dosud nespárovanými klíči, dokud nedostane korektní *connect* zprávu tj. korektní HTTP hlavičku s tělem obsahujícím existující device ID.

5. Server má sprárované device ID s klíčem a IP adresou.

Výše zmíněný mechanismus generování klíčů a obecně vzato symetrické šifrování v tomto pojetí sice přidává bezpečnost do komunikace, ovšem na úkor rychlosti připojení STM k serveru. Server totiž po každé dostane *connect* zprávu z „neznámé“ IP adresy a musí se tudíž pokusit tuto zprávu dešifrovat postupně jednak všemi dosud nespárovanými klíči, ale také klíči ze všech offline STM. Proces připojení STM k serveru může být poměrně zdlouhavé, zvlášť v případě kdy je mnoho STM offline. To ale v našem případě nevadí.

3. Architektura

3.1 STM

3.1.1 Firmware

Tato kapitola rozebírá některá specifika vývoje firmwaru pro STM, v porovnání například s vývojem softwaru pro PC. Firmware musí být celkově dostatečně malý, aby se vešel do flash paměti STM (256 KB). Také musí být relativně spolehlivý.

Spolehlivost

Alokace paměti Vzhledem k tomu, že od firmwaru se očekává, že poběží bez vypnutí dlouhou dobu, je potřeba aby dodržoval některé invarianty. Tyto invarianty by měly zaručovat, že firmware nepřestane fungovat svojí vlastní vinou. Samozřejmě vždy se může stát, že přestane fungovat některá z hardwarových komponent. Takové případy se ale nedají rozumně ošetřit a nezbyde nám nic jiného, než celý systém vypnout, je-li to vůbec možné. Chyba firmwaru by mohla být například ta, že se dostane do stavu kdy dojde paměť. Tomuto stavu se dá poměrně snadno zabránit tak, že v žádné části firmwaru nebudeme používat dynamickou alokaci paměti. V našem případě by nepoužívání dynamické alokace paměti neměl být problém, protože o všech datech, která v STM mohou být uložena, dokážeme předem určit jejich maximální velikost.

Real-time vlastnosti Real-time vlastností firmwaru máme na mysli tu skutečnost, že jsme teoreticky schopni spočítat nebo naměřit délku maximální resp. minimální trvání každého tasku. Mohli bychom například deklarovat, že GUI task nesmí trvat déle než 100 ms. Pomocí watchdog periferie [8] bychom tuto maximální dobu trvání tasku snadno zajistili - v případě, kdyby task trval déle, watchdog by celý systém resetoval.

Pro náš firmware jsou ale tyto vlastnosti nedůležité a tak nebudeme měřit dobu trvání jednotlivých tasků, ani nastavovat watchdog.

Programovací jazyk Máme na výběr pouze mezi C a C++. Nemá moc smysl omezovat se pouze na C, navíc v rámci GUI se nám objektová hierarchie včetně virtuálních metod velice hodí. V rámci celé aplikace jsme se rozhodli nepoužívat typy z STL (C++ Standard Template Library), hodně z těchto typů totiž ve svých vnitřnostech používá operátor `new`. Mohli bychom si sice implementovat vlastní alokátor, který by měl povoleno alokovat paměť pouze v inicializační fázi programu resp. po určitý časový úsek. Po uplynutí inicializační fáze programu bychom alokování vypnuli a jakýkoli pokus o alokaci další paměti by znamenal chybu. Dodejme ještě, že množství alokované paměti po dobu inicializační fáze se dá spočítat při překladu a tudíž se jedná de facto pouze o statickou alokaci paměti, přestože v programu během inicializační fáze používáme typy, které můžou volat operátor `new`. Kvůli přílišné složitosti tohoto řešení se ale raději spokojíme s tím, že STL nebudeme používat vůbec. Navíc by nám toto omezené používání

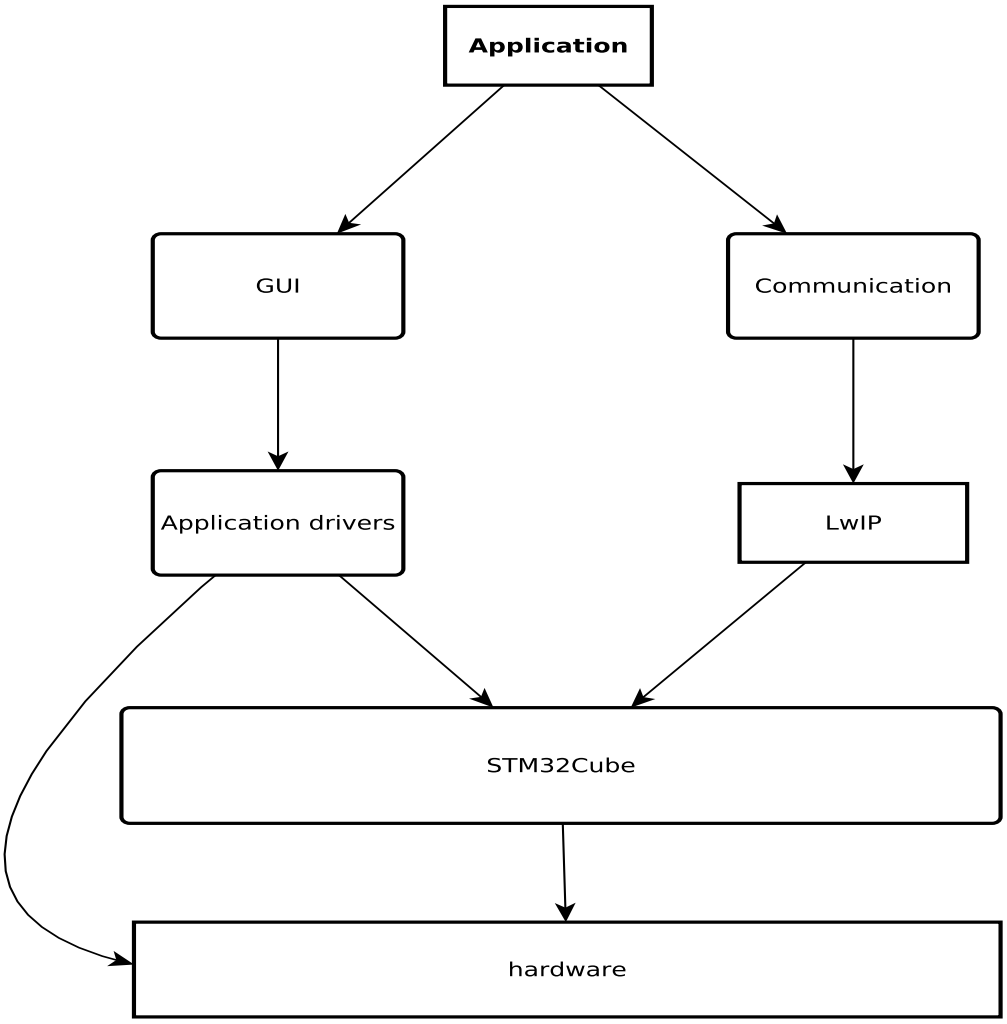
„dynamické alokace“ paměti pouze během inicializační fáze moc nepomohlo, protože nejvíc bychom to využili pravděpodobně při parsování HTTP zpráv a to už inicializační fáze rozhodně není.

Na závěr řekněme, že C++ používáme jako „C with classes“ a nikde v programu nepoužíváme ani operátor `new`, ani funkci `malloc`.

Nemožnost používání dynamické alokace paměti nám ztěžuje práci - všude totiž musíme používat pole s konstantní velikostí (některé z těchto velikostí jsou definovány v souboru `settings.hpp`, jinak jsou definované přímo v typech, které je používají) a indexy do těchto polí, a navíc musíme kontrolovat jestli pole nepřeteklo. V drtivé většině těchto případů by bylo daleko snazší použít například `std::vector`.

Přestože nepoužíváme STL, přidá nám C++ další abstrakci, která nám usnadní programování.

Struktura



Obrázek 3.1: Struktura firmwaru

Na obrázku 3.1 vidíme strukturu celého firmwaru rozdělenou do jednotlivých vrstev. Jednotlivé entity na obrázku odpovídají buď jednomu typu resp. třídě

(jako je tomu v případě `Application`), skupině tříd, nebo knihovně. `Application` je na nejvyšší úrovni abstrakce a provádí hlavní aplikační logiku. Jednak přímo pracuje se skupinou tříd `GUI`, pomocí které zobrazuje uživateli aktuální data, případně mu umožňuje nastavení některých dat, ale také pracuje přímo se skupinou `Communication`, která obstarává veškerou komunikaci se serverem. Skupina `Application Drivers` obsahuje aplikačně specifické wrappery hardwarových ovladačů nad `STM32Cube`, v jednom případě i třídu `OneWire`, která přistupuje k hardwaru přímo a ne přes `STM32Cube`.

Obrázek 3.1 slouží pouze jako nástin struktury firmwaru a nikoli jako detailní popis fungování jednotlivých typů. Bližší informace o struktuře skupiny `Communication` je uvedena v kapitole „Implementace komunikace“, struktura skupiny `GUI` je popsána v kapitole „GUI“.

3.1.2 Tasky

V následujícím textu budeme označovat *task* jako sémanticky samostatné výpočetní vlákno. V podstatě je to totéž jako task v kontextu RTOS. V STM je paralelismus pouze virtuální, protože procesor je jednojádrový.

Naše aplikace potřebuje provádět 3 tasky:

- **GUI task** má na starosti periodicky iterovat přes všechny GUI elementy na displeji a v případě potřeby je překreslit.
- **User input task** má na starosti zjišťovat jestli uživatel zadal nějaký vstup, ať už je to zmáčknutí joysticku, nebo jiného tlačítka.

  Uživatelský vstup lze zpracovávat i pomocí interrupt handlerů - joystick ale i jiná tlačítka lze nastavit tak, aby při stlačení vyvolali interrupt. Pro naši situaci to ale není výhodné, protože bychom museli mít více různých interrupt handlerů pro každý možný vstup. Lepší je nastavit periodický task tak, aby iteroval všechny možné vstupy.
- **Ethernet task** má na startosti zpracování paketů z ethernetové periferie. Je potřeba periodicky volat dvě funkce z LwIP knihovny - `ethernetif_input`, která se stará o nízko-úrovňový příjem nebo odesílání paketů resp. bajtů z ethernetové periferi, a `sys_check_timeouts`, která kontroluje platnost různých dočasných hodnot, jako je třeba ARP tabulka, a v případě potřeby je obnoví.
- **RTC task** má na starosti zpracovávání sekundového interruptu a jednou za minutu změřit aktuální teplotu.

Vzhledem k tomu, že firmware je jediné, co v STM poběží, je potřeba aby svůj běh neukončil hned po prvním vykreslení obrazovky resp. zpracování pár paketů. Potřebovali bychom mít někde ve firmware nekonečný cyklus, který bude zpracovávat naše tasky. Takovému nekonečnému cyklu budeme říkat *mainloop*. V rámci firmware pro embedded zařízení, který nemá RTOS, je použití *mainloop* běžné.

Kromě *mainloop* můžou být tasky vykonávané ještě v rámci interrupt handleru některé periferie, například časovače, nebo stisknutí tlačítka. Interrupt handlery

mají obecně tu výhodu, že jsou spuštěny vždy po určité události - v případě časovače je to uplynutí periody.

Ne všechny tasky je vhodné provádět v kontextu interrupt handleru. Například podle dokumentace LwIP není doporučeno volat jakoukoli funkci, která má odesílat nebo přijímat pakety z interrupt handleru. Proto je potřeba ethernet task provádět vždy v rámci mainloop. Naopak user input task je vhodné provádět v rámci interrupt handleru časovače nastaveného na rozumnou periodu. Pokud by se totiž měl provádat z mainloop, není jasné jak přesně by měl fungovat, protože čtení vstupu od uživatele je samo o sobě blokující funkce.

Softwarové časovače

Některé komponenty firmwaru musejí být periodicky notifikovány o vypršení určitého časového intervalu. V podstatě by takové komponenty využili fungování standardního hardwarového časovače. Vzhledem k tomu, že hardwarových časovačů je omezeně mnoho a jejich inicializace je pro tyto případy zbytečně složitá, hodí se vytvořit softwarové časovače. Jejich myšlenka je jednoduchá. Z *mainloop* kontrolujeme, jestli nedošlo k jejich vypršení a pokud ano, tak zavoláme jejich „timeout“ funkci. Pomocí funkce `HAL_GetTick` můžeme získat aktuální počet tiků systému v milisekundách, takže implementace softwarového časovače je skutečně jednoduchá.

Příklad použití softwarového časovače je `MainFrame`, který musí periodicky zjišťovat stav síťového připojení, aby ho mohl zobrazit uživateli.

3.1.3 Eventy

V této kapitole rozebereme některé důležité události, které mohou v rámci firmwaru STM vzniknout.

Mezi tyto události patří:

- Změna intervalů na STM. Jako reakci na tuto událost by STM mělo nově nastavené intervaly uložit do EEPROM a dále tyto intervaly poslat na server v případě, že na serveru jsou nastavené starší intervaly.
- Příjem intervalů ze serveru o které samo STM požádalo. Reakce na tuto událost je uložení intervalů do EEPROM.
- Připojení k serveru. Myšleno ne pouze navázané TCP spojení, ale spojení ve smyslu naší aplikace tj. server poslal svůj aktuální čas ve formátu timestampu. STM reaguje tak, že si právě přijatý timestamp nastaví jako svůj vlastní čas, tedy uloží timestamp do RTC.
- Právě naměřená teplota. Tuto hodnotu chceme zobrazit uživateli na STM a také ji poslat na server.
- Chyba síťové komunikace. Toutu chybou máme na mysli v podstatě libovolnou síťovou chybu od vypojení ethernetového kabelu až po nemožnost naparsování HTTP response. Na tuto událost STM reaguje restartováním celé komunikace. Kvůli složitosti implementace nemá vůbec smysl aby si STM pamatovalo, v jakém stavu chyba nastala a zkoušelo komunikace restartovat právě od tohoto momentu.

- Zapojení ethernetového kabelu. Jak jsme již zmiňovali v kapitole analýza, uživateli dáváme možnost zapojit nebo odpojit ethernetový kabel za běhu aplikace. Zapojení ethernetového kabelu by v ideálním případě mělo ihned odstartovat pokus o spojení se serverem a výsledek tohoto pokusu oznámit uživateli. Připojit se k serveru může STM pouze pokud má uložený klíč v EEPROM. Pro přehlednost a jednoduchost aplikace stanovíme ještě další podmínku, která musí být splněna před pokusem o připojení k serveru - aktuální obrazovka musí být `MainFrame`.

- Nastavení klíče. Tato událost nastává v momentě, kdy je aktuální obrazovka `KeyFrame`, a uživatel stiskne tlačítko „submit“. STM na to reaguje uložením klíče do EEPROM a také pokusem o připojení k serveru, pokud jsou všechny podmínky pro tento pokus splněny.

Každá z výše uvedených událostí je reprezentovaná třídou v programu. Problém s připojením je například reprezentován třídou `CommunicationErrorEvent`, nastavení klíče potom třídou `KeySavedEvent`, apod. Události mohou být generované v různých částech programu, `MeasuredTemperatureEvent` je generovaný například v `TempController`. Všechny tyto události musí být po vzniku předány třídě `Application`, která v tomto případě slouží jako centrální autorita ošetřující všechny druhy událostí.

Výhoda tohoto návrhu je v tom, že reakce na všechny důležité události máme na jednom místě. Nevýhoda je potom taková, že nesmíme zapomínat události generovat.

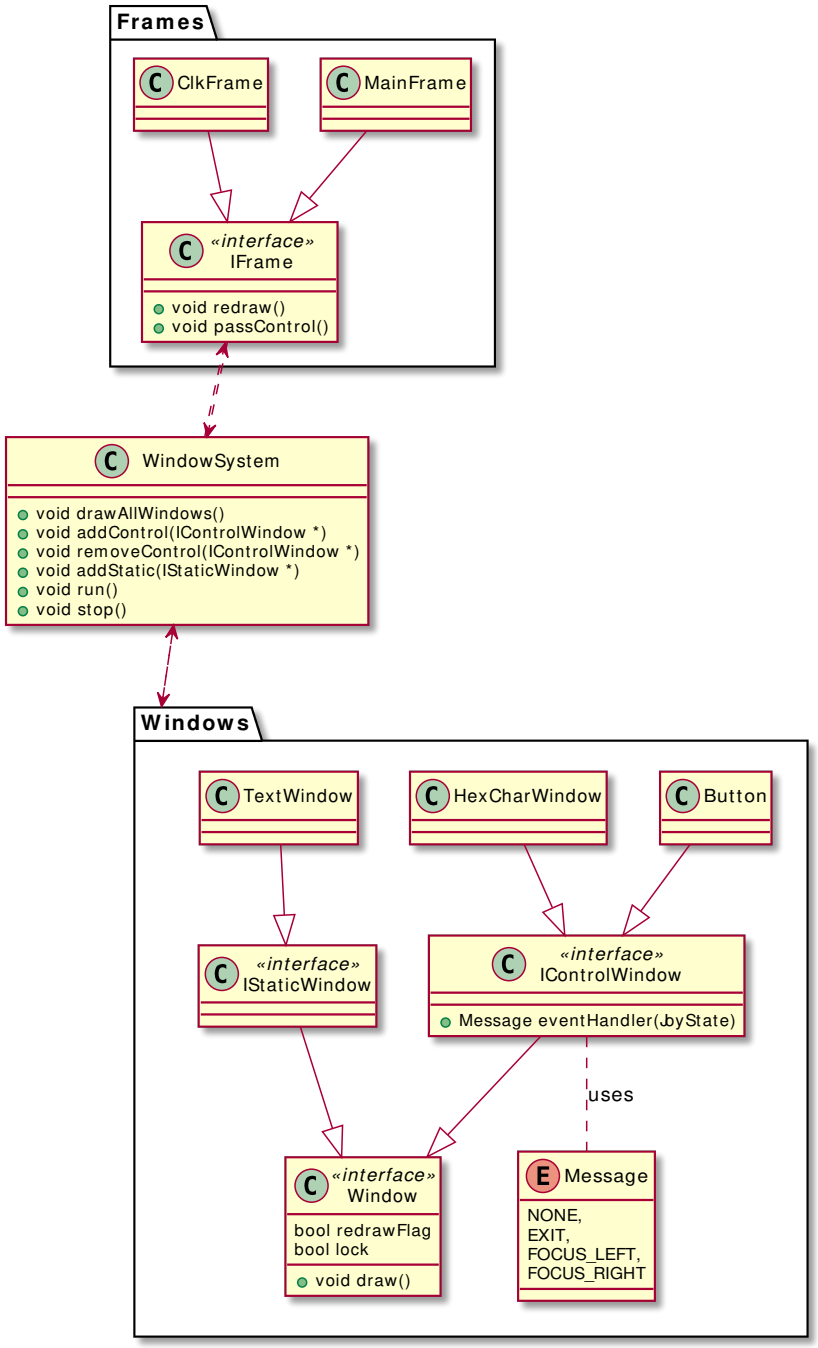
3.1.4 GUI

V rámci kapitoly 2.1.2 jsme specifikovali požadavky, které máme na uživatelské rozhraní. V této kapitole se detailněji podíváme na implementaci celého GUI.

Obrázek 3.2 zobrazuje hierarchii základních tříd a vztahy mezi nimi. Pro jednoduchost zobrazuje pouze pár konkrétních tříd jako příklad: `TextWindow` jako statické okno, které pouze zobrazuje text a `HexCharWindow` jako kontrolní okno, které je součástí `KeyFrame` a zobrazuje právě jeden hexadecimální znak, jehož hodnotu může uživatel zvětšovat či zmenšovat. Celé GUI je tvořeno dvěma základními druhy objektů - *windows* a *frames*. Každý frame reprezentuje jednu obrazovku tj. co všechno je v daný moment zobrazeno na displeji. Windows jsou velice jednoduchá okna - typicky zobrazující pouze neohraničený text resp. číselnou hodnotu. Frame obsahuje několik oken a definuje, kde přesně mají tato okna být zobrazena a jakého typu mají být. Každý frame má svůj `WindowSystem`, přes který může sám do sebe vkládat okna. Kromě toho, že `WindowSystem` zpracovává vstup, na základě kterého přepíná mezi jednotlivými okny, funguje také jako prostředník mezi frame a jeho okny.

Windows

Window může být buď *statické* (dědí od třídy `IStaticWindow`), nebo *kontrolní* (dědí od třídy `IControlWindow`). Statické okno slouží pouze pro zobrazování libovolného textu, kontrolní okno může uživatel „nakliknout“ a změnit jeho hodnotu. Jediný uživatelský vstup je joystick a řekněme, že jeho zmáčknutím



Obrázek 3.2: Hierarchie základních tříd v GUI

doprava nebo doleva se uživatel přesune mezi jednotlivými okny, a zmáčknutím nahoru nebo dolů může uživatel měnit hodnotu v právě nakliknutém okně. Přímé stisknutí joysticku má vliv pouze na okna typu `Button`, která typicky slouží buď jako potvrzení všech zadaných hodnot v rámci jedné obrazovky, nebo jako „exit“ tlačítko. Možnost nakliknutí a změny nějaké hodnoty kontrolního okna se do objektového návrhu promítá tak, že `IControlWindow` má čistě virtuální metodu `Message eventHandler(JoyState joyState)`. Zjednodušeně řečeno každé okno pomocí přetížení této metody specifikuje svoji reakci na vstup, kde tato reakce může být buď čistě interního charakteru tj. změna hodnot daného okna, nebo

může vrátit jednu z hodnot `Message::FOCUS_LEFT` nebo `Message::FOCUS_RIGHT`, čímž říká, že nakliknuté má být kontrolní okno, které je vlevo nebo vpravo od tohoto okna, případně může vrátit `Message::EXIT`, čímž říká že má být aktuální frame vypnut.

WindowSystem

`WindowSystem` je v podstatě kontejner pro všechna okna v rámci jednoho frame a ještě se stará o výše zmíněnou logiku. Frame při inicializaci pouze přidává kontrolní a statická okna do `WindowSystem`. Za běhu potom `WindowSystem` podle vstupu buď přepíná mezi kontrolními okny, nebo přímo vypne aktuální frame.

Zde je důležité si uvědomit, že zpracování vstupu probíhá v *user input task*, což je interrupt handler jednoho z časovače, který tento interrupt generuje jednou za 100 ms. Z pohledu kódu je tedy zpracování vstupu a všechno s tím spojené asynchronní. Protože zpracování vstupu běží v jiném kontextu než vykreslování, je potřeba rozlišovat metody podle toho ve kterém kontextu jsou volány. Označme metody volané v kontextu zpracování vstupu jako *callback* metody a definujme pro ně strukturu, kterou rozebereme v následujícím odstavci.

Callback metody Občas v rámci jednoho frame chceme reagovat na stisknutí některého z tlačítek. Například při stisku „Exit“ v `SetIntervalFrame` bychom chtěli ukončit celý frame a uložit uživatelem zadané hodnoty. Zavedeme několik rozhraní se dvěmi čistě virtuálními metodami: `callback` a `register`. [1] `register` metoda by měla objekt, který chce dostávat callbacky zaregistrovat k jejich zdroji. `callback` metoda potom implementuje reakci na konkrétní callback. Všechny tyto rozhraní dědí od `ICallbackReceiver`.

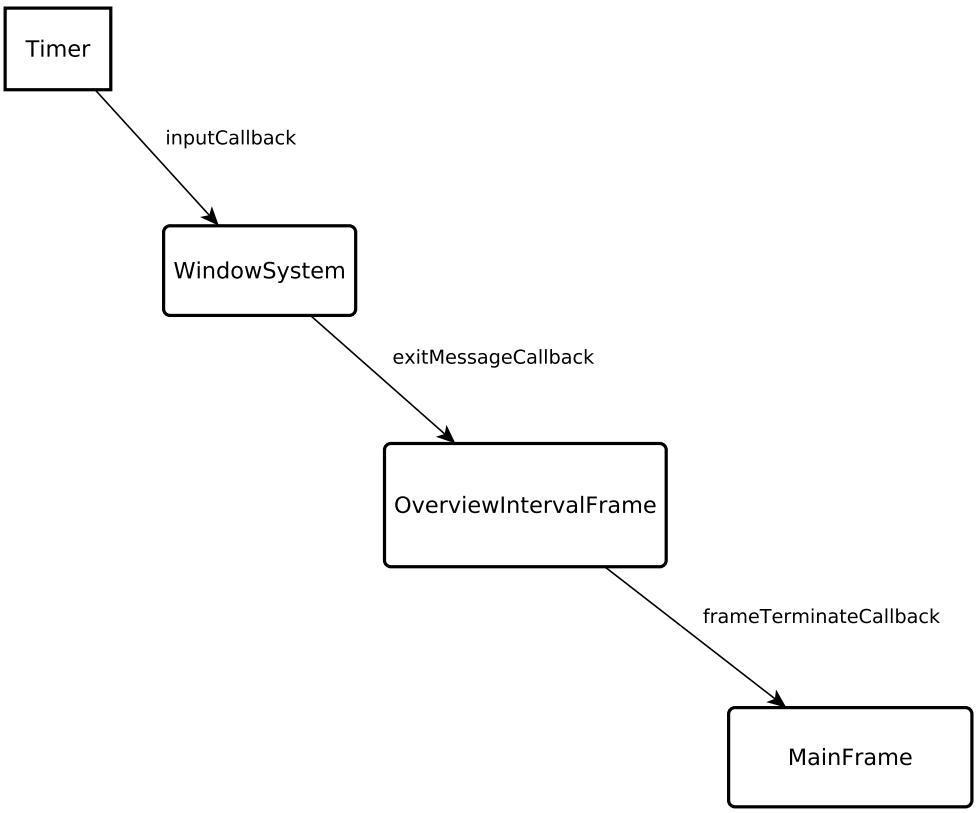
Myšlenka je taková, že když chceme, aby objekt A dostával určité callbacky od objektu B, musíme nejprve objekt A u objektu B registrovat jako „callback receiver“ a potom definovat metodu která na tento callback bude reagovat.

Místo toho, abychom vyjmenovali všechny možné typy callbacků a jejich parametry, zpřesníme strukturu callback metod na příkladě.

Diagram 3.3 ilustruje situaci, kdy je uživatel v `OverviewIntervalFrame`, stiskne tlačítko „exit“ a dostane se zpět do `MainFrame`. Proces na diagramu je následující:

1. Vygenerování interruptu časovačem.
2. Zavolání metody `IO::task`, ve které dojde k vytvoření callback objektu s parametrem.
3. Zavolání metody `WindowSystem::inputCallback`, která tento callback předá aktuálně nakliknutému oknu (v tomto případě je to `exitButton`), které vrátí `Message::EXIT` ze svého `eventHandler`.
4. Zavolání metody `OverviewIntervalFrame::exitMessageCallback`
5. Zavolání metody `MainFrame::frameTerminateCallback`, ve které MainFrame určí který frame skončil a nastaví sám sebe jako aktuální frame.

[1] Přesný název konkrétních metod se liší, ale callback nebo register je jejich součástí



Obrázek 3.3: Příklad hierarchie volání callback metod

Celý tento „framework“ je navržen s ohledem na co nejsnazší specifikaci vzhledu i chování nově přidávaného framu, ale také s ohledem na to, že čtení vstupu probíhá v kontextu interrupt handleru, kdežto vykreslování oken probíhá v *mainloop*. Mohlo by se tedy stát, že vykreslování okna bude přerušeno, v interrupt handleru se změní jeho hodnoty, a vykreslování se opět obnoví - toto může vést k tomu, že okno vykreslí špatné hodnoty. Každé okno má tedy zámek, který je zamčen před voláním metody `draw` nebo metody `eventHandler`. O toto zamykání se nemusí starat programátor při definování nového okna, děje se totiž už na úrovni třídy `Window`.

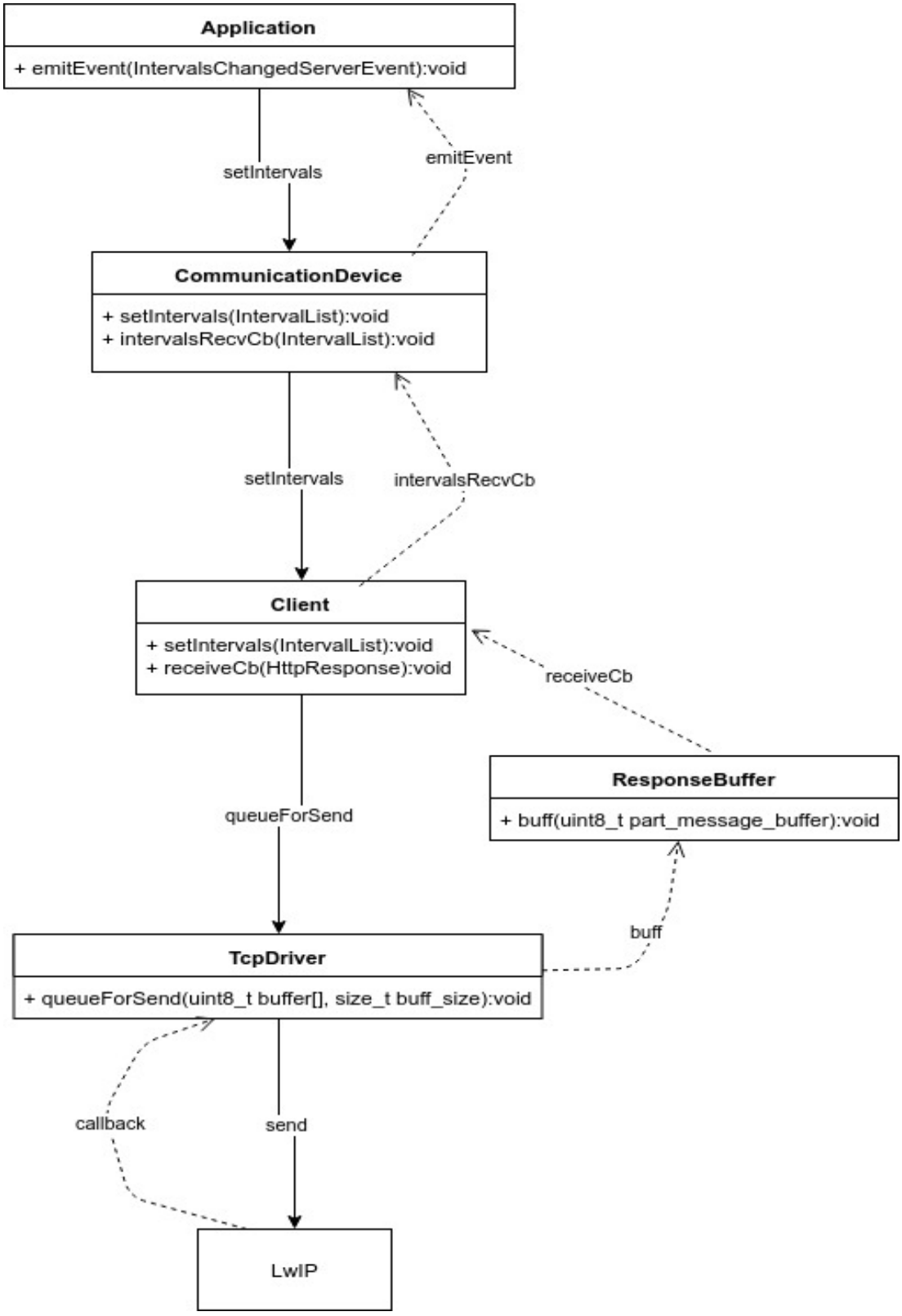
Pokud oknu změníme hodnotu a chceme, aby bylo v rámci *GUI task* překresleno, musíme mu explicitně nastavit `redrawFlag`. Díky `redrawFlag` máme zajištěno i to, že nemusíme zbytečně překreslovat okna, která to nepotřebují. Přeci jen je vykreslování na displej časově poměrně náročné.

3.1.5 Implementace komunikace se serverem

V této kapitole se detailněji podíváme na implementaci komunikace se serverem.

V diagramu 3.4 jsou zobrazeny nejdůležitější třídy, vztahy mezi nimi a zároveň je tam zobrazen i typický proces nastavení intervalů na STM a stažení intervalů ze serveru, který bude popsán později. Nejprve popíšeme třídy v diagramu a jejich funkcionalitu.

- `CommunicationDevice` který je objektovou reprezentací samotného STM



Obrázek 3.4: Diagram tříd v rámci komunikace

a `Application` do něj ukládá naměřenou teplotu pro odeslání na server a intervaly pro výměnu se serverem. CommunicationDevice notifikuje Appli-

cation o dvou důležitých událostech - ConnectedEvent a IntervalsChangedServerEvent - tedy připojení k serveru a přijetí intervalů ze serveru.

- `Client` je HTTP klientem specifickým pro naše použití. Jednak do něj CommunicationDevice přeposílá data, které má od Application, a jednak Client notifikuje CommunicationDevice o důležitých událostech týkajících se komunikace se serverem jako jsou: „intervaly byly odeslány“, „teplota byla odeslána“, „intervaly byly přijaty“, apod.
- `TcpDriver` je wrapper pro LwIP knihovnu.
- `ResponseBuffer` reprezentuje buffer pro HTTP odpovědi ze serveru. TcpDriver přeposílá vše, co přijme, do ResponseBuffer. Ten si tato data ukládá dokud nedostane celou HTTP zprávu. Typicky se totiž stává, že server neodešle celou HTTP odpověď v jednom paketu a proto je potřeba jednotlivé pakety ukládat a později z nich poskládat celou HTTP odpověď. V momentě, kdy ResponseBuffer má celou odpověď, notifikuje o tom Clienta. ResponseBuffer také potřebuje vědět jestli očekávaná odpověď od serveru má mít i tělo nebo pouze HTTP hlavičku.

Směrem dolů od Application až po LwIP probíhá odesílání dat na server resp. zařazení těchto dat do interních front LwIP - k samotnému odeslání dat typicky dojde později. Směrem nahoru od LwIP, přes TcpDriver, ResponseBuffer až po Application probíhá příjem dat ze serveru. Z pohledu kódu se dá směr dolů považovat za „synchronní“ a směr nahoru za „asynchronní“, proto jsou také všechny metody, které jsou volané směrem nahoru, definovány jako callback metody.

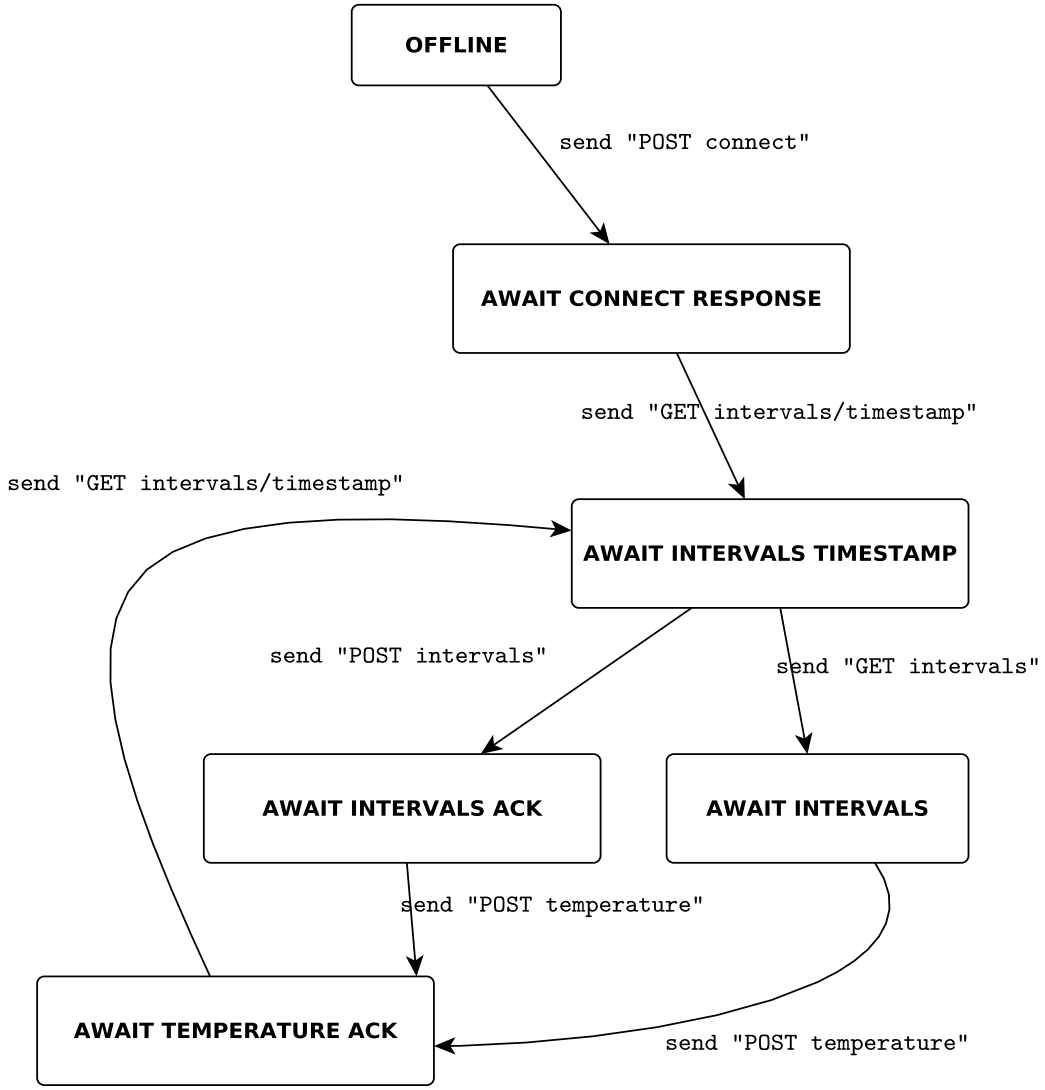
Na diagramu jsou znázorněny dva typické procesy:

Posílání intervalů na server Proces probíhající směrem dolů bychom mohli nazvat jako „uložení intervalů do interních struktur Clienta k budoucímu porovnání s intervaly serveru“ (porovnávat se ve skutečnosti bude pouze timestamp intervalů). Jakmile se intervaly dostanou do Clienta, musí zde Client počkat, dokud se jeho *komunikační cyklus* (zmíněný dále v této kapitole) nedostane do stavu, ve kterém porovnává timestamp svých intervalů a timestamp intervalů ze serveru. Nezávisle na tom, jak toto porovnání dopadne, musí Client poslat buď GET request, nebo POST request na server přes TcpDriver.

Příjem intervalů ze serveru Je proces probíhající směrem nahoru. LwIP předává přijatý payload TCP paketů do TcpDriver, který ho dále předává do ResponseBuffer. Jak již bylo zmíněno, ResponseBuffer postupně ukládá části HTTP odpovědi, dokud nesestaví celou, potom notifikuje Client pomocí callback metody. ResponseBuffer už HTTP odpověď naparsoval a Client na základě toho, v jakém stavu *komunikačního cyklu* se nachází, notifikuje CommunicationDevice (na diagramu notifikuje CommunicationDevice o přijetí intervalů ze serveru pomocí callback metody `intervalsReceivedCb`). CommunicationDevice dále vygeneruje `IntervalsChangedServerEvent` a pošle do Application k následnému šíření.

Client

Kromě toho že Client je HTTP klientem, musí také být implementován tak, aby byl kompatibilní s LwIP callback API.



Obrázek 3.5: Komunikační cyklus

Na diagramu 3.5 je znázorněn *komunikační cyklus* a jednotlivé stavy tohoto cyklu. Zjednodušeně řečeno se v rámci celého cyklu nejprve buď pošlou intervaly na server nebo stáhnou ze serveru - záleží na tom, jestli má STM větší timestamp než server - poté se pošle teplota na server. Pokud cyklus probíhá poprvé, musí se nejprve STM k serveru přihlásit. Celý cyklus je opakován v určitých časových intervalech. [2]

Mezi libovolnými dvěma stavy může dojít k chybě. Chyby ošetříme tak, že celý cyklus restartujeme od stavu `OFFLINE`. Mohli bychom sice uložit stav před chybou a po zpracování chyby pokračovat opět od tohoto stavu, bylo by to ale

[2]Konkrétně jsou to 2 sekundy, je pro to použit softwarový časovač

příliš komplikované. Navíc v našem případě je během celého cyklu přeneseno relativně malé množství dat, takže restartování od úplného začátku nás skoro nic nestojí.

3.1.6 EEPROM

EEPROM používáme jako persistentní datové uložiště, do kterého ukládáme konfiguraci intervalů a privátní DES klíč. Kromě samotných intervalů a klíče je ještě potřeba do EEPROM ukládat metadata - například počet intervalů, timestamp intervalů nebo příznak značící jestli klíč je uložen. [3]. Je zbytečně složité kvůli těmto pár datům a metadatům používat nějaký file systém.

Co se týká implementačních detailů, tak `EEPROM` je třída, která se stará o ukládání dat do EEPROM a mimo jiné obsahuje definice adres, na které jsou ukládány různá data a metadata. EEPROM při inicializaci nevyžaduje žádný specifický formát, což mimo jiné znamená, že vymazat všechna data z EEPROM znamená vyplnit celou EEPROM nulovými bajty.

Vymazání celé EEPROM je také něco, co je vhodné udělat předtím, než se uživatel STM rozhodne toto zařízení předat někomu jinému.

3.2 Webový server

Webový server je rozdělený do dvou na sobě nezávislých komponent `user_interface` a `stm_communication` tak, jak je to znázorněno na obrázku 3.6. `user_interface` zprostředkovává veškerou komunikaci mezi uživatelem (klientem) a jeho STM zařízeními sestávající z nastavování intevalů a čtení aktuálně naměřené teploty. `stm_communication` přijímá aktuálně naměřenou teplotu od STM zařízení a vyměňuje si s nimi intervaly. Obě komponenty čtou a zapisují data do společné databáze. Tím je nepřímo zajištěna komunikace mezi uživatelem a jeho zařízením skrz server.

V kapitole analýza jsme se rozhodli použít Django jako backend framework, k vývoji frontend používáme pouze Bootstrap a jQuery. Vzhledem k požadované interaktivitě frontend jsme si tímto rozhodnutím trochu zkomplikovali práci. Kdybychom chtěli přidat další funkcionalitu, jako je například zobrazení historie pro STM, bylo by vhodné do serveru integrovat nějaký frontend framework.

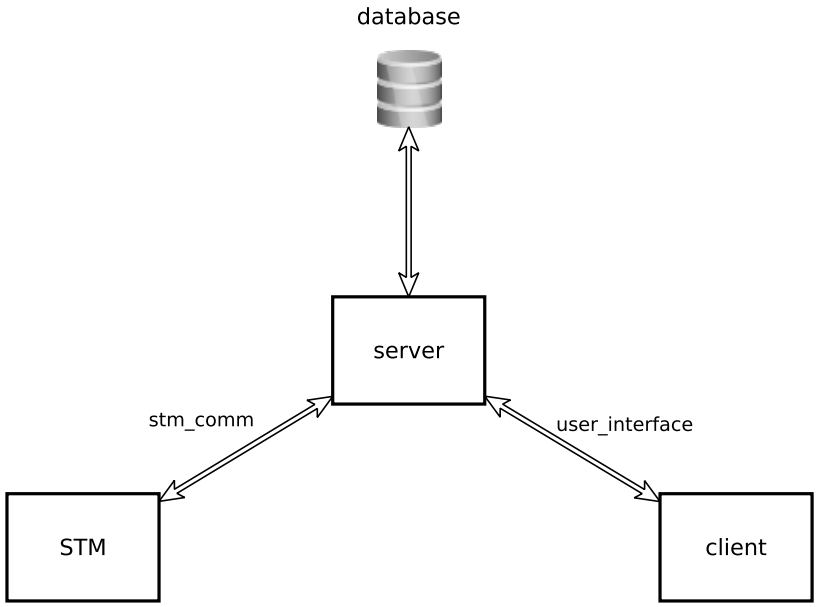
Samotná implementace frontendu není pro tuto práci příliš důležitá, proto se jí nebudeme zabývat. Implementace backend už důležitá je, ovšem v porovnání s implementací STM, kde na serveru máme k dispozici velikou abstrakci, máme zjednodušenou práci.

3.2.1 Databázové schéma

Databázové schéma 3.7 ukazuje základní objekty a vztahy mezi nimi.

Na server se může registrovat libovolný uživatel zadáním uživatelského jména, hesla a potvrzením emailové adresy. Každý uživatel si může přidat omezeně mnoho zařízení. Každé STM zařízení má svoje ID přiřazeno ještě před tím, než se dostane uživateli do ruky. Kromě toho, že toto ID je uloženo v databázi na

[3] DES klíč totiž může mít všech bajty nulové



Obrázek 3.6: Komponenty webového serveru

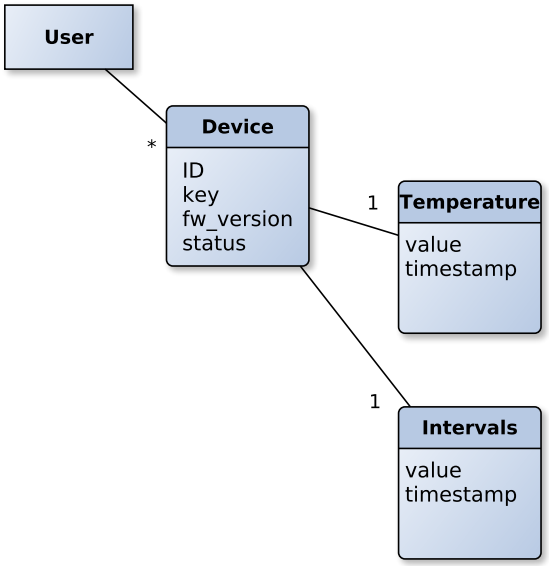
serveru, je potřeba aby bylo zakódovano do firmwaru konkrétního STM, což by neměl být problém. Každé STM má také přiřazenou verzi firmwaru, to z důvodu že by STM mohlo být teoreticky aktualizováno na dálku - viz [19]. Server a ani STM toto ale zatím nepodporují. `status` může být buď `connected` nebo `offline`. `key` se k STM přiřadí až po tom, co se poprvé přihlásí k serveru.

Co se týká jednotlivých položek, v našem případě `Temperature` a `Intervals`, tak ty se dělí na dva typy - *actual* a *config*. Actual položky jsou takové, které STM periodicky posílá na server a ten má za úkol je zobrazit uživateli. Config položky nejsou pouze ke čtení, ale i k zápisu. Config položky jsou synchronizovány mezi serverem a STM tak, že na obou stranách platí pouze hodnota s novějším timestampem. Každá položka v databázi má svůj timestamp a `value`. Temperature představuje teplotu, tedy jeho value je prostě číslo. Na druhou stranu Intervals představuje nastavení teplot po různé časové intervaly, což znamená, že ve value musí být uložená nějaká komplikovanější hodnota než je číslo. Vhodné řešení je ukládat do intervals textový řetězec v JSON formátu, zejména kvůli tomu, že v rámci Pythonu a Javascript se JSON řetězec velice jednoduše parsuje.

3.3 Komunice STM a web serveru

3.3.1 Specifikace HTTP zpráv

Tato sekce popisuje obsah jednotlivých HTTP zpráv mezi STM a serverem. Pro jednoduchost zde nejsou téměř vůbec uvedeny obsahy hlaviček. Důležité jsou zejména URL requestů a těla zpráv, kde konkrétní URL jsou z ilustrativních důvodů zjednodušeny. Těla zpráv jsou ještě navíc šifrována ale pro jednoduchost jsou všechny zprávy zobrazeny tak, jak by vypadaly bez použití jakéhokoli šifrování.



Obrázek 3.7: Databázové schéma

Fáze připojování

Nejprve je potřeba vyřešit připojení STM k serveru.

STM

```
POST /connect
Content-Type: text/plain

<device_id>
```

server

```
HTTP/1.1 200 OK
Content-Type: text/plain

<server_real_time>
```

`server_real_time` je timestamp odeslaný serverem sloužící k synchronizaci času mezi STM a serverem.

Odesílání naměřené teploty

STM

```
POST /actual/temp
Content-Type: text/plain

<temp_timestamp>
<temp_float>
```

Kde `temp_timestamp` je timestamp značící kdy byla teplota naměřena a `temp_float` je teplota v desetinné přesnosti.

Výměna intervalů

STM

```
GET /config/intervals/timestamp
```

server

```
HTTP/1.1 200 OK
Content-Type: text/plain

<interval_timestamp>
```

`interval_timestamp` je timestamp intervalů na serveru. STM porovná přijatý `interval_timestamp` s timestampem svých intervalů a pokud má novější nastavení než server, tak je odešle:

STM

```
POST /config/intervals
Content-Type: application/octet-stream

<intervals_timestamp><intervals>
```

Na rozdíl od odstatních zpráv je zde `intervals_timestamp` v binárních podobě jako 4 bajtové neznaménkové číslo, formát `intervals` bude zmíněn později.

Pokud má starší intervaly než server, tak si je od serveru vyžádá:

STM

```
GET /config/intervals
```

server

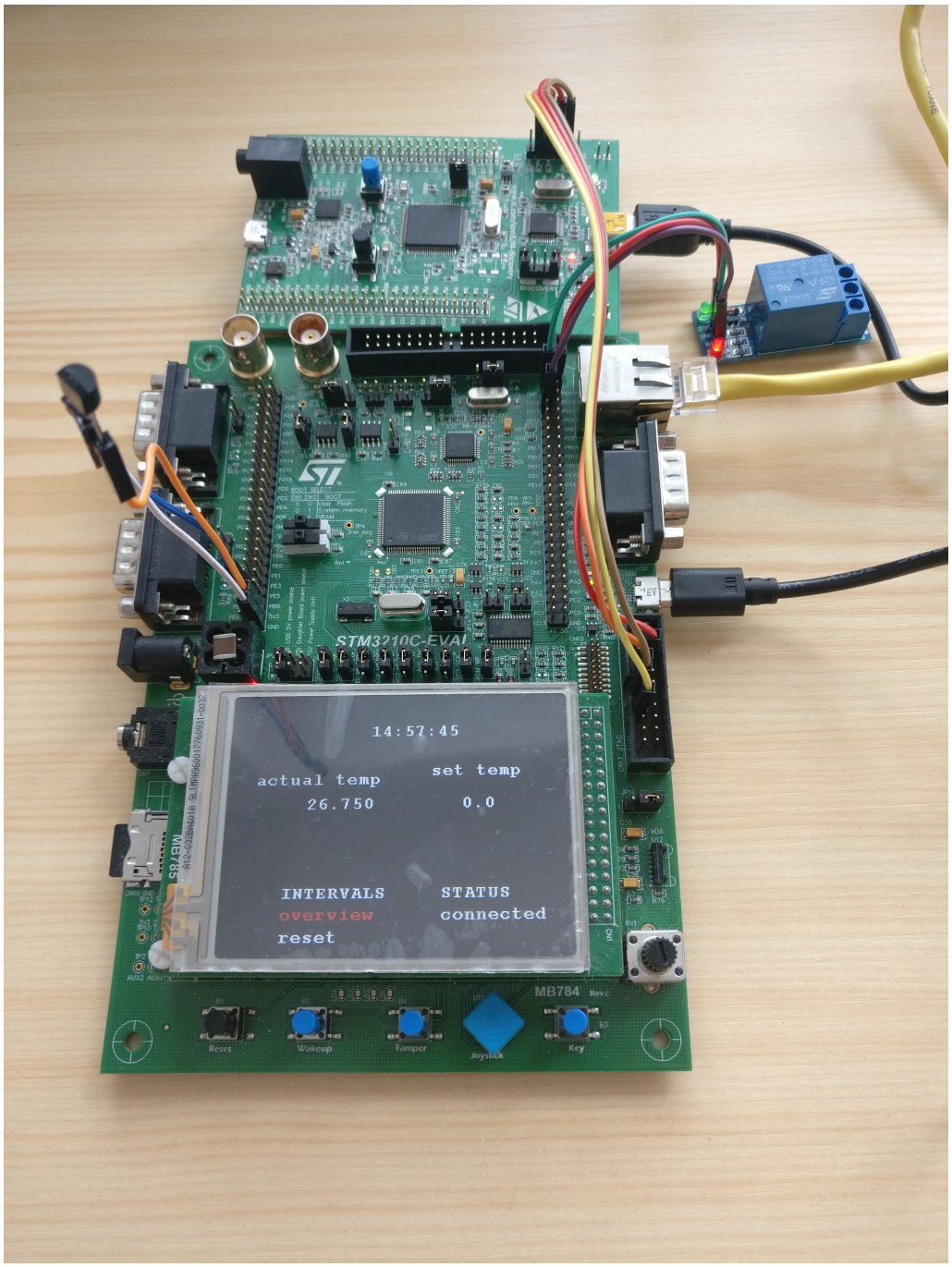
```
HTTP/1.1 200 OK
Content-Type: application/octet-stream

<intervals>
```

Formát intervalů Formát intervalů je stejný jako formát, ve kterém si samo STM intervaly ukládá do své EEPROM a má následující podobu: `<from> <to> <temp>`, kde `from` a `to` jsou časy v minutách v rámci dne a `temp` je teplota. Každá položka je reprezentována 4 bajtovým číslem bez znaménka.

4. Práce se systémem

V této kapitole ukážeme, jak se systém používá jako celek. Konkrétně ukážeme a popíšeme fotografii STM se zapojeným teplotním sensorem a relé, a dále screenshot webových stránek.

4.1 STM



Obrázek 4.1: Foto STM3210C-Eval board

Na obrázku 4.1 vidíme dvě STM zařízení. To s displejem je STM3210C-Eval

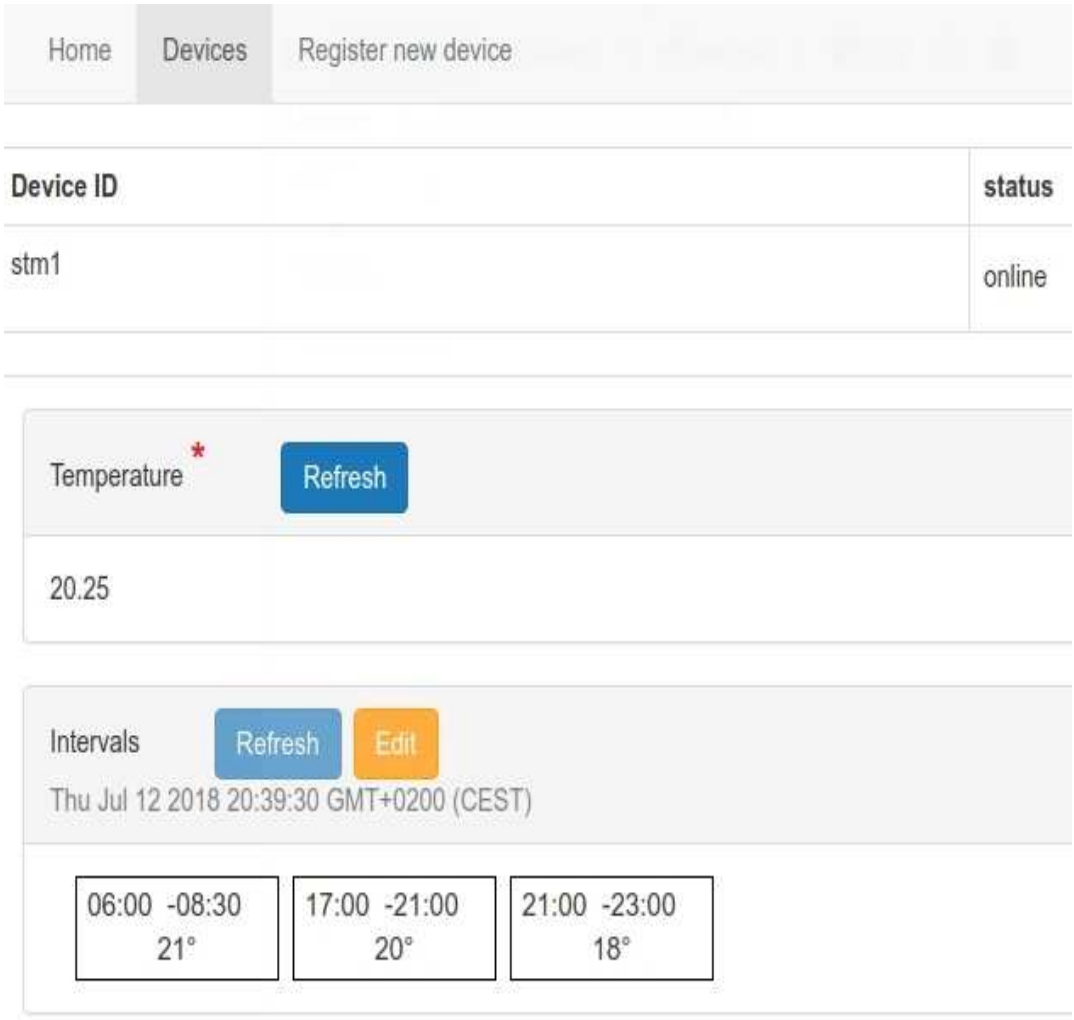
board, které používáme v celé práci. To druhé je STM32F4-Discovery [20], které používáme jen kvůli tomu, že je na něm zabudovaný ST-Link, přes který jednak nahráváme firmware do STM3210C-Eval boardu a jednak ho používáme pro ladění. V reálném použití nebude použití STM32F4-Discovery nutné.

Do STM3210C-Eval boardu (dále jen STM) je zapojen teplotní senzor Dallas DS18B20 [2] a relé. Konkrétní zapojení do GPIO pinů se dá velice snadno změnit ve zdrojovém kódu. Dole pod displejem STM je několik tlačítek, joystick a čtyři LED diody. Z toho reálně využíváme pouze joystick jako vstup a červenou diodu jako notifikaci o tom, že se v STM stala nějaká chyba. Černé „reset“ tlačítko slouží jako restart firmwaru STM.

Na displeji je aktuálně zobrazen `MainFrame`.

4.2 Webový server

4.2.1 Zobrazení všech STM



Obrázek 4.2: Screenshot „devices“ stránky

Na stránce 4.2 je vidět jedno zařízení s ID „stm1“, které je zrovna online. U

položky temperature je červená hvězdička, která značí že server přijal od STM novou hodnotu, která je připravena k aktualizaci pomocí „refresh“ tlačítka. U položky intervals je kromě „reresh“ tlačítka vidět i „edit“ tlačítko, pomocí kterého můžeme intervaly libovolně editovat, později uložit a poslat do STM. Uložení intervalů a jejich následné poslání do STM na screenshotu sice vidět není, nicméně funguje to tak, že po editaci a uložení intervalů se zobrazí tlačítko „save into device“, pomocí kterého můžeme intervaly synchronizovat s STM. U položky intervals je také vidět timestamp.

Dodejme ještě, že takto zobrazených zařízení může na stránce být víc.

4.2.2 Přidání nového zařízení

Přidání nového zařízení je proces, při kterém server vygeneruje nový privátní DES klíč, který uživatel zadá do STM a to dále šifruje zprávy pomocí tohoto klíče.

Na webový stránkách stačí kliknout na tlačítko „register new device“, zadat ID zařízení a dále kliknout na „generate new key“. Server zkontroluje jestli zadané ID je ID existujícího STM a následně vygeneruje DES klíč s omezenou životností.

5. Vyhodnocení

V následující kapitole rozebereme způsoby, jakým se dá testovat funkcionalita celého systému. Testování funkcionality serveru není problém, takže se zaměříme pouze na testování STM.

Firmware pro STM se poměrně těžko testuje. Hlavně kvůli tomu, že na STM není žádný operační systém, který by spouštěl automatické testy třeba jako samostatné procesy a potom nám vrátil vyhodnocení.

Části firmwaru, které nejsou závislé na hardwaru, je vhodné testovat na PC. Takovou částí je například celá síťová komunikace - tedy HTTP klient.

5.1 Hardwarově nezávislé komponenty

5.1.1 Device simulator

Device simulator je velice jednoduchý program, který simuluje síťovou část STM. Device simulator původně vznikl jako záměr vytvořit jádro síťové komunikace - tedy HTTP klienta, parser, a vše co s tím souvisí a to celé poskládat do konzolového programu, který by na vstupu dostával příkazy typu „připoj se k serveru“ nebo „změň teplotu“. Vybudovat jednoduchý mechanismus zpracování příkazů ze vstupu je snadné, nejtěžší část je síťová komunikace. Hlavní „produkt“ device simulatoru mělo být jádro síťové komunikace, které bychom mohli přímo použít na STM. Mechanismus zpracování příkazů je vedlejším „produktem“, nicméně poměrně užitečným - dovolí nám to totiž testovat připojení více STM k serveru najednou. Device simulator umí v podstatě pouze provádět *komunikační cyklus* se serverem.

Výhoda vývoje síťové komunikace v rámci device simulatoru je jasná - můžeme komunikaci testovat na jednom PC. Navíc je zde i lepší možnost ladění programu, protože po každé nemusíme program nejdřív nahrávat do flash paměti STM.

Device simulator je součástí přílohy této práce.

5.2 Hardwarově závislé komponenty

Zde se jedná zejména o *Application drivers*, do kterých patří třídy, které s hardware pracují buď přímo, nebo přes HAL. Pro tyto komponenty můžeme vytvořit něco jako unit testy, které přilinkujeme do zbytku firmwaru a explicitně je zavoláme z funkce `main`. Je to sice nepraktické, ale v některých případech se to může hodit. Například tímto způsobem testujeme `OneWire`.

Takových „unit testů“ ale není mnoho. Jednak kvůli jejich nepraktičnosti, ale zejména kvůli jejich nadbytečnosti. Tyto „unit testy“ by totiž musely testovat zejména *GUI* část firmwaru, která by se tímto způsobem testovala opravdu těžko.

5.3 Testování celkové funkcionality

Pro otestování celkové funkcionality, tedy STM, serveru a komunikace mezi nimi, je potřeba připravit prostředí, které se bude podobat reálnému použití. Ser-

ver tedy necháme běžet na PC s veřejnou, statickou IP adresou. STM připojíme do routeru, který je připojen k internetu. Ovšem nemusí to být vyloženě router, stačí když STM připojíme do jiného PC, na kterém poběží DHCP server, a bude přeposílat pakety z STM. V STM je na pevno zakódovaná URL adresa serveru a navíc je na něm implementováno DNS.

Po tomto zapojení už jen stačí manuálně vyzkoušet funkcionalitu serveru i STM.

Závěr

Cíle práce se dají rozdělit do tří částí: embedded část práce, web-server část práce a nakonec komunikace mezi webový serverem a embedded zařízením. Následuje shrnutí jakým způsobem jsme dosáhli cíle v rámci jednotlivých částí.

Embedded část práce V rámci této části práce jsme měli za cíl vytvořit firmware umožňující uživateli nastavovat různé teploty po různé časové úseky během dne a zároveň zobrazovat aktuálně naměřenou teplotu. Jako embedded zařízení jsme zvolili STM3210C-Eval board a firmware vytvořili pomocí knihoven od STM tak, aby byl přenositelný na de facto libovolné STM32 zařízení. Vzhledem k náročnosti programování v embedded prostředí jsme zvolili strategii udržet firmware co nejjednodušší. Specifikovali jsme několik různých obrazovek, které může STM3210C-Eval board zobrazovat na svém displeji a díky kterým může uživatel nastavit jednotlivé intervaly, vidět aktuálně naměřenou teplotu a připojit zařízení k serveru.

Část práce zabývající se webovým serverem Vzhledem k povaze embedded zařízení a požadavku na co nejjednodušší firmware, jsme se rozhodli implementovat webový server jako separátní entitu. Práci jsme si ulehčili použitím webového backendu Django. Přestože námi implementovaný webový server nemá příliš rozsáhlou funkcionalitu, mohl by do budoucna být rozšířen o spoustu dalších vlastností.

Komunikace mezi serverem a embedded zařízením V rámci této části práce jsme rozebrali několik možností, jak by šla komunikace mezi webový serverem a embedded zařízením realizovat. Nakonec jsme se rozhodli pro implementaci komunikace pouze pomocí HTTP. Specifikace komunikace by mohla být snadno rozšířena o několik dalších položek, aniž by se měnil její princip. Dále jsme rozebrali možnosti jak komunikaci zabezpečit a to jak ve smyslu autentizace STM na serveru, tak ve smyslu „skrytí“ obsahu komunikace pomocí šifrování. Nakonec jsme ještě rozebrali možnosti synchronizace intervalů mezi serverem a STM.

Shrnutí Přestože v rámci této práce podporujeme „pouze“ STM32 zařízení, které serveru posílá pouze naměřenou teplotu a vyměňuje si s ním nastavení časových intervalů, do specifikace komunikace by se poměrně snadno daly zařadit i další položky jako je například vlhkost vzduchu. Dodejme, že pro podporu jiných embedded zařízení by bylo potřeba přeprogramovat hardwarově specifickou část firmwaru, nicméně na serveru by přidání podpory pro nový typ zařízení nebylo složité.

Seznam použité literatury

[1] ST. STM3210C-Eval board. URL https://www.st.com/en/evaluation-tools/stm3210c-eval.html. Přístup: 2018-07-09.

[2] Dallas Semiconductor. DS18B20 datasheet. URL https://cdn.sparkfun.com/datasheets/Sensors/Temp/DS18B20.pdf. Přístup: 2018-07-09.

[3] ST. STM32Cube MCU Package for STM32 F1 series. URL https://www.st.com/en/embedded-software/stm32cubef1.html. Přístup: 2018-07-09.

[4] Oficiální stránky projektu FreeRTOS. URL https://www.freertos.org/. Přístup: 2018-07-09.

[5] ST. STemWin. URL https://www.st.com/en/embedded-software/stemwin.html. Přístup: 2018-07-09.

[6] Oficiální stránky projektu LwIP. URL https://savannah.nongnu.org/projects/lwip/. Přístup: 2018-07-09.

[7] Mark Dermot Ryan. Symmetric-key cryptography. URL https://www.cs.bham.ac.uk/~mdr/teaching/modules/security/lectures/symmetric-key.html. Přístup: 2018-07-09.

[8] ST. Reference manual for STMF1xx. URL https://www.st.com/content/ccc/resource/technical/document/reference_manual/59/b9/ba/7f/11/af/43/d5/CD00171190.pdf/files/CD00171190.pdf/jcr:content/translations/en.CD00171190.pdf. Přístup: 2018-07-09.

[9] Unix timestamp. URL https://www.unixtimestamp.com/index.php. Přístup: 2018-07-09.

[10] Internet Engineering Task Force (IETF). HTTP/1.1 persistent connections. URL https://www.w3.org/Protocols/rfc2616/rfc2616-sec8.html. Přístup: 2018-07-09.

[11] Oficiální stránky projektu Django. URL https://www.djangoproject.com/. Přístup: 2018-07-09.

[12] Oficiální stránky projektu Bootstrap verze 3.3. URL http://getbootstrap.com/docs/3.3/. Přístup: 2018-07-09.

[13] Oficiální stránky projektu jQuery. URL https://jquery.com/. Přístup: 2018-07-09.

[14] Internet Engineering Task Force (IETF). Network time protocol verze 4. URL https://tools.ietf.org/html/rfc5905. Přístup: 2018-07-09.

[15] Internet Engineering Task Force (IETF). Simple Network Time Protocol verze 4. URL https://tools.ietf.org/html/rfc4330. Přístup: 2018-07-09.

[16] Network Working Group. Time Protocol. URL https://tools.ietf.org/html/rfc868. Přístup: 2018-07-09.

[17] Oryx Embedded. CycloneSSL. URL https://www.oryx-embedded.com/cyclone_ssl.html. Přístup: 2018-07-09.

[18] Oryx Embedded. CycloneTCP. URL https://www.oryx-embedded.com/cyclone_tcp.html. Přístup: 2018-07-09.

[19] ST. STM32F107 In-Application Programming (IAP) over Ethernet. URL https://www.st.com/content/ccc/resource/technical/document/application_note/95/96/37/0e/37/1a/4f/23/CD00275365.pdf/files/CD00275365.pdf/jcr:content/translations/en.CD00275365.pdf. Přístup: 2018-07-09.

[20] ST. STM32F4-Discovery board. URL https://www.st.com/en/evaluation-tools/stm32f4discovery.html. Přístup: 2018-07-09.

Seznam obrázků

Seznam použitých zkratek

TCP/IP - Transmission Control Protocol/Internet Protocol
HTTP - Hypertext Transfer Protocol
HTTPS - Hypertext Transfer Protocol Secure
TLS - Transport Layer Security
HTML - Hypertext Markup Language
DNS - Domain Name System
DHCP - Dynamic Host Configuration Protocol
URL - Uniform Resource Locator
JSON - Javascript Object Notation

A. Přílohy

V přiloženém archivu jsou dvě složky stm_device a stm_server, kde každá z nich obsahuje readme soubor, který popisuje základní informace typu: softwarové závislosti, jak kód přeložit, apod. stm_device obsahuje všechny zdrojové kódy a všechny knihovny potřebné pro přeložení firmwaru pro STM. stm_server obsahuje zdrojové kódy a jednoduchý obsah databáze pro webový server.

Vzhledem k nedokončené práci na *Device simulator* není obsažen v příloze.