UNIVERZITA PARDUBICE

Fakulta elektrotechniky a informatiky

SOFTWARE PRO AUTOMATIZOVANÉ TESTOVÁNÍ FIRMWARE PRVKŮ PRŮMYSLOVÉ AUTOMATIZACE

Bc. Martin Poledno

Diplomová práce

2024

Univerzita Pardubice
Fakulta elektrotechniky a informatiky
Akademický rok: 2023/2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Bc. Martin Poledno**
Osobní číslo:	**I22191**
Studijní program:	**N0714A150005 Automatické řízení**
Téma práce:	**Software pro automatizované testování firmware prvků průmyslové automatizace**
Zadávající katedra:	**Katedra řízení procesů**

Zásady pro vypracování

Cílem práce je implementace a ověření software pro automatizované testování firmware (FW) zařízení. Výsledný software bude umožňovat provedení testu FW podle předem vytvořených scénářů a bude realizován s ohledem na modularitu a případné snadné budoucí rozšíření.

Teoretická část: Stručná historie a rešerše aktuálního stavu poznání v oblasti testování FW a HW. Obecný popis automatizovaného testování a teoretický popis diplomantem navrženého řešení. Dále popis a zhodnocení přínosů automatických testů – zkrácení času pro vývoj zařízení, definovaná opakovatelnost testování, automatizované generování testovacích reportů apod.

Implementační část: SW implementace a podrobný popis funkce aplikace pro automatizované testování FW. Praktický popis jednotlivých fází testování. Analýza přínosů realizovaného řešení s uvedením rozdílů časů potřebných pro ruční a automatizované testování FW.

Rozsah pracovní zprávy: **cca 60 stran**
Rozsah grafických prací:
Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

DOMAT CONTROL SYSTEM, spol. s r.o., 2021a. *EPC102 regulátor topení, komunikativní* [katalogový list].
DOMAT CONTROL SYSTEM, spol. s r.o., 2021b. *FCR015: komunikativní regulátor pro VAV systémy* [katalogový list].
SUNHA, Ahn, 2016. *Automated firmware testing using firmware-hardware interaction patterns* [dissertation thesis]. Department of Electrical Engineering, Princeton University.

Vedoucí diplomové práce: **Ing. Libor Kupka, Ph.D.**
Katedra řízení procesů

Datum zadání diplomové práce: **8. listopadu 2023**
Termín odevzdání diplomové práce: **17. května 2024**

L.S.

Ing. Zdeněk Němec, Ph.D. v.r.
děkan

Ing. Daniel Honc, Ph.D. v.r.
vedoucí katedry

V Pardubicích dne 14. listopadu 2023

Prohlášení

Prohlašuji:

Práci s názvem Software pro automatizované testování firmware prvků průmyslové automatizace jsem vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Beru na vědomí, že v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů, a směrnicí Univerzity Pardubice č. 7/2019 Pravidla pro odevzdávání, zveřejňování a formální úpravu závěrečných prací, ve znění pozdějších dodatků, bude práce zveřejněna prostřednictvím Digitální knihovny Univerzity Pardubice.

V Pardubicích dne 02. 04. 2024

Bc. Martin Poledno v.r.

PODĚKOVÁNÍ

Rád bych poděkoval svému vedoucímu diplomové práce, panu Ing. Liboru Kupkovi, Ph.D. a konzultantovi, Ing. Petru Vaníčkovi za odborné rady a vedení mé diplomové práce. Také bych rád poděkoval rodině za podporu, kterou mi poskytovala po celou dobu mého studia.

V Pardubicích dne 02.04.2024

Bc. Martin Poledno

ANOTACE

Diplomová práce se zabývá návrhem, realizací a programováním automatizovaného testeru pro FW prvků průmyslové automatizace. Teoretická část práce popisuje různé modely vývoje software a firmware, upozorňuje na úskalí nedostatečného testování při vývoji software a firmware s poukázáním na nejzásadnější chyby s globálním dopadem. Praktická část představuje vývoj automatizovaného testeru a související obslužné aplikace. Podrobně popisuje použité komponenty, rozhraní pro vývoj aplikace a stanovuje scénáře příslušných testů vč. vývojových diagramů.

KLÍČOVÁ SLOVA

firmware, automatizovaný test, regulátor, průmyslová automatizace, řízení budov

TITLE

Software for automated testing of firmware components in industrial automation

ANNOTATION

The thesis deals with design, implementation, and programming of an automated tester for industrial automation FW components. The theoretical part describes various software and firmware development models, highlighting pitfalls of insufficient testing during software and firmware development, with emphasis on the most critical errors with global impact. The practical part presents the development of an automated tester and related service applications. It provides detailed description of the components used, interfaces for application development, and establishes scenarios for relevant tests including development diagrams.

KEYWORDS

firmware, automated test, controller, industrial automation, building management

OBSAH

SEZNAM ZKRATEK A ZNAČEK

COM	komunikační port (Communication)
FTP	protokol sdílení souborů (File Transfer Protocol)
FW	firmware
MaR	měření a regulace
SW	software
UI	uživatelské rozhraní (User Interface)
VPN	virtuální soukromá síť (Virtual Private Network)
ROM	paměť pouze pro čtení (Read-Only Memory)
RTU	jednotka reálného času (Real-Time Unit)

SEZNAM ILUSTRACÍ

SEZNAM TABULEK

ÚVOD

V dnešní době energetické krize, kdy ceny energií neustále rostou, nabývají systémy měření a regulace (dále jen MaR) čím dál většího významu. Na systémy MaR je tedy nezbytné myslet už při projektování nových budov, či při jejich rekonstrukcích. MaR se stávají nedílnou součástí nových staveb a představují tak účinný prostředek ke snižování energetických nákladů a optimalizaci provozních procesů.

V tomto kontextu dochází ke zvyšování poptávky po těchto systémech a k jejich neustálému vývoji. Zařízení používaná k automatizaci budov se stále více zdokonalují a adaptují na požadavky moderní doby.

Jedním z klíčových prvků vývoje těchto zařízení je jejich testování, které má za cíl ověřit jejich spolehlivost, efektivitu a schopnost plnit požadované úkoly. Testovat zařízení lze z více perspektiv. Jedním pohledem jsou zátěžové testy, které zařízení dostávají na hranici definovaných elektrických parametrů (např. spínání připojené zátěže na limitu spínacího prvku), jiným pohledem jsou logické testy, které kontrolují správnost a robustnost algoritmu.

Tato diplomová práce se zabývá funkční logikou zařízení. Hlavním cílem je navrhnout a zrealizovat automatizovaný tester pro prvky průmyslové automatizace (konkrétně pokojové regulátory), spolu s příslušnou obslužnou aplikací, která umožní provádět testy firmware s minimálními nároky na lidskou interakci. Při testování firmwaru je kladen důraz na spolehlivost, efektivitu a opakovatelnost prováděných testů.

Práce je rozdělena do dvou částí. V teoretické části jsou popsány modely vývoje SW a základní metodiky testování. Praktická část popisuje obslužnou aplikaci a ukázku testů, které jsou pro zařízení naprogramovány.

1 TEORETICKÁ ČÁST

Pro správné vysvětlení problematiky testování firmware (FW) je nejdříve nutné definovat a popsat jeho samotný vývoj, testovací metodiky a použitou sběrnici Modbus RTU, skrze kterou jsou automatizovaný tester a testovaná zařízení ovládány. Tato témata jsou zpracována v teoretické části diplomové práce.

1.1 ROZDÍL MEZI SOFTWARE A FIRMWARE

Jelikož se tato diplomová práce zabývá návrhem a realizací SW pro testování FW prvků automatizace, primárně regulátorů, je nutné definovat rozdíl mezi software a firmware. Oba pojmy se mezi sebou výrazně překrývají, ale i tak mají určité rozdíly. FW je druh SW nízké úrovně, který je často úzce spojen s vlastním HW. Zatímco za SW se obecně považuje program vyšší úrovně běžící v rámci operačního systému.

FW je nízko úrovňový druh SW který má za úkol obsluhu připojených periferií k procesoru a případně běh programu bez operačního systému. Pro vykonávání dané posloupnosti úkonů využívá FW instrukční sady použitého procesoru, která definuje mechanismy pro vykonání FW. Každý HW vyžaduje nějakou formu FW, aby fungoval. Např. tiskárna, aby správně fungovala, musí mít v sobě uložen nějaký program, tomuto programu se říká FW. Nejznámějším příkladem FW je BIOS. Ten se nachází na základní desce počítače, konkrétně je uložen v ROM základní desky. Obvykle není přístupný přímo z operačního systému a přímý přístup je možný ještě před spuštěním operačního systému. BIOS je jeden z mála typů FW, který také disponuje zjednodušeným uživatelským rozhraním. Pomocí tohoto rozhraní může uživatel provádět úpravy, které se přímo propíšou do chování HW.

SW je jakýkoliv program pracující na PC, stejně tak jako operační systém. FW je vlastně jen jiný typ SW, který má specifičtější roli. Je nutné tedy rozlišovat mezi FW a SW? Ve většině situací není důvod, ale aby bylo jasně definováno, čím se tato diplomová práce vlastně zabývá, byl vysvětlen rozdíl. Program, který obsluhuje přímo konkrétní zařízení, bez činnosti operačního systému se nazývá FW. Tento program byl v praktické části testován (Summerson, 2022).

1.2 ZNÁMÉ CHYBY V SOFTWARE

Jelikož SW programují lidé, žádný SW není dokonalý, proto je důležité ho vždy co nejlépe otestovat. V následujících pododdílech, jsou uvedeny známé chyby, které způsobily nemalé finanční škody.

1.2.1 Chyba v dělení s pohyblivou řádovou čárkou u procesoru Intel Pentium

Po zadání výpočtu,

$$\frac{4\,195\,835}{3\,145\,727} * 3\,145\,727 - 4\,195\,835,$$

do kalkulačky počítače by měla být výsledkem samozřejmě nula. U procesorů Intel Pentium roku 1994 tomu ale tak nebylo. Jednalo se o SW chybu, která byla vypálena do mnoha počítačových čipů. Chyba byla odhalena Dr. Thomasem R. Nicelym dne 30. října 1994. Odhalenou chybu zveřejnil na internetu, kde došlo k následnému potvrzení od ostatních uživatelů tohoto procesoru. Záhy byly odhaleny i jiné situace, které vedly k nesprávným výsledkům. Naštěstí tyto chyby byly poměrně vzácné, a běžný uživatel za celou dobu používání počítače s daným procesorem nemusel na zmíněný problém narazit. Chybné výsledky se objevily pouze vzácně u vědeckých výpočtů, které byly extrémně náročné na matematické operace. Nejzajímavějším faktem je to, jak se k dané chybě zachovala firma Intel (Janeba, 1995).

Vývojáři na zmíněnou chybu přišli již v době, kdy prováděli testy procesoru, tj. ještě před uvedením procesoru na trh. Vedení firmy došlo k závěru, že zjištěná chyba není natolik závažná, aby vyžadovala opravu. Ve chvíli, kdy byla chyba veřejností zjištěna, se společnost Intel snažila snížit přikládanou závažnost této chyby. Intel sice nabídl výměnu vadných čipů, ale jen uživatelům, kteří byli chybou v procesoru finančně poškozeni. Aby Intel vyvrátil tvrzení článků, ve kterých se psalo, že je společnost nedůvěryhodná a nezodpovědná, byl pod velkým tlakem nucen uvolnit přes 400 miliónů dolarů na náhrady vadných čipů. Z tohoto příkladu je evidentní, jak je testování SW důležité a jak se společnost Intel zachovala nesprávně při vývoji procesoru (Janeba, 1995).

1.2.2 Přistávací modul na Mars, NASA, 1999

Během přistávání modulu sondy NASA letící na Mars 3. prosince 1999 došlo k nehodě, kdy se modul zcela ztratil. Bylo zjištěno, že nejpravděpodobnější příčinou bylo nastavení jediného bitu, na jehož základě se vypínal přísun paliva k brzdným motorům. Pro NASA bylo velmi znepokojivé, proč se na tuto závadu nepřišlo při interních testech. Teorie přistání byla následující: během volného pádu modulu atmosférou planety se měl rozvinout padák na přibrzdění modulu. Ihned po otevření padáku se měly vysunout tři nohy, které se měly nastavit do pozice připravené pro přistáni. K zažehnutí pomocných přistávacích motorů, díky kterým měl být zpomalen další pokles směrem k povrchu, a k vypuštění padáku sondy mělo dojít ve výšce 1800 metrů nad samotným povrchem Marsu (Patton, 2002).

NASA chtěla uspořit finance, a tak zjednodušila mechanismus, který měl za úkol rozhodnout o správném okamžiku vypnutí motorů. Na spodní plochu nohy sondy byl umístěn obyčejný kontakt, který nastavil v počítači určitý bit a nařídil příkaz k vypnutí přívodu paliva. Tzn. že se motory měly vypnout v tu chvíli, kdy se nohy dotknou povrchu (Patton, 2002).

Výborem pro vyšetřování chyb bylo zjištěno, že jakmile byly nohy modulu vysunuty, vibrace při přistávání falešně sepnuly kontakt pro vypnutí motorů, čímž byl nastaven i onen bit. S největší pravděpodobností počítač rozhodl o vypnutí motorů, protože si myslel, že je již na povrchu planety (Patton, 2002).

Poměrně zásadní katastrofu způsobila velmi jednoduchá příčina. Na testování přistávacího modulu se podílelo několik týmů. Každý tým, ale řešil problematiku odděleně bez přesahu do jiných týmů. To byla chyba, protože se testovací metody mechanismů nepřekrývaly. Mechanismus vysouvání nohou testoval první tým a proces přistávání od okamžiku vysunutí testoval tým druhý. Jestli byl bit sepnutý, dotykovým kontaktem první tým nezajímalo, protože neměli v popisu práce řešit dotykový kontakt. Naopak druhý tým resetoval počítač vždy, když začínali s testováním. Dokud se tyto dva mechanismy nespojily, chovaly se správně (Patton, 2002).

Na základě tohoto příkladu je patrné, že je nutné vždy při vývoji SW myslet více za hranice svých úkolů.

1.2.3 Problém roku 2000 (Y2K)

Y2K byla chyba v počítačových SW, která mohla způsobit problémy při práci s daty po datu 31. prosince 1999. Této chybě následující den, tj. 1. ledna 2000, čelilo spousta uživatelů a programátorů na celém světě. Mezi 60. až 80. lety minulého století, počítačoví inženýři a

programátoři používali pro zaznamenání roku pouze dvouciferný formát. Např. místo 1985 bylo v počítačových pamětích zaznamenáno pouze číslo 85. Důvodem byly nákladné paměti PC, které bylo nutné co nejvíce šetřit. Jak se rok 2000 blížil, programátoři si začali uvědomovat problém, že počítač nemusí interpretovat dvojčíslí „00“ správně. Běžné činnosti by byly touto špatnou interpretací poškozeny. Např.: datum 1. ledna 2000 by mohly počítače interpretovat jako datum 1. ledna 1900 (Rutledge, 2022).

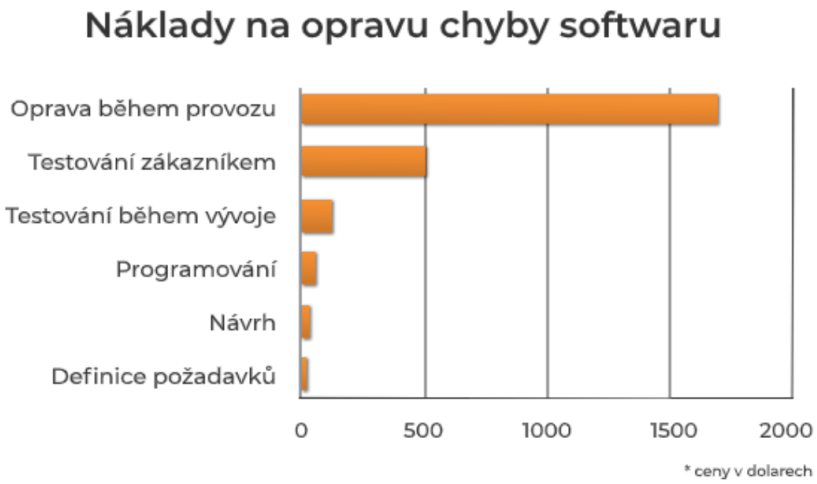
Tento problém mohl např. postihnout řadu bank. Úrokové sazby banky počítají na základě rozdílu času. Z problému Y2K vyplývá, že by místo úroků za jeden den mohly počítače bank spočítat úrok téměř za 100 let (Rutledge, 2022).

Ohrožena mohla být také doprava, neboť její správný chod je závislý na přesném datumu a času. Zvláště počítačové systémy leteckých společností představovaly jisté riziko. Systémy řídící letecký provoz, rezervační systémy a další kritické systémy používaly datum a čas pro řadu důležitých funkcí. Nesprávným rozpoznáním roku 2000 mohlo dojít k problémům v leteckém provozu (Rutledge, 2022).

Naštěstí problémů bylo nakonec jen velmi málo. Např. Itálie se na problém Y2K velmi málo připravovala a neměla o nic větší technologické problémy než země, které na přípravu tohoto problému utratily miliony dolarů (Rutledge, 2022).

1.3 VÝŠE VÝDAJŮ V ZÁVISLOSTI NA ČASU ODHALENÍ CHYBY

Ze zmíněných příkladů, je patrné, že testování je velmi důležitá fáze vývoje SW. I když zařízení, která bude možné testovat naprogramovanou aplikací (viz praktická část) nepracují s tak kritickými systémy jako počítače v uvedených příkladech, je nutné k problematice vývoje FW přistupovat systematicky a důsledně. Když nedochází během vývoje SW (nebo FW) k pravidelnému a systematickému testování, dochází ke zbytečně vysokým nákladům v dalších fázích vývoje daného zařízení. Např. když by pracovníci ve společnosti Intel hned na začátku vývoje procesoru odhalenou chybu v dělení odstranili, ušetřili by miliony dolarů a nepřišli by o dobrou pověst společnosti. SW by měl vznikat metodickým vývojovým postupem. Chyby v SW je možné rozpoznat v každé fázi vývoje, od specifikace požadavků až po dobu, co je SW v produkčním provozu. Vývoj výdajů v závislosti na dobu odhalení chyby je zobrazen na obr. 1.1.



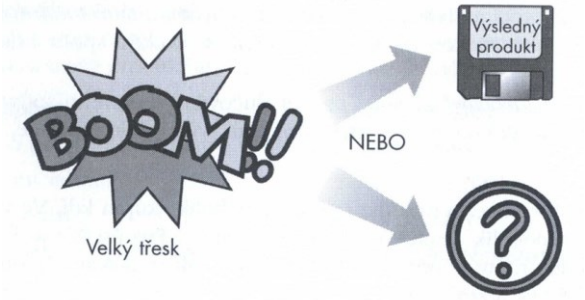
Obr. 1.1 – Náklady na opravu chyby SW (Bohuslav, 2020)

1.4 MODELY VÝVOJE SW

Jak již bylo zmíněno, každý SW by měl vznikat metodickým postupem. Je jasné, že v praxi tomu tak vždy nemusí být. Existují SW, které jsou vyvíjeny chaoticky a naopak SW, které jsou vyvíjeny s přísnou disciplínou. V následujících oddílech, jsou popsány jednotlivé modely vývoje SW.

1.4.1 Model velkého třesku

Model velkého třesku ve vývoji SW je principiálně podobný jako teorie o vzniku vesmíru. Když se nahromadí na jedno místo veliké množství energie (peněz) a hmoty (lidí), vznikne SW produkt. Problémem tohoto modelu je to, že nikdy není dopředu známo, jak daný produkt bude vypadat a jestli vůbec plní to co je požadováno. Také není známa ani alespoň přibližná cena a možná ani termín dodání. Model je velmi jednoduchý, nevyžaduje žádné plánování, ani rozvržení prací. Úsilí je soustředěno pouze na samotné programování SW. Testování SW v tomto modelu vůbec není, což neumožňuje vyprodukovat kvalitní SW produkt (Patton, 2002).



Obr. 1.2 – Model velkého třesku (Patton, 2002)

1.4.2 Model „programuj a opravuj“

Oproti modelu velkého třesku, tento model představuje jisté zlepšení, protože vyžaduje alespoň určité neformální specifikace produktu před začátkem vývoje SW. Vývojový tým začíná hrubým návrhem výsledného produktu, následuje dlouhý nepřetržitý koloběh programování, testování a opravování chyb, do té doby, než projektový manažer vývoj ukončí a rozhodne se uvést produkt na trh. Bohužel plánování ani dokumentace v tomto modelu nehrají významnou roli (Patton, 2002).

Ani u tohoto modelu nedochází k systematickému testování, ale na rozdíl od modelu velkého třesku, alespoň nějakým testováním disponuje (Patton, 2002).

Programátoři a testeři pracují v neustálém cyklu. Testerům jsou každý den předkládány nové, nebo aspoň aktualizované verze SW k překontrolování. Mnohdy testeři nestihnou verzi dotestovat a už dostávají k otestování verzi novou. Zmíněný model je v praxi nejčastější (Patton, 2002).

Obr. 1.3 – Model „programuj a opravuj“ (Patton, 2002)

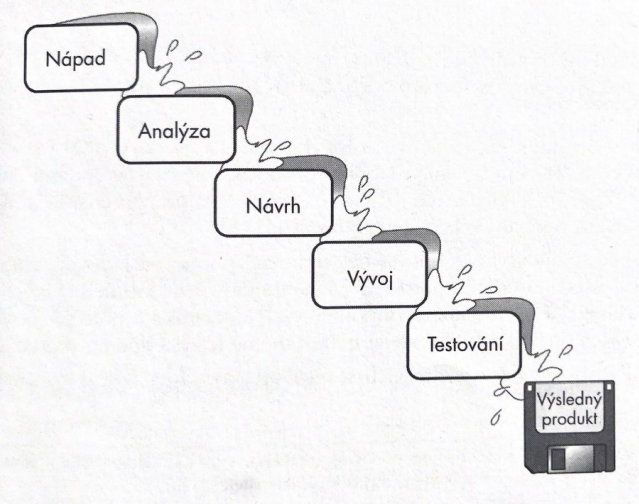
1.4.3 Model vodopád

Jedná se o jednoduchý a elegantní vývojový model vývoje SW. Projekt řízený vodopádovým modelem postupuje po schématu ze shora dolů po jednotlivých stupních. Projekt začíná zrodem nápadu a končí výsledným produktem. Na konci každého stupně vývojový tým zhodnotí, zda je připraven pokračovat. Dokud není projekt dostatečně zralý pro další stupeň, nemůže se posunout dále (Patton, 2002).

Důraz je kladen na pečlivou specifikaci konečné podoby produktu. Zajímavým faktem je to, že vývojový proces, tedy programování je zastoupen jediným blokem. Vývojové stupně projektu jsou od sebe odděleny a není možné se v modelu vracet zpět. V momentě, kdy je projekt v určitém úrovni, je nutné splnit veškeré úkoly. Až následně je možné postoupit ve schématu dále (Patton, 2002).

Na první pohled by se mohlo zdát, že tento přístup je omezující, avšak v případě projektů s jasnou definicí produktu a organizovaným vývojovým týmem přináší dobré výsledky. Model má jednu nespornou výhodu oproti ostatním zmíněným modelům. Veškeré části projektu jsou pečlivě zdokumentovány a specifikovány, což je ohromná výhoda pro testera, který přesně ví, jak má SW pracovat. Tester si tedy může dopředu jednoduše vytvořit rozvrh a plán testů (Patton, 2002).

Nevýhodou tohoto modelu je to, že SW je testován pouze na konci vývojového cyklu. Hrozí tudíž, že se na začátku při specifikacích produktu udělá chyba, která se odhalí až v momentě, kdy má jít finální produkt na trh (Patton, 2002).

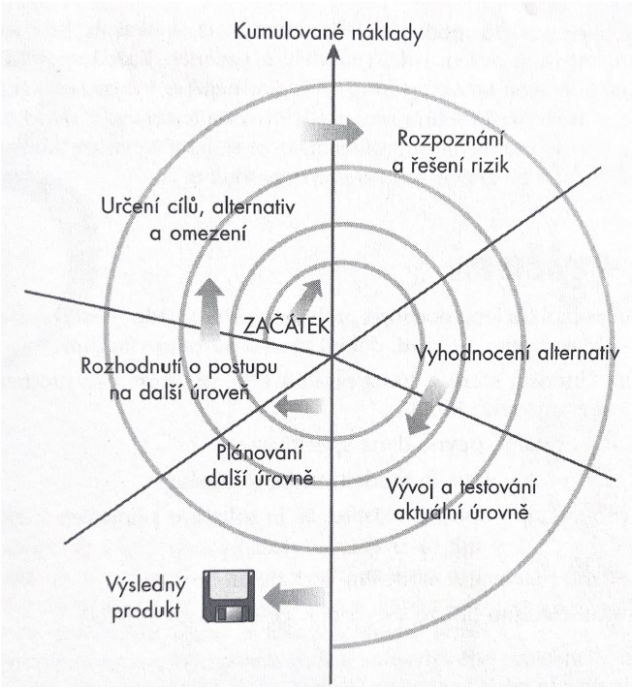


Obr. 1.4 – Model vodopád (Patton, 2002)

1.4.4 Spirálový model

Kombinací všech modelů je model spirálový. Vychází z takového poznatku, že není možné na začátku vývoje definovat celý systém podrobně. Vývojáři začínají s obecnou definicí nejdůležitějších funkcí SW, kterou následně vyzkouší a nechají si schválit od zákazníka. Jakmile je zákazníkem SW schválen, proces se opakuje na další úrovni, kde jsou již specifikace SW podrobnější (Patton, 2002).

Tento model eliminuje výši nákladů na chyby v SW, protože testeři tyto chyby nalézají průběžně a od samotného začátku vývoje SW produktu. Při tomto přístupu může nastat problém s naceněním vývoje, protože dopředu není známo, co všechno bude produkt obsahovat, což je problém, který vede k rozporům mezi dodavatelem a zákazníkem.



Obr. 1.5 – Spirálový model vývoje SW (Patton, 2002)

1.5 SPECIFIKACE TESTOVÁNÍ

V každém ze zmíněných modelů (kromě modelu velkého třesku) existuje určitá specifikace produktu. U pokojových regulátorů, jejímž testováním FW se práce zabývá, se může jednat např. o to, jak má regulátor reagovat v případě, když se na okenním vstupu objeví signál o otevřeném okně, nebo např. jaké maximální teploty může regulátor brát v úvahu.

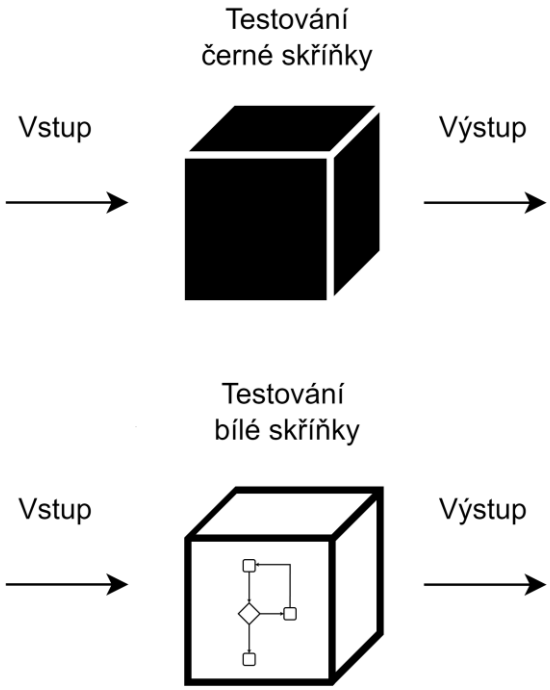
Specifikace produktu musí být napsána dosti podrobně, aby výsledný produkt (FW regulátoru) přesně odpovídal tomu, co zákazník potřebuje. Také to je jedinou možností, jak správně naplánovat potřebné testování. Další nespornou výhodou podrobné specifikace je to, že možné chyby v FW může tester odhalit již před samotným zahájením testování (Patton, 2002).

1.5.1 Testování černé, bílé a šedé skříňky

Testování SW se dělí na dva základní pojmy, testování černé a bílé skříňky, které jsou vyobrazeny na obr. 1.6.

Když tester pracuje se SW systémem černé skříňky, znamená to, že tester neví, jakým způsobem SW došel ke konkrétnímu výstupu. Např. u pokojového regulátoru by tester mohl chtít nastavit na regulátoru výstup na povel „topit“ a zkontrolovat, jestli regulátor pracuje podle očekáváného nastavení. Tester neví, jak logický algoritmus uvnitř regulátoru pracuje, ale ví, co

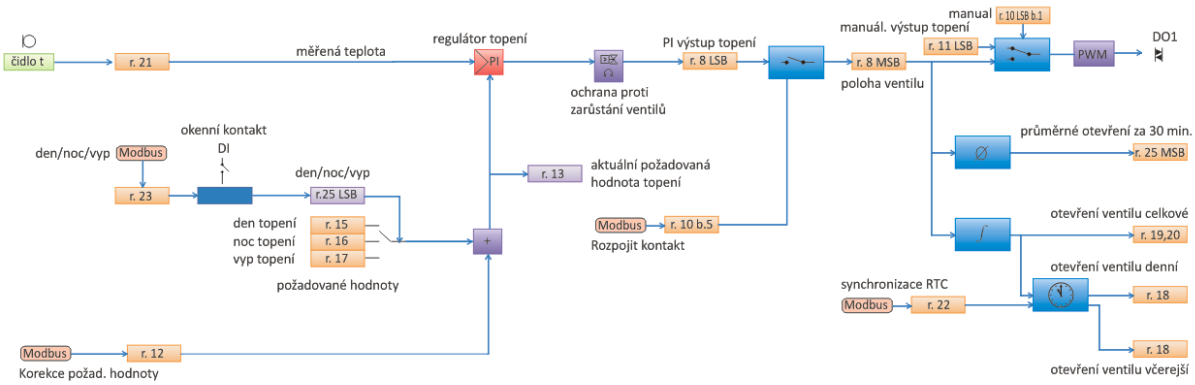
se má stát. V případě, že by tester věděl, jakým způsobem uvnitř regulátor pracuje, resp. měl by přístup k programu (FW), který regulátor obsluhuje, jednalo by se o testování bílé skříňky (Black box vs white box testing, 2024).



Obr. 1.6 – Testování černé a bílé skříňky

Aplikace, která je popsána v praktické části, která umožňuje testy FW automatizovat, používá kombinaci obou zmíněných přístupů. Při přípravě testů sice není možné nahlédnout do obslužného kódu regulátoru, ale ani není funkce definována jen obecným popisem.

K regulátoru náleží diagram funkce (viz obr. 1.7), podle kterého lze testy připravit. Diagram funkce je částí regulačního obvodu. Informace z čidla t představuje měřenou veličinu, nastavení po sběrnici Modbus představuje žádanou hodnotu a výstup regulátoru, přivedený následně na výstup DO1, představuje akční zásah. Na funkční mapě jsou uvedené čísla registrů, která korespondují s Modbusovou mapou, obsaženou v dokumentaci regulátoru.



Obr. 1.7 – Funkční mapa regulátoru EPC102 (Domat Control System s.r.o., 2020)

1.5.2 Statické a dynamické testování

Testování SW lze dále dělit na statické a dynamické. Statické testování je technika, díky které jsou hledány chyby v SW ještě před spuštěním programu. Jedná se o přístup, který kontroluje systémové požadavky, návrh kódu a dokumentaci, která je k SW tvořena. Příkladem statického testování může být např. kontrola syntaxe kódu nebo domluvená konvence psaní kódu v daném vývojovém týmu (Chernyak, 2024).

Dynamické testování SW je naopak funkční kontrola programu, ke které musí být daný program spuštěn. Cílem statického testování je ověřit funkčnost a chování algoritmu. Praktická část se právě tímto testováním zabývá (Chernyak, 2024).

Analogií pro snazší pochopení těchto pojmů může být kontrola stavu automobilu. Když je u auta kontrolována hladina oleje nebo chladící kapaliny před startem motoru, je prováděno statické testování. V případě, že už jsou kontrolovány jízdní vlastnosti, jedná se o dynamické testování.

1.6 PROTOKOL MODBUS

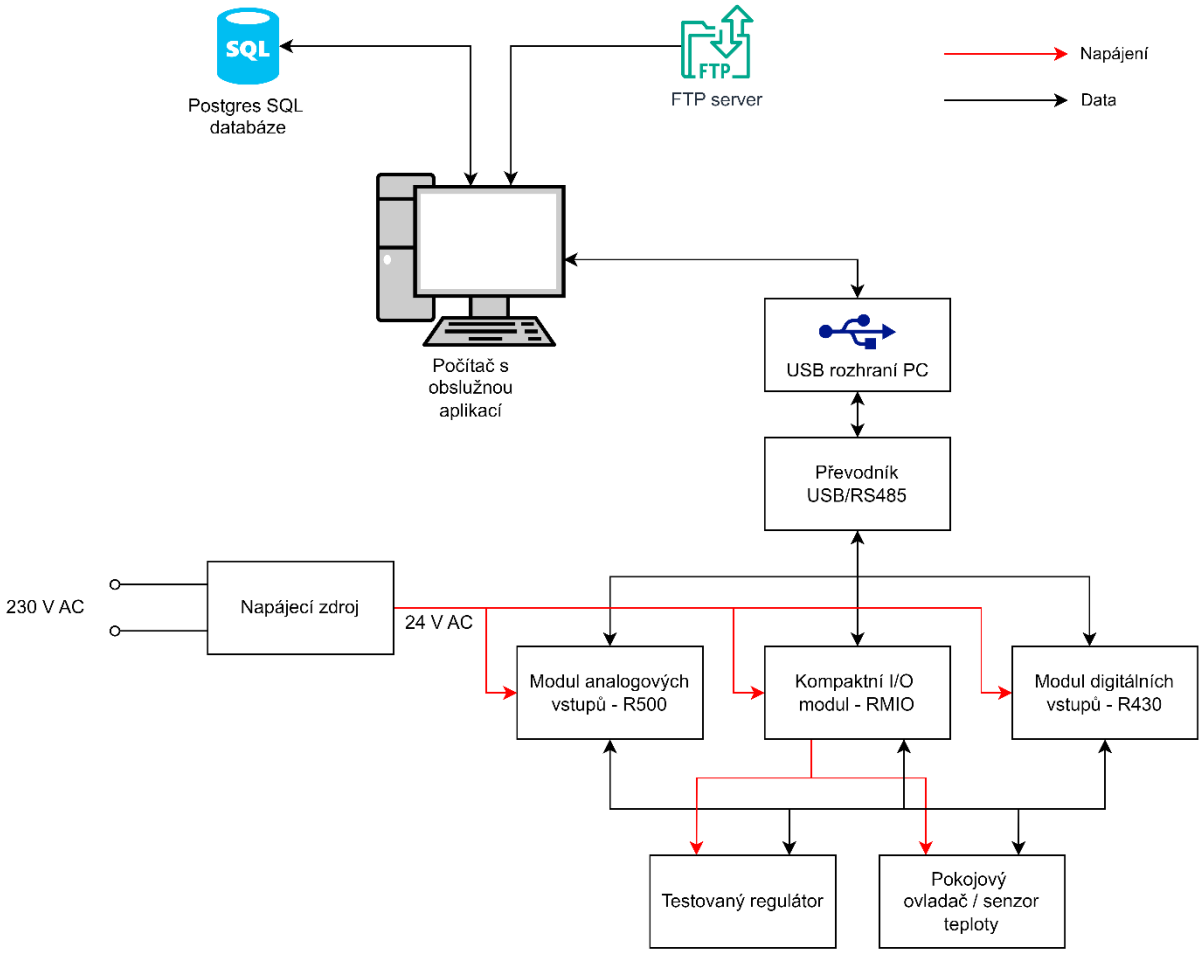
Ikdyž protokol Modbus byl standardizován před více než 40 lety, stále poskytuje spolehlivou a rychlou komunikaci mezi programovatelnými automaty, senzory, termostaty, frekvenčními měniči a dalšími zařízeními. Data mohou být přenášena prostřednictvím rozhraní RS485, RS422, RS232 (Modbus RTU), nebo pomocí TCP/IP protokolu v síti Ethernet (Modbus TCP). Tento otevřený komunikační protokol funguje na základě architektury master/slave neboli klient/server. Protokol má mnoho výhod – je jednoduchý, rychlý, nezávislý na konkrétním výrobci/PLC a snadno se implementuje. Důkazem jeho adaptability je i to, že po zavedení standardu Ethernet a IP protokolu byl Modbus začleněn do relační vrstvy těchto protokolů. Existují dva hlavní režimy přenosu dat: Modbus RTU (použit v praktické části) a Modbus TCP (Komunikační protokol Modbus, 2021).

2 PRAKTICKÁ ČÁST

Praktická část je zaměřena na detailní popis automatizovaného testeru, který obsahuje rozbor jeho konstrukce a obslužné aplikace, sloužící k jeho ovládání. Dále je popsáno, jakým způsobem probíhají vlastní testy jednotlivých zařízení s ohledem na dosažení co největší efektivity, spolehlivosti a opakovatelnosti jednotlivých procedur.

2.1 BLOKOVÉ SCHÉMA AUTOMATIZOVANÉHO TESTERU

Blokové schéma automatizovaného testeru (viz obr. 2.1) popisuje rozdělení systému na jednotlivé bloky. Na schématu je patrné, jak jsou jednotlivé bloky napájeny a mezi kterými bloky se přenášejí data. Zobrazené směry šipek odpovídají skutečnosti, to znamená, že např. z FTP serveru se data pouze čtou. Naopak z databáze jsou stahována data, která uživatele informují o testovaných zařízení, zároveň jsou do databáze ukládány výsledky testů.



Obr. 2.1 – Blokové schéma automatizovaného testeru

2.2 POUŽITÉ KOMPONENTY AUTOMATIZOVANÉHO TESTERU

Pro testování připojených zařízení, primárně regulátorů, jsou použity vstupně-výstupní moduly, které jsou připojeny na komunikační linku s rozhraním RS485. Komunikaci zajišťuje protokol Modbus RTU.

2.2.1 Kompaktní I/O modul – RMIO

Jedná se o komunikativní vstupně-výstupní modul, který je optimalizovaný pro aplikace menšího rozsahu, např. malé kotelny. Tento modul byl zvolen, protože má k dispozici výstupy s elektromechanickými relé a polovodičové výstupy (Domat Control System s.r.o., 2018).

Obr. 2.2 – Kompaktní I/O modul (Domat Control System s.r.o., 2018)

Tab. 2.1 – Technické parametry kompaktního I/O modulu

Napájení	24 V ss/st ± 20%
Spotřeba	7 W
Komunikace	Modbus RTU RS485, 1200…115200 bit/s
Galvanická izolace	1 kV
Max. délka sběrnice	1200 m
Max. počet modulů na sběrnici	256
Analogové vstupy	2× (AI1, AI2): 0…10 V, 0…20 mA ss, 0…1600 Ω, 0…5000 Ω; rozlišení 16 bitů
Analogové výstupy	2× (AO1, AO2): 0…10 V ss
Digitální vstupy	4× (DI1… DI4): 24 V ss/st, vstupní proud 4 mA
Digitální výstupy	5× (DO1…DO5) relé SPST 5 A 2× (DO6…DO7) solid state relé

(Data použita z: Domat Control System s.r.o. 2018)

2.2.2 Modul analogových vstupů – R500

Jelikož testovaná zařízení vyžadují k ověření funkce více než 2 analogové vstupy, které poskytuje I/O modul RMIO, bylo potřeba použít rozšiřující modul analogových vstupů. Komunikativní modul disponuje osmi analogovými vstupy s volitelným rozsahem. Může být použit pro měření napětí, nebo měření proudové smyčky (Domat Control System s.r.o., 2019).

Obr. 2.3 – Modul analogových vstupů (Domat Control System s.r.o., 2019)

Tab. 2.2 – Technické parametry modulu analogových vstupů

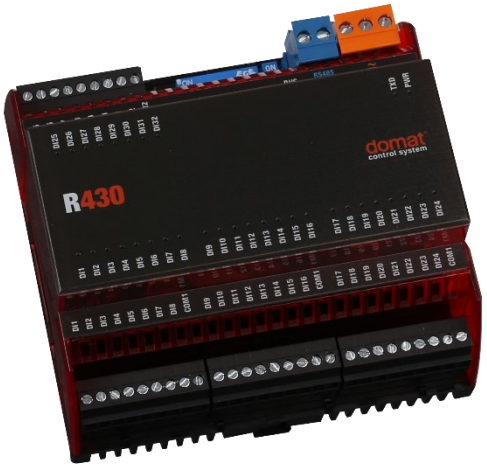
Napájení	24 V ss/st ± 20%
Spotřeba	1,5 W
Komunikace	Modbus RTU RS485, 1200…115200 bit/s
Galvanická izolace	1 kV
Max. délka sběrnice	1200 m
Max. počet modulů na sběrnici	256
Analogové vstupy	8× (AI1…AI8): 0…10 V, 0…20 mA ss; rozlišení 16 bitů

(Data použita z: Domat Control System s.r.o. 2019)

2.2.3 Modul digitálních vstupů – R430

Kompaktní modul RMIO poskytuje 4 digitální vstupy, což pro některé testované regulátory, např. regulátor FCR010, není dostatečné. Z tohoto důvodu byl ještě doplněn modul digitálních vstupů R430. Modul podporuje vstupní napětí do 50 V ss a 30 V st. Mezní hranice pro logickou jedničku a nulu jsou zobrazeny v tab. 2.3. Jelikož svorky GND1 a GND2 nejsou

uvnitř modulu propojeny, umožňuje modul připojení různých potenciálů k jednotlivým skupinám svorek.



Obr. 2.4 – Modul digitálních vstupů (Domat Control System s.r.o., 2017)

Tab. 2.3 – Technické parametry modulu digitálních vstupů

Napájení	24 V ss/st ± 20%
Spotřeba	1 W
Komunikace	Modbus RTU RS485, 1200…115200 bit/s
Galvanická izolace	1 kV
Max. délka sběrnice	1200 m
Max. počet modulů na sběrnici	256
Digitální vstupy	32× (DI1…DI32): (logická nula <5 st/ss, logická jednička >18 st/ss)

(Data použita z: Domat Control System s.r.o. 2017)

2.2.4 Převodník USB–RS485 – R080

Pro připojení automatizovaného testeru k počítači je nezbytné převést sběrnici RS485 na sběrnici USB, což je zajištěno převodníkem, který je popsán v tomto pododdílu.

R080 je servisní převodník, který se primárně používá k adresování a nastavování I/O modulů a pokojových ovladačů. V této diplomové práci je použit pro ovládání automatizovaného testeru. Při připojení převodníku R080 k počítači dojde k vytvoření virtuálního sériového portu COM. Vytvořený port umožňuje libovolnému softwaru komunikovat s připojeným zařízením pomocí převodníku R080 tak, jako by bylo zařízení připojeno přímo k sériovému portu počítače (Domat Control System s.r.o., 2020).

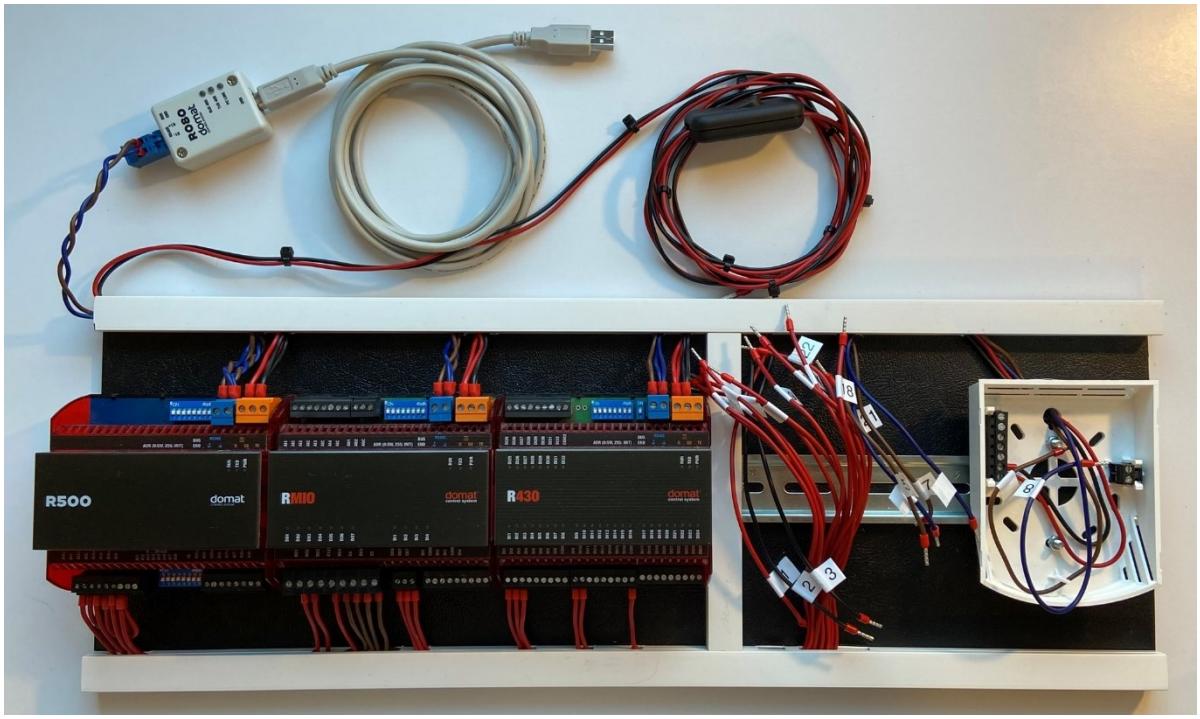
Obr. 2.5 – Převodník USB-RS485 (Domat Control System s.r.o., 2020)

Tab. 2.4 – Technické parametry převodníku USB-RS485

Napájení	Napájeno z USB
Komunikace	Modbus RTU RS485, 300…230400 bit/s
Galvanická izolace	1 kV
Max. počet modulů na sběrnici	256

(Data použita z: Domat Control System s.r.o. 2020)

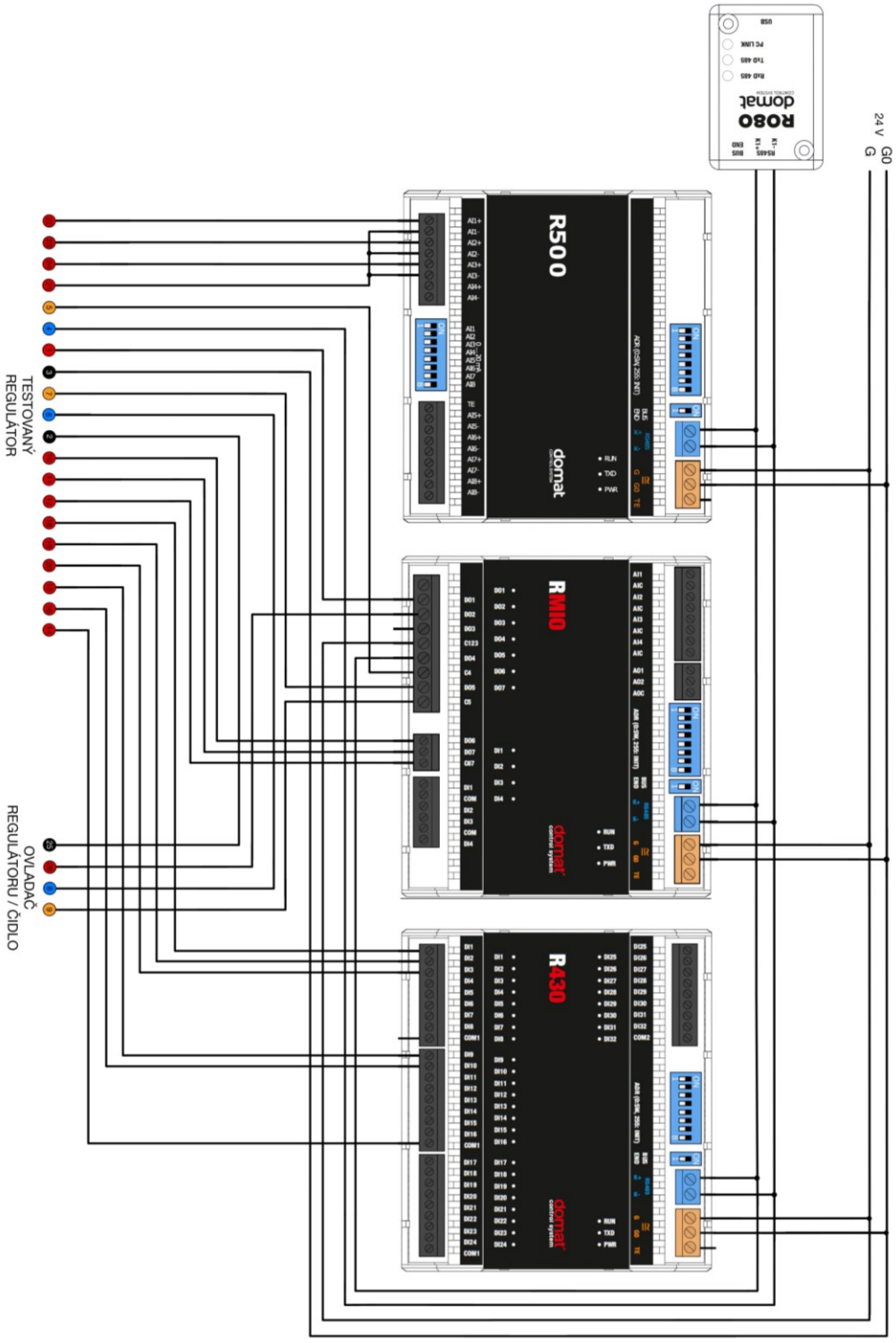
2.3 AUTOMATIZOVANÝ TESTER



Obr. 2.6 – Automatizovaný tester

Automatizovaný tester (viz obr. 2.6) není navrhnutý pouze pro testování jednoho konkrétního regulátoru, ale může být použit pro testování zařízení s různou skladbu I/O a dalších připojených periferií, ať už se jedná pouze o externí čidlo teploty či pokojový ovladač.

Tester je rozdělen na dvě části. V levé části se nachází I/O moduly, které slouží pro vlastní testování vstupů a výstupů testovaného zařízení. V pravé části se nachází prostor pro připojení testovaného regulátoru. Propojovací vodiče jsou pro přehlednost očíslovány, což usnadňuje opakované připojování testovaného zařízení. Operátor testeru, tedy jen připojí regulátor podle schématu, které je zobrazeno aplikací a nemusí vůbec otevírat katalogový list což vede k úspoře



Obr. 2.7 – Automatizovaný tester – schéma

času v přípravné fázi každého testu. To že jsou vodiče značeny podle čísel a ne např. „DO1“, „DO2“, atd. má svůj důvod. Připojení regulátorů se totiž může lišit a tester je takto připraven na otestování různých typů zařízení. Schéma testeru je zobrazeno na obr. 2.7. Ze schématu je patrné, že číslované „bubliny“ jsou barevně odlišeny. Barva odpovídá barvě vodiče, pro snazší orientaci při připojování vodičů.

2.4 TESTOVANÁ ZAŘÍZENÍ

Aby byla problematika testování správně popsána, musí být definována funkce testovaných zařízení (regulátorů). V tomto oddílu jsou popsány funkce dvou regulátorů, na které jsou napsány ukázkové testy zařízení.

2.4.1 EPC102

EPC102 je pokojový regulátor topení s jedním digitálním výstupem a vstupem. Typicky se připojuje na termoelektrické hlavice radiátorů. Regulátor pracuje autonomně, nebo lze pomocí sběrnice Modbus RTU připojit k nadřazenému PLC, které regulátor ovládá. Teplota je měřena pomocí externího čidla teploty po sběrnici 1-Wire. Korekce požadované teploty či provozního módu lze u tohoto typu regulátoru provést pouze přes nadřazený systém a není možná lokálně za pomocí ovládacích prvků které toto zařízení neobsahuje. Zadané a změřené hodnoty zpracovává regulační algoritmus PI. Jelikož výstup regulátoru je dvoustavový, na výstupu je modulační člen, který převádí PI výstup regulátoru na PWM sekvenci s periodou 20 s. Kvazi-spojitý signál na výstupu regulátoru není pro připojené termoelektrické hlavice problém, jelikož mají z principu funkce velkou setrvačnost a doba přestavení z jedné krajní polohy do druhé je cca 3 minuty v závislosti na konkrétním modelu. Výstup může pracovat ve dvou režimech, buď se nastaví na kvazi-spojitý (zmíněná PWM sekvence), nebo na dvoustavový (ON/OFF). Regulátor může pracovat ve třech režimech: den, noc a vypnuto. Režimy se liší přednastavenou žádanou teplotou v místnosti, resp. žádaným pásmem kde uživatel chce teplotu udržovat. Režimy lze přepínat pomocí sběrnice, tj. z nadřazeného PLC. Digitální vstup slouží pro připojení okenního kontaktu. Regulátor má tedy informaci o tom, jestli je v místnosti otevřené okno, a na základě této informace přepíná regulátor do režimu vypnuto (Domat Control System s.r.o., 2017).

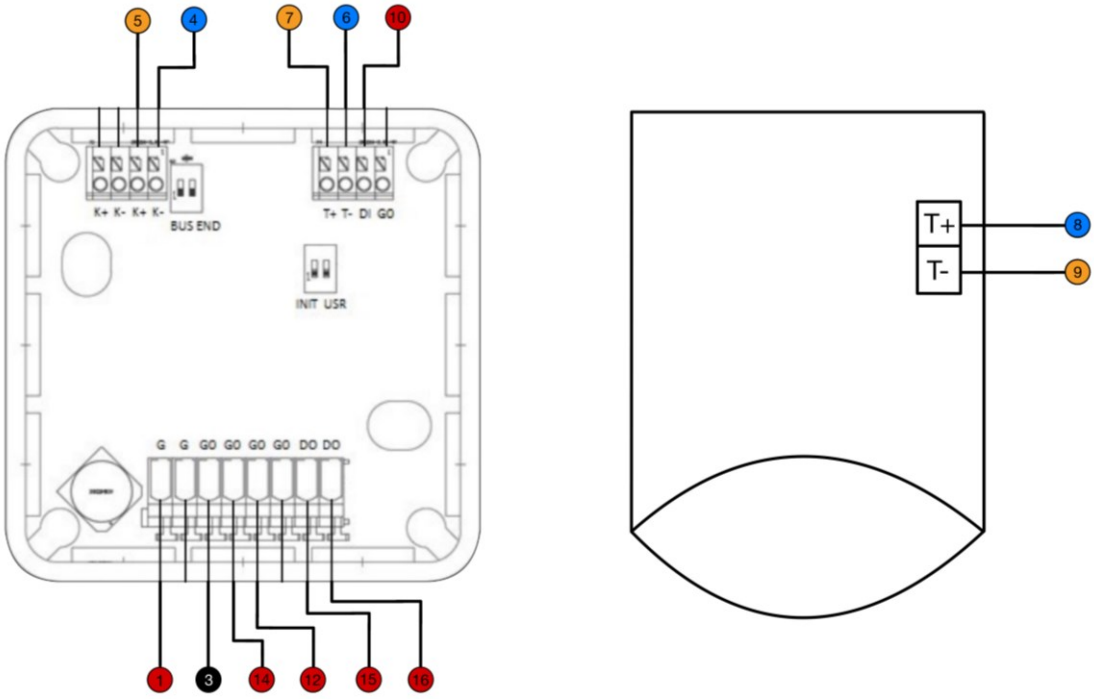
Obr. 2.8 – Regulátor EPC102

Tab. 2.5 – Technické parametry regulátoru EPC102

Napájení	24 V st ± 10%
Spotřeba	6 VA (5 VA pro připojené periferie)
Komunikace	Modbus RTU RS485, 300…115 200 bit/s
Galvanická izolace	1 kV
Max. počet modulů na sběrnici	256
Digitální vstup	1 × DI bezpotenciálový kontakt, 24 V, 5 mA, volitelná logika
Digitální výstup	1 × solid state relé se spínáním v nule pro střídavou zátěž, maximální spínaný proud 1 A.
Max. vzdálenost vedení k čidlu	30 m
Rozsah měření	10…45 °C

(Data použita z: Domat Control System s.r.o. 2017)

Na obr. 2.9 je zobrazeno připojovací schéma regulátoru EPC102, které se zobrazuje automaticky v obslužné aplikaci po vybrání testovaného zařízení. Na obr. 2.10 je zobrazen připojený regulátor k automatizovanému testeru.



Obr. 2.9 – Připojovací schéma regulátoru EPC102

Obr. 2.10 – Připojený regulátor EPC102 k automatizovanému testeru

2.4.2 FCR010

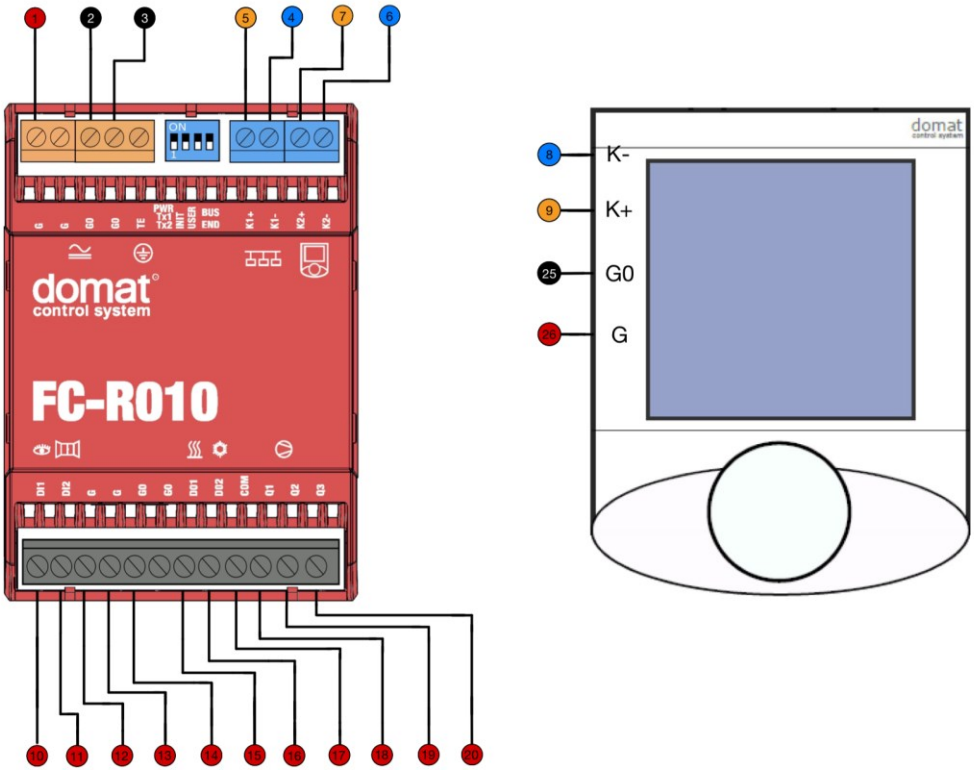
FCR010 je zařízení pro regulaci pokojových fancoilů obsahující řízení otáček ventilátoru za pomocí přepínání jednotlivých vinutí elektromotoru. Umožňuje regulovat ventil pro topení i chlazení. Pracuje buď autonomně nebo dle pokynů z nadřazeného PLC systému. Pro nastavení teploty nebo stupně ventilátoru slouží pokojový ovladač. Snímaná a nastavená teplota pokojovým ovladačem je zpracovávána v regulačním algoritmu PI. Výstupem tohoto algoritmu je PWM sekvence pro triaky, které ovládají ventily fancoilu pro topení nebo pro chlazení. Regulátor obsahuje dva digitální vstupy. Jeden je připojený na okenní kontakt a druhý je napojen na čidlo přítomnosti. Jelikož se regulátor často využívá v hotelových pokojích, vstup přítomnosti může být napojen na signál od přístupové karty k pokoji. Stupně ventilátoru lze měnit buď automaticky podle regulační odchylky z PI regulátoru nebo ručně pokojovým ovladačem (Domat Control System s.r.o., 2017).

Tab. 2.6 – Technické parametry regulátoru FCR010

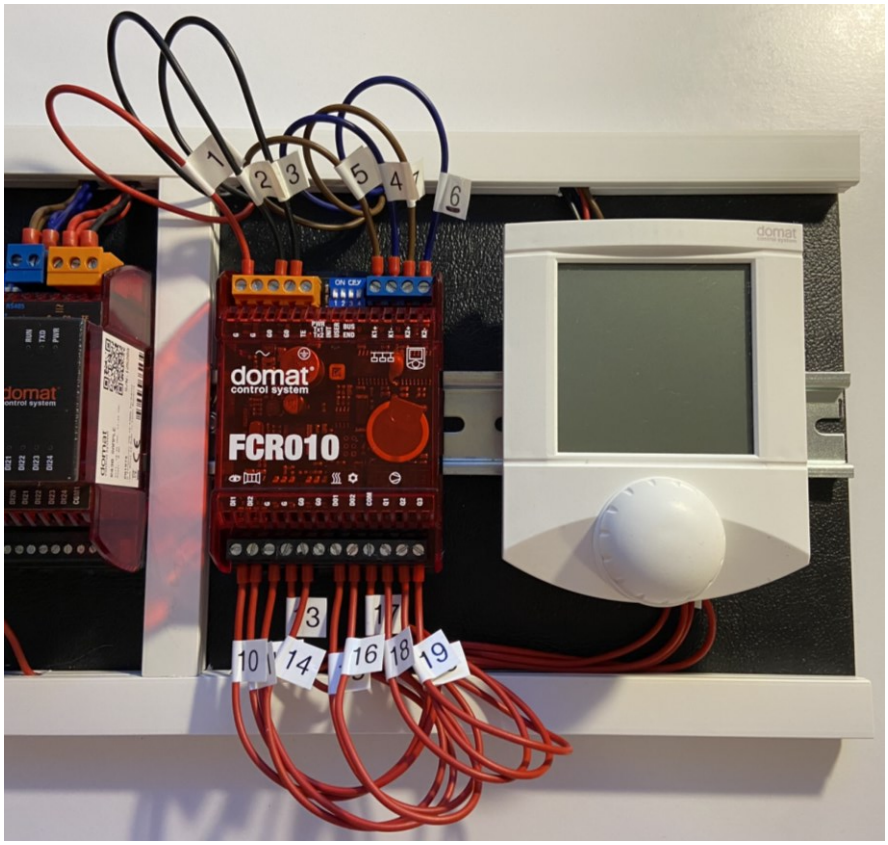
Napájení	24 V st ± 10%
Spotřeba	max 1,8 W
Komunikace	Modbus RTU RS485, 300…115 200 bit/s
Galvanická izolace	1 kV
Max. počet modulů na sběrnici	256
Digitální vstupy	2 × DI bezpotenciálový kontakt, 24 V, 5 mA, volitelná logika
Digitální výstup	2 × solid state relé se spínáním v nule pro střídavou zátěž, maximální spínaný proud 0,4A 3 × relé 230 V st, 5 A
Rozsah měření teloty	0…50 °C

(Data použita z: Domat Control System s.r.o. 2017)

Na obr. 2.11 a 2.12 jsou zobrazeny obdobná schémata, která jsou již popsána v předchozím pododdíle.



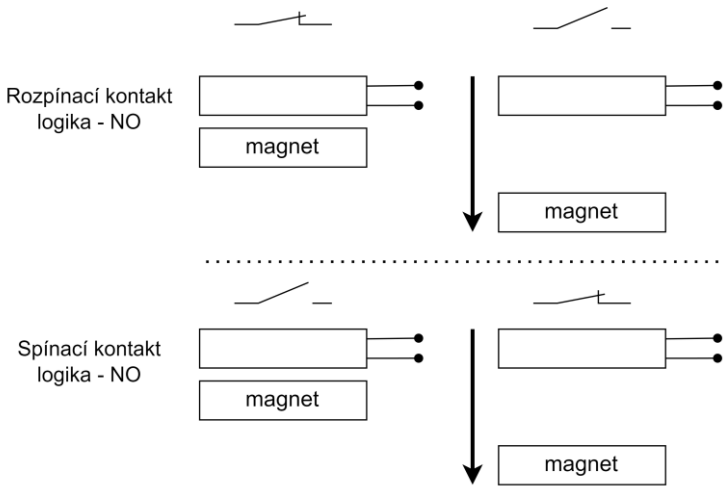
Obr. 2.11 – Připojovací schéma regulátoru FCR010



Obr. 2.12 – Připojený regulátor FCR010 k automatizovanému testeru

2.4.3 Logika vstupů

Většina pokojových regulátorů, disponuje digitálním vstupem pro okenní kontakt. V ukázkových testech, je pracováno s jeho nasimulovaným spínáním. Je zmíněna i logika okenního kontaktu. Z tohoto důvodu, jsou na obr. 2.13 zobrazeny varianty magnetických kontaktů s jazýčkovým relé, které se běžně používají např. v zabezpečovací technice.



Obr. 2.13 – Logika okenního kontaktu

2.5 Databáze pro obslužnou aplikaci

Aplikace pro správu dat používá relační databázi PostgresSQL, která je koncipována s ohledem na efektivní organizaci dat a snadné rozšiřování v budoucnu. Tato databáze je strukturována do tří tabulek, které jsou vzájemně propojeny prostřednictvím klíčů. Pro lepší pochopení struktury je k dispozici logický model databáze, který je znázorněn na obr. 2.14. Dále je každá tabulka popsána a je zde zobrazen ukázkový záznam dat, aby bylo možné lépe pochopit, jak mezi sebou data souvisí. Databáze je navrhnuta v anglickém jazyce s ohledem na kód aplikace, který je také psán v angličtině. Zajišťuje to větší kompaktnost a orientaci v kódu. Sloupce ukázkových záznamů jsou popsány v češtině. Tabulka „device“ obsahuje obecné údaje testovaných zařízení.

Tab. 2.7 – Ukázkový záznam tabulky "device"

identifikační číslo zařízení	název zařízení	stručný popis zařízení	vstupní napětí	typ napětí	příkon	počet Modbusových registrů
1	EPC102	regulátor topení, komunikativní	24 V	AC	6 VA	40

V tabulce „script“ jsou uchovávána data o konkrétním skriptu, který má za úkol zařízení otestovat. Tato data jsou zobrazována při vybírání testů, které chceme pomocí aplikace spustit. Jakým způsobem jsou v aplikaci data zobrazena je popsáno v pododdílu „Hlavní okno aplikace“.

Tab. 2.8 – Ukázkový záznam tabulky "script"

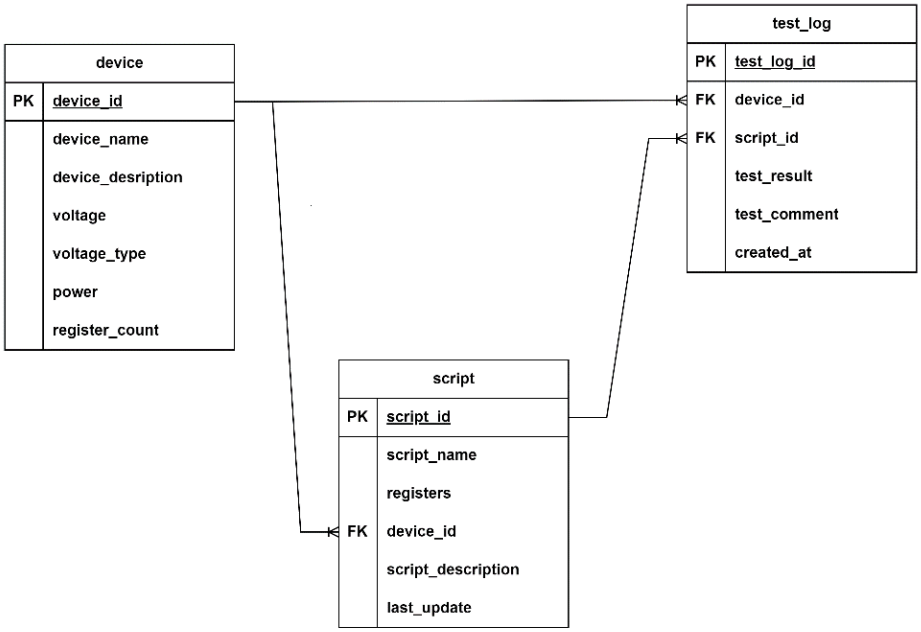
identifikační číslo skriptu	název skriptu	použité Modbusové registry	identifikační číslo zařízení	popis skriptu	datum a čas poslední aktualizace skriptu
3	test okenního kontaktu	7MSB	1	Test kontroluje reakci regulátoru na vstupní DI kontakt.	31.03.2024 21:56

Poslední tabulkou databáze je tabulka „test_log“. Do této tabulky se zapisují výsledky vykonaných testů. Každému řádku v tabulce odpovídá výsledek jednoho testu. Když se tedy v jeden moment spustí např. 10 testů. V tabulce „test_log“ přibude 10 řádků. Proces ukládání výsledků testů je tedy plně automatizován aplikací. Záznam obsahuje informace:

Tab. 2.9 – Ukázkový záznam tabulky "test_log"

identifikační číslo záznamu testu	identifikační číslo zařízení	identifikační číslo skriptu	výsledek testu	komentář výsledku testu	datum a čas testu
145	1	3	true	Zařízení reaguje správně na sepnutý kontakt	31.03.2024 22:14

Vazby mezi tabulkami jsou definovány následovně. Tabulka „device“ je spojena vazbou 1:N s tabulkami „script“ a „test_log“ Toto vazbové schéma znamená, že právě jednomu zařízení může odpovídat více přepisů skriptů a více výsledků testů. Dále je tabulka „script“ propojena vazbou 1:N s tabulkou „test_log“, což znamená že každému skriptu může odpovídat více výsledků testů. Na logickém modelu databáze je zobrazeno, které sloupce jsou primárními a cizími klíči.



Obr. 2.14 – Logický model databáze

Jelikož SQL databáze je typu Postgres, pro správu databáze je použito prostředí *pgAdmin.* V tomto prostředí je dále zobrazen příklad spojení tabulek. Spojení tabulek, tzv. „Join“, je potřeba využít při zobrazení výsledků testů. V ukázkovém záznamu tabulky „test_log“ jsou zobrazeny sice výsledky testů, ale v této tabulce nejsou zobrazeny vůbec informace o zařízeních, která byla testována. Jsou tam pouze identifikační čísla zařízení, protože konkrétní data jsou v tabulce „device“. SQL příkaz zobrazený na obr. 2.15, spojí tabulky „test_log“ a „device“ přes sloupec „device_id“ a zobrazí sloupce: „device_name“, „test_result“ a „created_at“. Výsledná data jsou zobrazena na obr. 2.16.

```
SELECT device_name, test_result, result_comment, created_at
FROM test_log
INNER JOIN device
ON test_log.device_id = device.device_id;
```

Obr. 2.15 – Příkaz pro spojení tabulek

device_name character varying (50)	test_result boolean	result_comment character varying (500)	created_at timestamp without time zone
EPC102	false	Výsledek testu 1 - Nejedná se o EPC102	2024-04-28 17:36:03.431249
EPC102	true	Výsledek testu 1 - Jedná se o správné zařízení.	2024-04-28 18:05:30.071078
EPC102	true	Výsledek testu 2 - Zařízení je dostupné na adresách 1 - 252.	2024-04-28 18:14:14.360977
EPC102	true	Výsledek testu 3 - Zařízení reaguje správně na okenní kontakt.	2024-04-28 18:14:57.882975
EPC102	true	Výsledek testu 4 - Regulátor zaznamenal odpojenou sběrnici one-wire.	2024-04-28 18:18:01.033221
EPC102	true	Výsledek testu 5 - Regulátor reaguje správně na všechny změny provozních režimů.	2024-04-28 18:18:13.70575
EPC102	true	Výsledek testu 6 - Regulátor reaguje správně na vypnutí provozního režimu den a noc, zapomocí otevření okna.	2024-04-28 18:19:22.635162
EPC102	true	Výsledek testu 7 - Regulátor správně reaguje na změnu výstupu a na změnu polarity ventilu při manuálním řízení.	2024-04-28 18:19:52.793795
EPC102	true	Výsledek testu 8 - Výstup regulátoru správně přepočítává zadanou hodnotu na PWM sekvenci při NC polaritě.	2024-04-28 18:22:14.521634

Obr. 2.16 – Výsledná data po spojení tabulek

2.6 FTP server

Na FTP serveru, jsou uložena data, která jsou aplikací zobrazována uživateli a není možné je uložit do databáze. Jedná se o tato data:

- fotky regulátorů,
- schémata připojení regulátorů k automatizovanému testeru,
- připojovací pokyny.

Každá skupina dat, je uložena ve své vlastní podsložce. Při každém startu aplikace je vytvořena kopie dat z FTP serveru v adresáři „C:\Users\Uzivatel\AppData\Roaming\AutomaticTest\ \download“. Je tím zajištěna aktuálnost a centralizace. Je to velice výhodné z pohledu nezávislosti spuštění na konkrétním počítači a možnosti editace z jednoho místa.

Mohlo by být namítnuto, proč nejsou připojovací pokyny uloženy v databázi, stejně jako ostatní textová data. Připojovací pokyny jsou totiž uloženy v *html* souboru, který v sobě uchovává i informace o formátování textu. Důvodem tohoto rozdělení dat aplikace bylo to, že připojovací pokyny v okně aplikace jsou různě zabarveny podle vodičů automatizovaného testeru, obdobně jako v připojovacích schématech. Při případné úpravě zapojení, není nutné složitě upravovat kód aplikace, ale pouze html soubor centrálně, tj. na FTP serveru.

2.7 OBSLUŽNÁ APLIKACE

2.7.1 Verzování aplikace při vývoji

Během vývoje aplikace byl použit verzovací systém Git, který je standardem pro zálohování a správu verzí aplikací. Pomocí tohoto systému jsou zaznamenávány změny souborů a ukládány do tzv. "commitů". Mezi těmito "commity" bylo možné složku (repozitář) s projektem aplikace přepínat, což umožňovalo vrátit ji do stavu, kdy určitá část aplikace ještě fungovala správně, v případě, že došlo k chybě při úpravách kódu.

2.7.2 Konfigurační soubor

Konfigurační soubor zajišťuje oddělení konfiguračních informací od zdrojového kódu aplikace, čímž je zajištěna modulárnost tohoto řešení. Umožňuje snadné změny konfiguračních dat bez nutnosti úprav v samotném kódu aplikace. Tímto způsobem je zaručena vyšší modularita a snadnější údržba aplikace. Z hlediska bezpečnosti je důležité, že přístupové údaje k částem aplikace jsou v konfiguračním souboru uloženy. Typ souboru byl zvolen JSON, což

je zkratka JavaScript Object Notation. Jedná se o populární formát, který je používán napříč programovacími jazyky. Jak je z názvu patrné, vzešel z programovacího jazyka JavaScript. Výhodou JSON souboru je, že jej lze upravovat v obyčejném textovém editoru. Na obr. 2.17 je zmíněný soubor zobrazen. Ukázkový soubor je upraven tak, aby nezobrazoval žádné přístupové údaje (Dokumentace JSON, 2019).

Obr. 2.17 – Konfigurační soubor

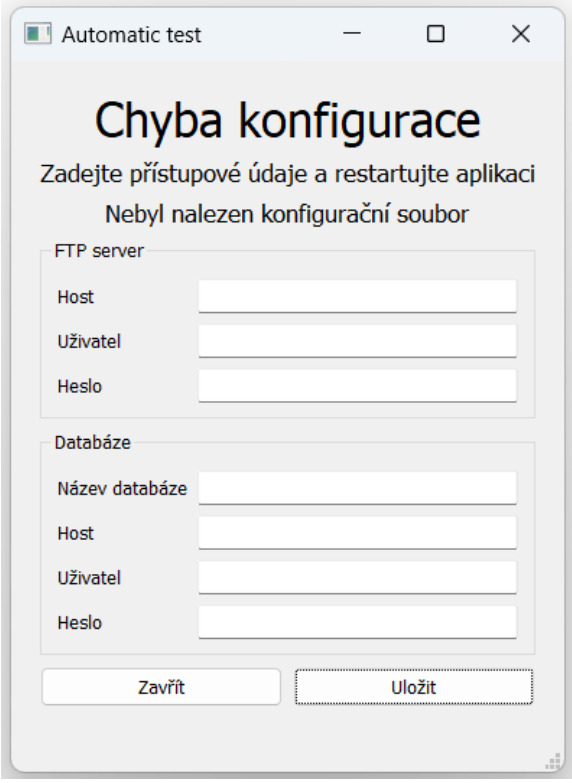
2.7.3 Spuštění aplikace

Při prvním spuštění aplikace dochází k několika klíčovým procesům. Aby bylo možné aplikaci spustit na jakémkoli počítači s operačním systémem Windows, byla do aplikace přidána následující logika. K připojení na FTP server a SQL databázi, jsou potřeba přístupové údaje. Jelikož není správnou programovací praktikou ukládat přístupové údaje aplikace přímo do kódu aplikace, ani do složky s projektem aplikace, jsou tato data uložena v konfiguračním souboru, který je při startu aplikace přečten.

Konfigurační soubor je správně ukládán do složky *„\AutomaticTest\“*, která je automaticky vytvořena při prvním spuštění aplikace v cestě: *„C:\Users\Uzivatel\AppData\ \Roaming\“*. Složka *„\AppData\Roaming\“* je standardně určena pro ukládání systémových dat aplikací v operačním systému Windows. V rámci uživatelských účtu systému Windows jsou systémová data vzájemně oddělena díky čemuž lze provádět nezávislou konfiguraci obslužné aplikace (Hartinger, 2022).

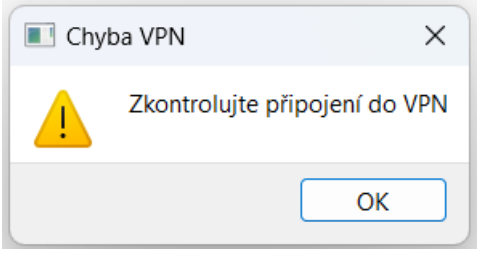
Aplikace vždy automaticky odkazuje na složku *„\AppData\Roaming\“* právě přihlášeného uživatele.

Při prvním spuštění aplikace, přístupové údaje ještě nebyly nastaveny, tj. nebyl vygenerován konfigurační soubor. Z tohoto důvodu, je zobrazeno chybové okno (viz obr. 2.18). To uživateli sdělí, jak konfiguraci nastavit. Po následném zadání přístupových údajů je



Obr. 2.18 – Chybové okno aplikace 1

vygenerován konfigurační soubor, který je uložen do složky *„\AutomaticTest\“*. Při stisku tlačítka uložit se zobrazí varovné „pop-up“ okno (viz obr. 2.19), které uživatele informuje o zkontrolování připojení počítače do sítě VPN. Zmíněný FTP server, na kterém jsou uložena data zobrazována aplikací, je totiž dostupný pouze z VPN sítě. Po kliknutí na tlačítko ok, se obě okna zavřou, a uživatel může aplikaci opětovně spustit.



Obr. 2.19 – Varovné „pop-up“ okno

V případě, že uživatel zadá správné přístupové údaje k FTP serveru a SQL databázi, aplikace stáhne z FTP serveru data a uloží je do složky *„\AutomaticTest\download\“*. Následně

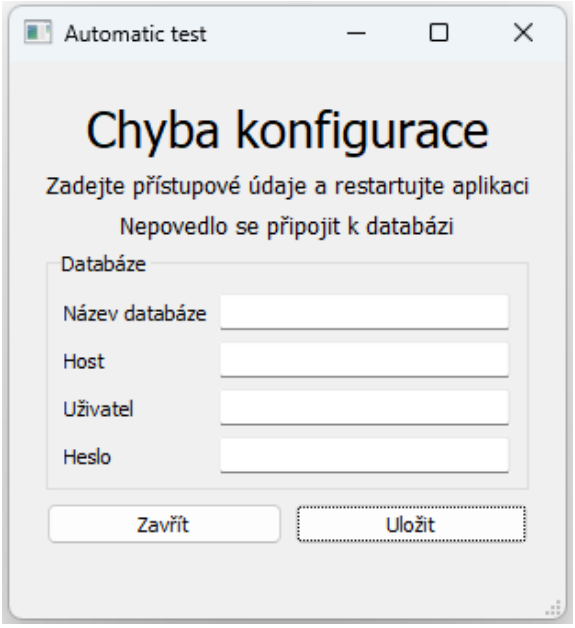
je spuštěno hlavní okno aplikace (viz obr. 2.22). Stahovaná data jsou popsána v pododdílu FTP server.

Může ale dojít k situaci, kdy uživatel zadá špatné přístupové údaje třeba jen k databázi. V tomto případě, je sestavenou chybové okno tak, aby uživatel opravil pouze chybné přístupy databáze. V okně můžou být zobrazeny následující chybové hlášky, které uživatele informují o vadné konfiguraci aplikace:

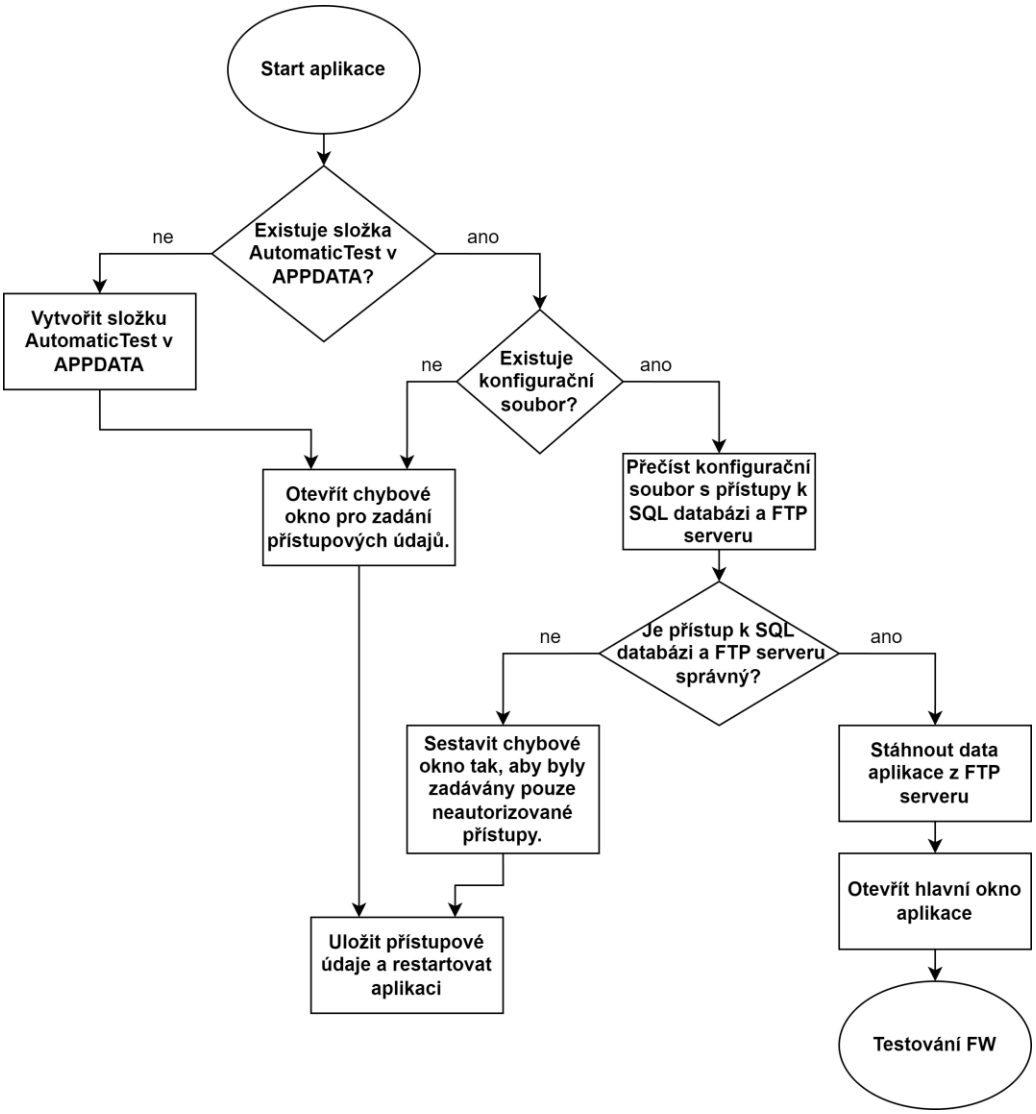
- Nepovedlo se připojit k databázi.
- Nepovedlo se připojit k FTP serveru.
- Nepovedlo se připojit k databázi a FTP serveru.
- Nebyl nalezen konfigurační soubor.

Příklad takto sestaveného chybového okna, je zobrazen na obr. 2.20. Vstupní pole pro zadání přístupů k FTP serveru se v tomto příkladu chybového okna nezobrazují, protože by se musely znovu vyplnit, což není žádoucí, když je už napojení na FTP server aktivní. Při zadávání hesel, jsou samozřejmě místo skutečných znaků hesla zobrazovány pouze hvězdičky.

Pro snadnější pochopení, jak aplikace při startu pracuje, je nakreslen zjednodušený vývojový diagram startu aplikace (viz obr.2.21).

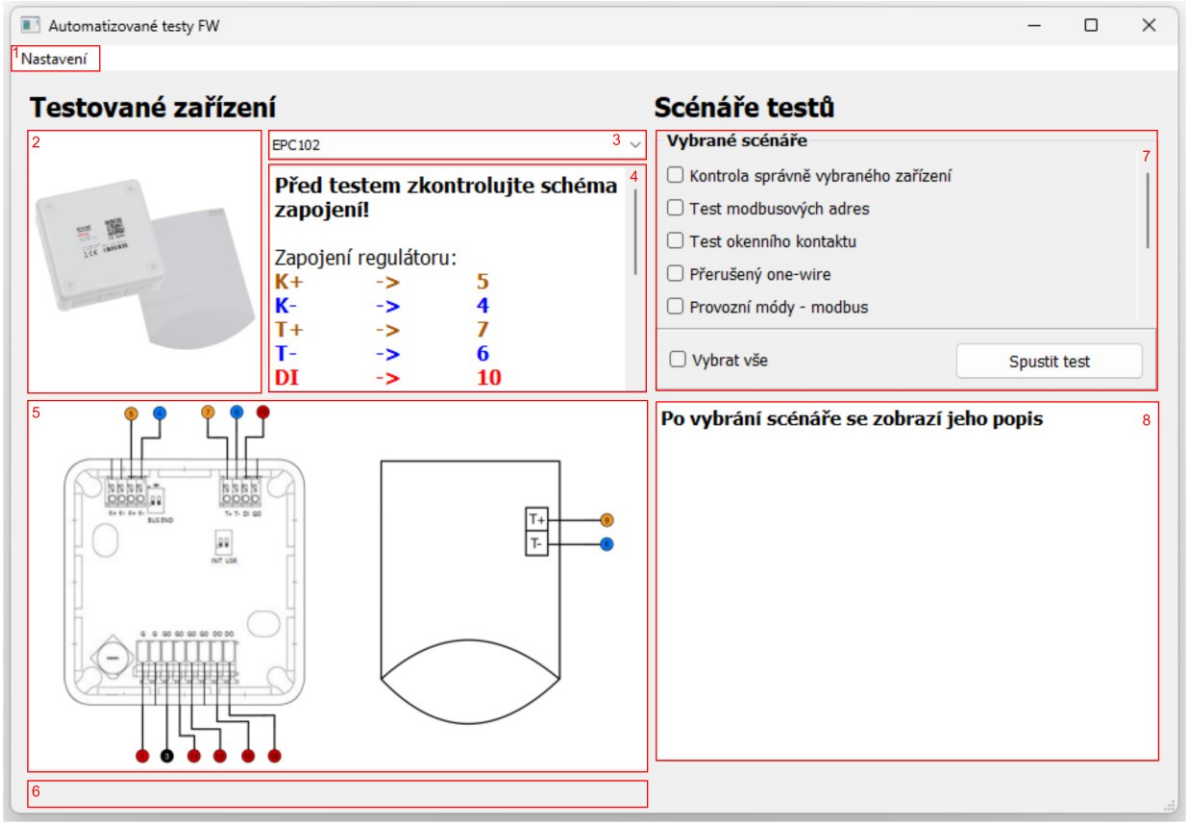


Obr. 2.20 – Chybové okno aplikace 2



Obr. 2.21 – Vývojový diagram startu aplikace

2.7.4 Hlavní okno aplikace



Obr. 2.22 – Hlavní okno aplikace

Hlavní okno aplikace se začne načítat až v momentě, kdy jsou při spuštění aplikace zadány správné přístupové údaje. Při načítání aplikace se na pozadí stáhnou potřebná data z FTP serveru. Po stažení dat, je zobrazeno hlavní okno aplikace (viz obr. 2.22). Proces načítání trvá cca 4 vteřiny. Hlavní okno je složeno z několika oblastí, tzv *widgetů*, které jsou pro názorný popis očíslovány a ohraničeny červenými obdélníky. V textu dále je na konkrétní *widgety* pro jasnější vysvětlení funkce okna aplikace odkazováno.

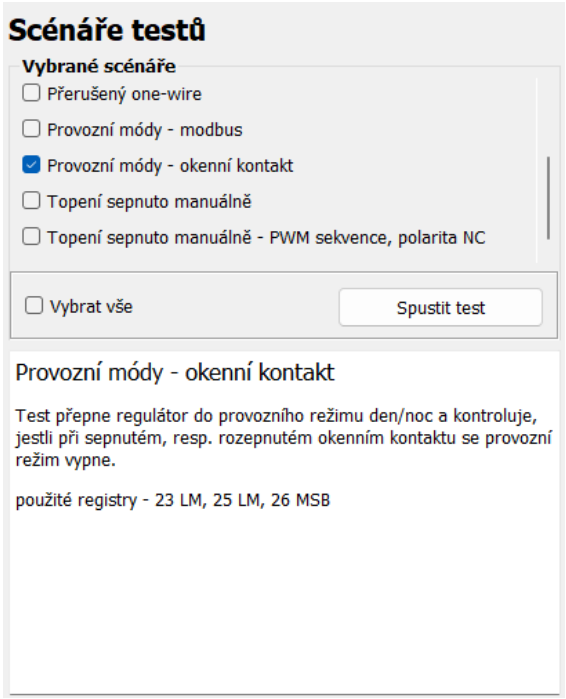
Okno je rozděleno do dvou částí. Levá část se týká testovaného zařízení, pravá scénářů testů naprogramovaných pro dané zařízení. Operátor musí vybrat zařízení, které chce nechat aplikací automaticky otestovat. Výběr zařízení, které je možné otestovat se provede pomocí rozbalovací lišty (oblast 3).

Po vybrání se *widgety* okna aplikace překreslí. V levé části jsou zobrazeny tato data:

- fotka testovaného zařízení (oblast 2),
- připojovací pokyny zařízení k automatizovanému testeru s popisem zapojení, které je barevně označeno (barva textu odpovídá barvě vodiče) (oblast 4),
- připojovací schéma zařízení k automatizovanému testeru (oblast 5).

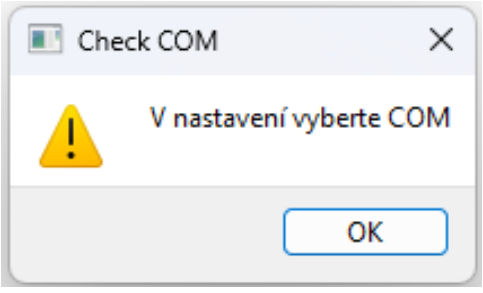
Tato data aplikace jsou centrálně uložena na FTP serveru.

V pravé části aplikace jsou *widgety*, která se týkají samotných testů zařízení. V oblasti 7 jsou k vybrání scénáře testů. Vybrat scénáře lze pomocí zaškrtávacích políček jednotlivě, nebo všechny najednou pomocí políčka „Vybrat vše“. Zaškrtávací políčka se generují po vybrání testovacího zařízení, na základě dat v Postgress SQL databázi. Po vybrání testovacího zařízení jsou z databáze vrácena data s názvy testů, na základě kterých jsou vygenerována zaškrtávací políčka. Po výběru testovacího scénáře v textové oblasti 8 dojde k zobrazení popisu konkrétního testu a použité Modbusové registry zařízení. Tato data jsou rovněž vyčteny z Postgress SQL databáze. Příklad takto zobrazeného popisu testu je na obr. 2.23.



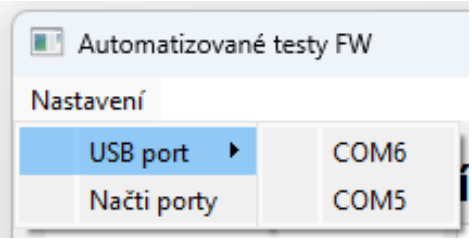
Obr. 2.23 – Hlavní okno aplikace – vybrání testu

Po stisku tlačítka spustit test je zobrazeno varovné okno (viz obr. 2.24), které uživatele informuje o kontrole nastavení COM portu automatizovaného testeru. Pro nastavení COM portu je nutné otevřít v horní liště záložku nastavení. Pro zjištění čísel COM portu slouží tlačítko „Načíst porty“. Po stisku tohoto tlačítka dojde k zobrazení seznamu dostupných COM portů v počítači (viz obr. 2.25). Je doporučeno před a po připojení automatizovaného testeru do



Obr. 2.24 – Varovné okno – nastavení COM portu

USB portu provést kontrolu dostupných portů. Port, který se objeví v nabídce po připojení testeru, je portem přidělený tomuto zařízení

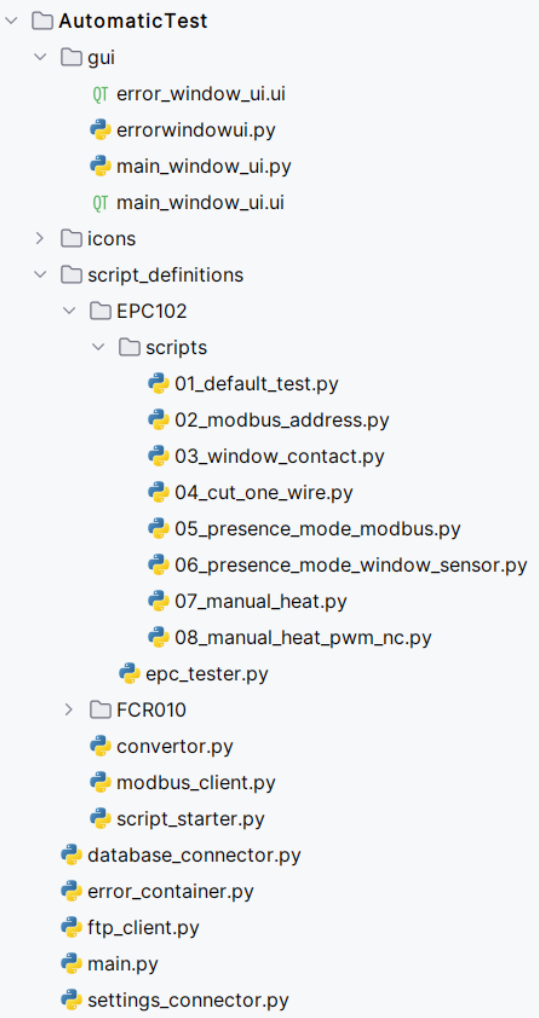


Obr. 2.25 – Nastavení COM portu

Poslední částí hlavního okna aplikace je stavový proužek (oblast 6), který informuje operátora testu o stavu aplikace po úspěšném spuštění testovací sekvence. Ve stavovém proužku je zobrazena hláška – „Probíhá test – nevypínejte aplikaci“. Po každém ukončeném testu je ve stavovém proužku zobrazena hláška, která operátora informuje o výsledku daného testu např. „Výsledek testu 1 – Jedná se o správné zařízení“ nebo „Test 1 selhal“.

2.7.5 Struktura aplikace

V tomto pododdílu je popsána struktura nesestavené aplikace, kterou lze zobrazit např. ve vývojovém prostředí PyCharm (viz obr. 2.26). Rozdělení kódu do jednotlivých souborů bylo vymyšleno tak, aby bylo co nejvíce srozumitelné a v budoucnu byla zajištěna snadná rozšiřitelnost. Celá aplikace je psaná tak, že každé třídě odpovídá jeden soubor, to znamená, že nejsou definovány v jednom souboru dvě třídy, ale právě jedna. Každá třída uchovává metody, které se zabývají funkcionalitou konkrétní části aplikace.



Obr. 2.26 – Struktura obslužné aplikace

2.7.6 Popis programu

Následující odstavce korespondují s jednotlivými soubory obslužné aplikace (projektu), přičemž jsou uspořádány tak, aby odpovídaly provozní sekvenci aplikace. Ke každému odstavci je zobrazen vybraný kód. Výňatky kódu nemusí zahrnovat všechny části příslušného souboru, například nejsou zahrnuty importy mezi soubory a importy použitých knihoven. Celý kód obslužné aplikace je k dispozici v příloze A.

Jako první je popsán soubor *„main“* (viz obr. 2.27), který je hlavním spustitelným skriptem programu. Na začátku souboru je zkontrolováno, zda je aktuální soubor spuštěn jako hlavní program pomocí podmínky *„if __name__ == '__main__'“* Tento řádek kódu zajišťuje, že kód uvnitř této podmínky se provede pouze tehdy, když je tento soubor spuštěn přímo jako samostatný program, a nikoli když je importován jako modul do jiného skriptu. To je obecná praxe v jazyce Python, která pomáhá oddělit kód určený ke spuštění, od kódu, který slouží jako

modul pro importování do jiných částí projektu. Dále jsou v souboru „main“ vytvořeny objekty na obsluhu chybového kontejneru, konfiguračního souboru, FTP serveru, SQL databáze a grafického uživatelského rozhraní, které jsou popsány v následujících odstavcích (Pecinovský, 2020).

```
if __name__ == "__main__":
    err_container = ErrorContainer()
    sett_connector = SettingsConnector()
    if (sett_connector.read_settings_file()):
        ftp_c = FtpClient()
        db_c = DatabaseConnector()
        is_ftp_connected = ftp_c.test_ftp_connection()
        is_db_connected = db_c.test_database_connection()
        if (is_ftp_connected and is_db_connected):
            if (ftp_c.download_device_photo() and ftp_c.download_schemes() and ftp_c.download_schemes_description()):
                Main_window_ui.run_application()
        if (is_ftp_connected and not is_db_connected):
            ErrorWindowUi.run_application( what_hide: "ftp", message: "Nepovedlo se připojit k databázi")
        if (not is_ftp_connected and is_db_connected):
            ErrorWindowUi.run_application( what_hide: "db", message: "Nepovedlo se připojit k FTP serveru")
        if (not is_ftp_connected and not is_db_connected):
            ErrorWindowUi.run_application( what_hide: None, message: "Nepovedlo se připojit k databázi a FTP serveru")
    else:
        ErrorWindowUi.run_application( what_hide: None, message: "Nebyl nalezen konfigurační soubor")
    print(err_container.get_errors())
```

Obr. 2.27 – Soubor main.py

Během vývoje aplikace bylo potřeba na jednom místě shromažďovat chyby, které nastaly. Tento problém byl vyřešen vytvořením třídy *„ErrorContainer“* (viz obr. 2.28), která byla navrhnuta podle návrhového vzoru *singleton*. Tento návrhový vzor zjišťuje, že instance dané třídy existuje pouze jednou. V programování se často vyskytuje potřeba sdílet jednu instanci objektu mezi různými částmi programu, aniž by bylo nutné tuto instanci předávat v konstruktorech. Prvním krokem je zabránit uživateli vytvářet nové instance třídy. Toho dosáhneme vytvořením privátního konstruktoru bez parametrů. Poté vytvoříme běžnou instanční proměnnou a inicializujeme ji instancí, kterou chceme sdílet v programu. V tomto případě instancí objektu pro skladování chybových hlášek běhu kódu. Chyby jsou ukládány do obyčejného listu (pole). Pro vyčítání a ukládání chyb slouží metody: *„get_errors()“* a *„log_error(message)“*. Pomocí tohoto vzoru jsou také definovány třídy: „*SettingsConnector*“, „*FtpClient*“ a „*DatabaseConnector*“ (Hartinger, 2021).

```
class ErrorContainer:
    _instance = None
    M4rtinP0 *
    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
            cls._instance.errors = []
        return cls._instance
    11 usages  M4rtinP0 *
    @classmethod
    def log_error(self, message):
        self.errors.append(message)
    1 usage  M4rtinP0 *
    @classmethod
    def get_errors(self):
        return self.errors
```

Obr. 2.28 – Třída ErrorContainer

„SettingsConnector“ je třída, která spravuje metody pro práci s konfiguračním souborem. Jedná se opět o návrhový vzor *singleton*, protože připojení ke konfiguračnímu souboru je potřeba ve více souborech a je zbytečné vždy vytvářet novou instanci. Nebylo by to ani správné, protože při každé nové definici objektu by se přepsaly atributy třídy. Jelikož v atributech třídy jsou ukládány přístupové údaje, je žádoucí, aby byly z konfiguračního souboru přečteny pouze jednou, a to při startu aplikace. K popisu byly vybrány metody pro vytvoření složky včetně konfiguračního souboru (viz obr. 2.29). Dále jsou ve třídě metody pro aktualizaci konfiguračního souboru a k jeho přečtení, ty jsou ale obdobné metodám již zmíněným, tj. pro jeho vytvoření. Obě metody, resp. většina kritického kódu aplikace, je obalena *try-except* blokem. V případě zkolabování části kódu obsaženého v bloku *try* dojde k vykonání kódu obsaženém v bloku *except*. V bloku *except* jsou právě zaznamenávány dříve zmíněné chyby. K vytvoření složky je potřeba zjistit cestu k aktuální složce *AppData* (záleží na přihlášeném uživateli) a spojit ji s názvem složky. Následně je zkontrolováno, jestli složka již existuje a případě je v zadané cestě vytvořena, v opačném případě je tento příkaz přeskočen. V metodě pro vytvoření konfiguračního souboru, je cesta rozšířena právě o název tohoto souboru. Na vstupu metody je objekt, který uchovává slovník (datová struktura v Pythonu) s daty k uložení, který je rozdělen do dvou klíčů *„ftp“* a *„database“*. Následně je soubor uložen jako JSON a naformátován tak, aby ho bylo snadné číst v textovém editoru. Což zajišťuje parametr „indent=2“.

```
1 usage
def create_appdata_folder(self):
    try:
        self.file_path = os.path.join(os.getenv("APPDATA"), "AutomaticTest")
        if not os.path.exists(self.file_path):
            os.mkdir(self.file_path)
        self.file_path = os.path.join(self.file_path,self.settings_file_name)
    except:
        self.err_container.log_error(f"Nepodařilo se vytvořit složku {format(self.file_path)},"
                                     f" zkontrolujte prosím svá oprávnění.")
1 usage
def create_settings_file(self, settings):
    try:
        file_path = os.path.join(os.path.join(os.getenv("APPDATA"), "AutomaticTest"),self.settings_file_name)
        with open(file_path, 'w') as js_file:
            json.dump(settings, js_file,indent=2)
    except:
        self.err_container.log_error("Nepovedlo se vytvořit settings.json soubor.")
```

Obr. 2.29 – Metody třídy SettingsConnector

Třída *„Ftp_client“* je určena k provádění operací stahování dat z FTP serveru. Před samotným stahováním dat z FTP serveru, je správnou a doporučenou programovací praktikou, si nejdříve ověřit funkčnost napojení na FTP server (viz obr. 2.30). Na základě tohoto ověření je v kódu rozhodováno, jestli je spuštěno hlavní nebo chybové okno aplikace.

```
def test_ftp_connection(self):
    try:
        self.ftp.connect(self.server,self.port)
        self.ftp.login(self.username, self.password)
        self.ftp.cwd(self.ftp_default_folder)
        return True
    except Exception as e:
        self.err_container.log_error(f"Nastala chyba: {e}")
        return False
```

Obr. 2.30 – Ověření připojení k FTP serveru

Pro stahování jednotlivých typů dat z FTP serveru jsou implementovány metody. Tento proces zahrnuje získávání fotografií testovaných zařízení, připojovacích schémat a popisů připojení v HTML souborech. I přes rozdílné typy dat je základní princip stahování stejný, a proto je popsána jen jedna metoda *„download_device_photo()“* (viz obr. 2.31). Průběh této metody začíná vytvořením lokální cesty pro ukládání stažených dat a následnou kontrolou existence adresářů, případně jejich vytvořením. Poté se metoda přihlásí k FTP serveru pomocí přihlašovacích údajů a nastaví pracovní adresář na umístění dat na serveru. Dalším krokem je získání seznamu názvů fotografií voláním metody *„nlst()“* na FTP serveru. Pro každou fotografii v seznamu je vytvořena lokální cesta a samotná fotografie je stahována z FTP serveru pomocí metody *„retrbinary()“*, která umožňuje přenos binárních dat. V případě výskytu jakéhokoliv selhání při stahování, je chyba zaznamenána a metoda vrátí hodnotu *false*. V

opačném případě, tj. v případě úspěšného stažení dat, je vrácena hodnota *true*, což indikuje úspěšné provedení operace.

```
def download_device_photo(self):
    try:
        local_download_path = os.path.join(self.appdata_path, "download")
        local_device_photo_path = os.path.join(local_download_path, "device_photo")
        if not (os.path.exists(local_download_path)):
            os.mkdir(local_download_path)
        if not (os.path.exists(local_device_photo_path)):
            os.mkdir(local_device_photo_path)
        remote_folder = self.ftp_default_folder + "/" + "device_photo"
        self.ftp.connect(self.server, self.port)
        self.ftp.login(self.username, self.password)
        self.ftp.cwd(remote_folder)
        self.remote_devices = self.ftp.nlst()
        for im in self.remote_devices:
            local_im_path = os.path.join(local_device_photo_path, im)
            with open(local_im_path, 'wb') as f:
                self.ftp.retrbinary('RETR ' + im, f.write)
        return True
    except:
        self.err_container.log_error(f"Nepovedlo se stáhnout data z FTP serveru")
        return False
```

Obr. 2.31 – Stažení dat z FTP serveru

Pro komunikaci s databází je definována třída *„DatabaseConnector“*. Metody třídy posílají na databázi SQL příkazy. Pomocí těchto SQL příkazů jsou z databáze potřebná data stahována, nebo naopak ukládána. Jsou to metody pro:

- ověření databázového připojení – *„test_database_connection()“*,
- zápis výsledků testů do databáze – *„write_test_result_to_databas()“*,
- navrácení definovaných názvů testů – *„return_script_names(device_name)“,*
- navrácení popisu testů – *„return_script_info(device_name, script_name)“*.

Ověření databázového připojení funguje obdobně, jako ověření FTP serveru. Na databázi je poslán dotaz, který vrátí informaci o její verzi. V případě úspěšného vykonání a zpracování dotazu je připojení považováno za funkční.

Příklad metody, která zapisuje výsledky do databáze je na obr. 2.32. Nejprve je vytvořeno spojení s databází pomocí poskytnutých přihlašovacích údajů. Poté se pomocí SQL příkazu zapíše nový záznam do tabulky *„test_log“* Pokud zápis proběhne úspěšně, funkce vrátí hodnotu *true*, v opačném případě vypíše chybu a vrátí *false*. Zbylé dvě metody pracují podobně, jen vracejí jiná potřebná data na základě vstupních parametrů.

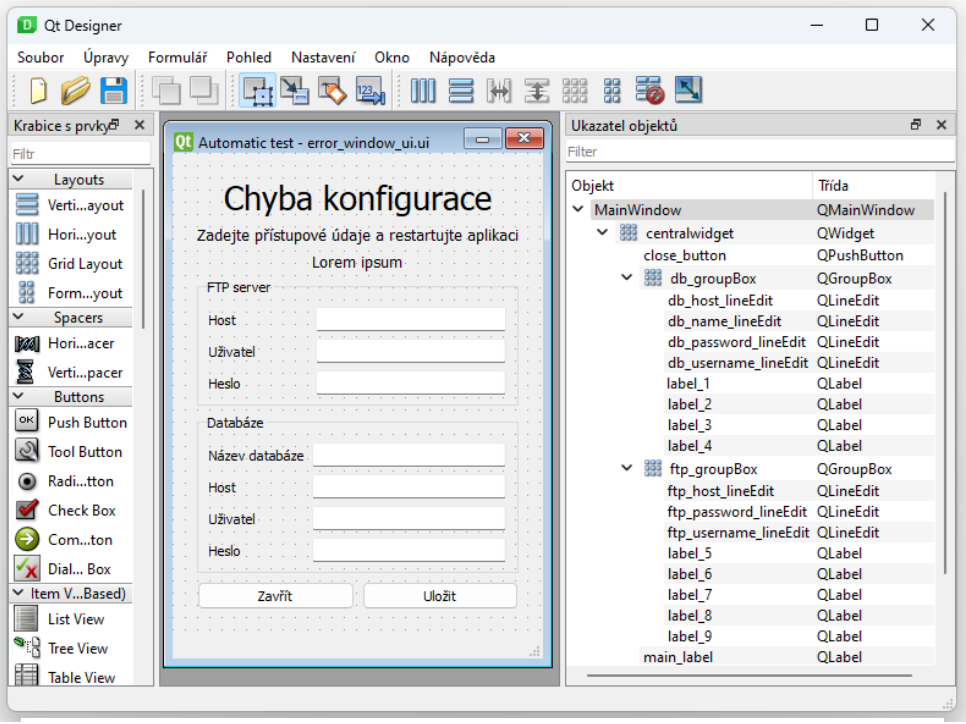
```
def write_test_result_to_database(self, device_id, script_id, test_result, result_comment):
    try:
        connection = psycopg2.connect(
            dbname=self.dbname,
            user=self.user,
            password=self.password,
            host=self.host,
            port=self.port
        )
        cursor = connection.cursor()
        query = "INSERT INTO test_log (device_id, script_id, test_result, result_comment) VALUES (%s, %s, %s, %s);"
        val = (device_id, script_id, test_result, result_comment)
        cursor.execute(query, val)
        connection.commit()
        return True
    except Exception as e:
        print(f"Nepovedl se zapsat výsledek do databáze - {e}")
        return False
```

Obr. 2.32 – Zápis výsledku testů do databáze

Grafické uživatelské rozhraní aplikace bylo navrženo pomocí návrhového programu PyQT Designer, který je integrován do knihovny PyQt. Tento proces umožňuje snadnou tvorbu a úpravu okenních prvků pomocí vizuálního rozhraní. *Widgety* oken aplikace, byly v designeru přesně navrženy a pojmenovány tak, aby odpovídaly jejich funkcím a účelu. Výstupem designeru jsou soubory *„error_window.ui“* a *„main_window.ui“*. Pro implementaci těchto návrhů do kódu aplikace jsou použity třídy *„MainWindow“* a *„ErrorWindow“*. V konstruktorech těchto tříd, jsou jednotlivé widgety z UI soborů vyčteny, a následně je s nimi pracováno v definovaných metodách. (např. znovu vygenerování zaškrtávacích políček). Těmto třídám odpovídají soubory *„main_window.py“* a *„error_window.py“*. Zmíněný postup návrhu okenních aplikací umožňuje efektivní a flexibilní správu uživatelského rozhraní. Vyčtení widgetů a designer jsou zobrazeny na obrázcích 2.33 a 2.34 (Ruscica, 2019).

```
self.findPorts_Action = self.findChild(QAction,"findPorts_Action")
self.USB_port_Qmenu = self.findChild(QMenu,"USB_port_Qmenu")
self.device_foto_label = self.findChild(QLabel,"device_foto_label")
self.device_comboBox = self.findChild(QComboBox,"device_comboBox")
self.device_textBrowser = self.findChild(QTextBrowser, "device_textBrowser")
self.schema_foto_label = self.findChild(QLabel,"schema_foto_label")
self.scrollArea = self.findChild(QScrollArea, "scrollArea")
self.scrollAreaWidgetContents = self.findChild(QWidget,"scrollAreaWidgetContents")
self.select_all_checkBox = self.findChild(QCheckBox,"select_all_checkBox")
self.start_script_pushButton = self.findChild(QPushButton,"start_script_pushButton")
self.script_description_textBrowser = self.findChild(QTextBrowser, "script_description_textBrowser")
self.statusbar = self.findChild(QStatusBar, "statusbar")
```

Obr. 2.33 – Vyčtení widgetů z UI souboru



Obr. 2.34 – pyQT designer

Během testování FW zařízení, je potřeba přepisovat jeho Modbusové registry. V některých případech je potřeba nastavit pouze jeden bit v celém registru naopak někdy je potřeba zapsat dekadickou hodnotu (např. procentuální otevření ventilu). Do Modbusových registrů je ale možné zapisovat pouze v binární formě, proto byla vytvořena třída, která uchovává statické metody pro přepočet dekadické hodnoty na binární list (pole). Na obr.2.35 je zobrazena metoda pro přepočet vstupní hodnoty na osmibitový binární list. Metoda nejdříve zkontroluje, jestli se jedná o dekadickou hodnotu v rozsahu 0 až 255, následně převede hodnotu do binárního čísla, které zleva doplní nulami tak, aby výsledné číslo mělo 8 znaků. Tyto znaky jsou následně rozloženy do listu, který je metodou navrácen.

```
@staticmethod
def convert_8bit_val_to_list(input_value):
    if(input_value>=0 and input_value <=255):
        binval = bin(input_value)[2:].zfill(8)
        output_list = []
        for bit in binval:
            output_list.append(bit)
        list_intu = list(map(int, output_list))
        return(list_intu)
    else:
        return(-1)
```

Obr. 2.35 – Přepočet dekadické hodnoty do binárního listu

Pro zapisování a vyčítání Modbusových registrů pomocí sériové linky počítače byla použita knihovna *„minimalmodbus“*, která disponuje příkazy všech Modbusových funkcí. Tato knihovna byla použita ve třídě *„ModbusClient“* představující klienta pro komunikaci pomocí protokolu Modbus. Při inicializaci v konstruktoru třídy jsou nastaveny parametry jako jsou port, adresa zařízení, aktuální rychlost přenosu dat a parita. Metody třídy umožňují čtení a zápis coilů, ale i celých registrů. Příklad metody pro čtení coilů je na obr. 2.36. Do metody vstupují parametry: číslo registru, počáteční a koncový bit, vyčítaná data a mód, kterým je rozhodnuto, jestli se čte celý 16bitový registr, nebo jenom jeho LSB nebo MSB část. Výstupem metody je 16prvkový nebo 8prvkový list (pole), který odpovídá hodnotě v Modbusovém registru.

```
def read_coils_01(self, start_register, start_bit, end_bit, mode="whole"):
    mode = mode.lower()
    values = []
    try:
        start_register -= 1
        end_bit += 1
        if mode == "whole" or "lsb":
            values = self.instrument.read_bits((start_register * 16) + start_bit, end_bit - start_bit, functioncode=1)
        elif mode == "msb":
            start_bit += 8
            values = self.instrument.read_bits((start_register * 16) + start_bit, end_bit - start_bit, functioncode=1)
        output_value = list(reversed(values))
        return output_value
    except Exception as e:
        return False, e
```

Obr. 2.36 – Modbus funkce Read Coils

2.8 SCÉNÁŘE TESTŮ

Scénáře testů, které jsou popsány v následujícím oddílu, jsou navrhnuty pro regulátor EPC102. V dalším oddílu jsou zjednodušené odpovídající vývojové diagramy. Vývojové diagramy neodpovídají na 100 % kódu testovacích skriptů, ale jsou z důvodu přehlednosti zjednodušeny tak, aby byl zachován princip a logika testu.

2.8.1 Kontrola správně vybraného zařízení

Zdá se to jako bezvýznamný krok, ale určitě je dobrou praktikou na začátku testovacího procesu zkontrolovat správnost výběru připojeného zařízení. Každý regulátor má v Modbusovém registru *1 LM – module id* uloženo své identifikační číslo. Tento registr se nachází v EEPROM paměti a je zabezpečen proti přepisu. Identifikační číslo, které je kontrolováno a odpovídá regulátoru EPC102 má hodnotu 322_{hex} neboli 802_{dec}.

2.8.2 Test Modbusových adres – SW reset

Regulátor musí být dostupný a fungovat na všech Modbusových adresách (0-255). V praxi není běžné dávat na jednu sběrnici více jak 100 Modbusových zařízení, nicméně je vhodné provádět testy v celém možném rozsahu Modbusových adres. Test v cyklu automaticky nastaví na zařízení adresy v rozsahu 0-252 (adresy 253,254 a 255 nejsou uvažovány, protože na těchto adresách pracují moduly automatizovaného testeru). Adresy jsou následně přečteny. Když se přečtená adresa rovná nastavené, zařízení na dané adrese je funkční. Aby se povedla přečíst adresa na zařízení, musí být v kódu vygenerován nový Modbusový klient s novou adresou. Přenastavená Modbusová adresa musí být uložená v EEPROM paměti, aby po restartu zařízení zůstala uložená. Pro nastavení nové adresy je nutné nejdříve u zařízení povolit zápis do EEPROM paměti nastavením nultého bitu v registru *3 LSB – status LSB*. Tento test používá pro restart zařízení SW reset. Restartovat zařízení lze i odpojením a následným připojením napájení, to ale není součástí tohoto testu. SW reset musí být nejdříve povolen nastavením prvního bitu v registru 3 LSB – *status LSB.* Dále je možné zapsat novou adresu do registru *4 LSB – address*. SW resetu se provede po zapsání nenulové hodnoty do registru *1002 LM – SW reset.*

2.8.3 Test okenního kontaktu

V Modbusové mapě zařízení je stavový registr, ve kterém jsou uchovávány data, resp. stavy regulátoru. Je to konkrétně registr *7 MSB – inputs.* První bit tohoto registru uchovává informaci o stavu vstupního okenního kontaktu. Kontakt je tedy sepnut automatizovaným testerem a ze zařízení je po uplynutí přepisovací doby (20 s) vyčtena hodnota bitu. Když regulátor zaznamená sepnutý kontakt, chová se správně. V opačném případě nikoliv. Doba nutná k přepisu registrů se liší. U tohoto bitu je to cca 20 s včetně rezervy, u jiných registrů to může být více. Je to způsobeno tím, že regulátor má v sobě systém priorit a registry s vyšší prioritou přepisuje přednostněji. Jelikož regulátor řídí velmi pomalé soustavy (typicky teplotu v místnosti za pomocí termoelektrických hlavic), rychlejší reakční doba není nutná.

2.8.4 Přerušené 1-Wire čidlo

Pro spolehlivé řízení tepelné soustavy musí být zajištěna ochrana proti nesprávným hodnotám teploty, vlivem odpojeného čidla regulátoru. Když dojde k přerušení vodičů mezi regulátorem a 1-Wire čidlem, regulátor by neměl generovat jakýkoliv akční zásah, protože

nemá informaci o aktuální teplotě v místnosti. Tento test kontroluje, jestli regulátor po odpojení 1-Wire čidla zaznamená odpojené čidlo a přestane generovat akční zásah. Test trvá cca 3 minuty, protože registr, ze kterého je vyčítán stavový bit se přepisuje jednou za cca 1,5 minuty. V testu je reakce zkontrolována 2krát, poprvé při rozepnutí komunikace mezi čidlem a regulátorem, podruhé při opětovném sepnutí. Bit, který je vyčítán a kontrolován, se nachází v registru – *7MSB – inputs*, jedná se o 3. bit.

2.8.5 Provozní módy – Modbus

Regulátor disponuje třemi provozními módy, mezi kterými přepíná na základě požadavků ze sběrnice Modbus RTU. Jsou to provozní módy – den, noc a vypnuto. V každém provozním módu jsou předdefinované různé žádané teploty. Nejpříznivější teplota místnosti je nastavená pro provozní mód „den“. Podle žádaných teplot regulátor generuje akční zásah. V tomto testu je testováno, jestli regulátor na základě požadavků ze sběrnice přepne do správného provozního módu.

2.8.6 Provozní módy – okenní kontakt

Jak již bylo zmíněno, regulátor disponuje digitálním vstupem, který je připojen na okenní kontakt. Při otevřeném okně je okenní kontakt aktivní a na tento stav poté regulátor reaguje. Logika regulátoru je vymyšlena tak, že při otevřeném oknu regulátor negeneruje akční zásah nebo jen minimální. Jak velký akční zásah generuje záleží na nastavené teplotě v provozním módu „vypnuto“. Regulátor totiž při aktivním okenním kontaktu do tohoto provozního módu přepne. Logika okenního kontaktu lze nastavit v registru *26 MSB – inputs settings (inputs enable, inputs logic)* 3. bitem. NO – *normally open* odpovídá logické jedničce. NC – *normally close* odpovídá logické nule. V tomto testu je použita výchozí logika vstupu NO. To znamená, že při otevřeném okně, je digitální kontakt sepnut a regulátor má informaci o otevřeném okně, logika NC funguje obráceně. Jaká logika je zvolena, záleží na typu okenního čidla, které je k regulátoru, resp. k oknu nainstalováno. Během testu musí být povolena reakce okenního kontaktu na změnu provozního módu 1. bitem v registru *26 MSB – inputs settings (inputs enable, inputs logic).* Dále je nastaven regulátor do provozního módu den. Po uplynutí třech vteřin je nasimulováno otevření okna aktivací digitálního vstupu. V tuto chvíli by regulátor měl být přepnut do provozního módu vypnuto. Na tuto reakci testovací aplikace čeká patnáct vteřin. Když regulátor reaguje správně, test pokračuje dál a testovací sekvenci provede

i pro režim den. Žádaný provozní mód se nastavuje v registru *23 LM – set presence mode*. Podle toho, který provozní mód je žádaný, jsou nastaveny bity 0-2:

0. bit…den,
1. bit…noc,
2. bit…vypnuto.

Aby se předešlo nechtěným přepnutím provozního módu. Logika přepínání je ošetřena tak, že se při přepínání musí současně nastavit i 15. bit. Z tohoto důvodu je možné použít pro testování této funkcionality pouze Modbusovou funkci *16 – write register*, která zapisuje do celého registru v jeden okamžik.

2.8.7 Manuální sepnutí výstupu

V regulátoru lze nastavit manuální řízení výstupu, to znamená, že lze ovládat digitální výstup nezávisle na výstupu vnitřního PI regulátoru. Pro přepnutí výstupu na manuální řízení je potřeba nastavit 1. bit v registru *10 LSB manual control*. Obdobně jako u digitálního vstupu lze nastavit logiku výstupu na NO nebo NC. Když je výstup regulátoru nastaven na logiku NO, požadavek na topení odpovídá rozepnutému triakovému výstupu, což znamená, že na výstupu je nulové napětí. V opačném případě, kdy logika výstupu je nastavena na NC, požadavek na topení odpovídá sepnutému triakovému výstupu a bylo by naměřeno cca napájecí napětí regulátoru. Zvolená logika výstupu záleží na vybraném akčním členu, který je k regulátoru připojen. Změnit logiku lze přes registr *26 LSB – regulator settings 5*. bitem: 0 – NC,1 – NO.

V testu je výstup přepnut na manuální řízení a je zadán příkaz na topení na 100 a 0 %. Po zadání příkazu je na automatizovaném testeru vyčtena hodnota z modulu, který je připojen jako protikus. Když vyčtená hodnota souhlasí se zadanou, regulátor se chová správně. V tomto testu jsou testovány obě logiky výstupu.

2.8.8 Manuální sepnutí výstupu – PWM sekvence, polarita NC

Jelikož výstup regulátoru je dvoustavový (přivedeno, nepřivedeno napájecí napětí), nemůže generovat spojitý signál. Když ale vnitřní PI sekvence spočítá akční zásah, třeba na 25 % musí být nějakým způsobem tento požadavek na výstup předán. Tento problém řeší PWM modulátor, který akční zásah přepočítává na PWM sekvenci s periodou 60 vteřin. Když je tedy zmíněný akční zásah aktivní, výstup se chová tak, že je 5 vteřin triakový výstup sepnutý a 15 vteřin rozepnutý. Tato pomalá PWM sekvence akčnímu členu nevadí, protože se obvykle

používají termoelektrické hlavice, které mají dobu přestavení mezi krajními body ventilu cca 3 minuty.

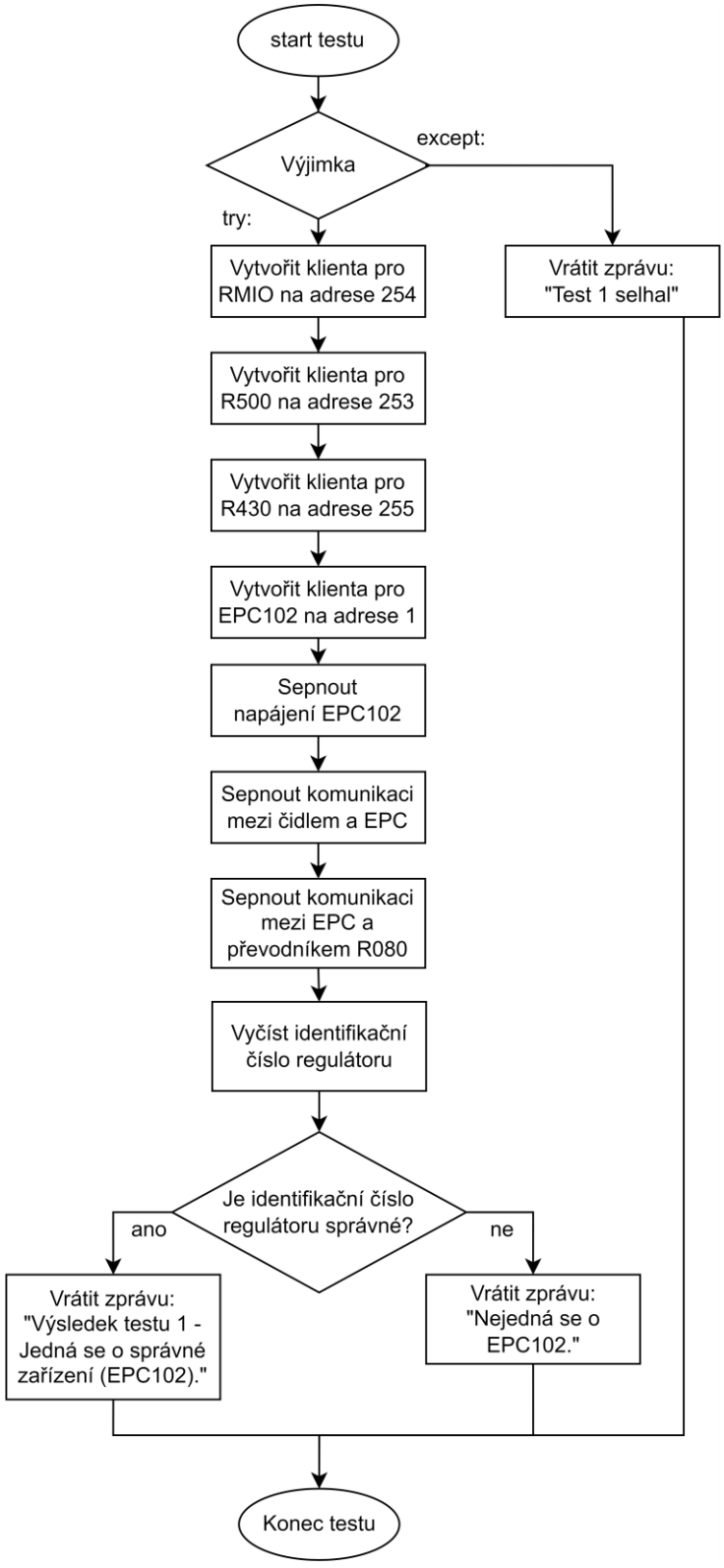
V tomto testu je tedy kontrolován zmíněný PWM modulátor a to tak, že zadaná hodnota musí odpovídat PWM sekvenci. Jelikož by bylo obtížné měnit žádanou teplotu, tak aby PI regulátor generoval předem stanovený akční zásah (který záleží na aktuální teplotě). PWM modulátor je testován za pomocí manuálně přepnutého výstupu. V testu jsou zadány 3 žádané hodnoty a je počítáno, jestli se zadané hodnoty shodují s časy PWM sekvence. Zvolené hodnoty odpovídají dobám zobrazených v tab. 2.10.

Tab. 2.10 – Ukázkový záznam tabulky "script"

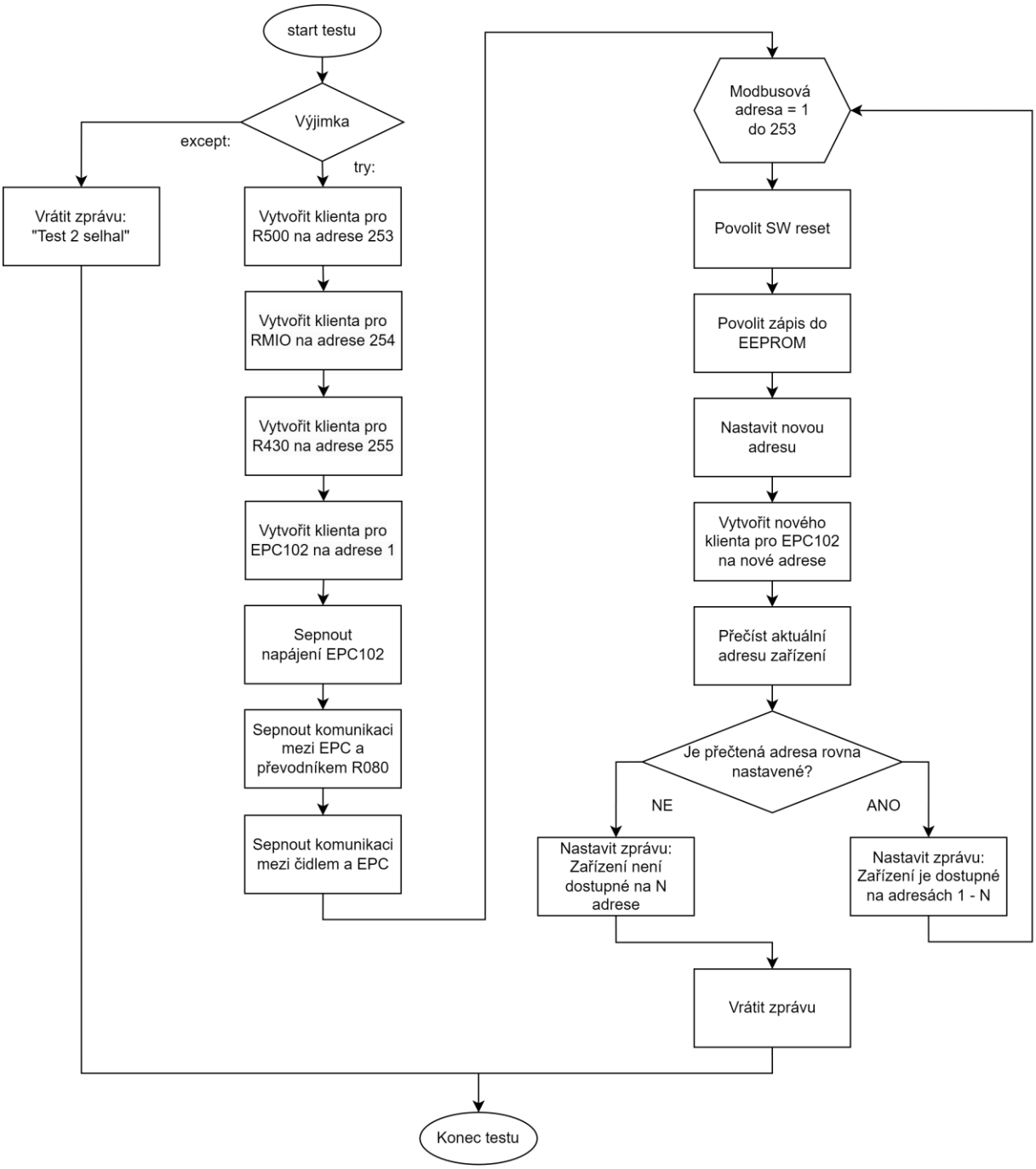
Otevření ventilu	Doba sepnutého výstupu (topí)	Doba rozepnutého výstupu(netopí)
25 %	5 s	15 s
50 %	10 s	10 s
75 %	15 s	5 s

Automatizovaný tester měří dobu sepnutého výstupu. Aby bylo dosaženo objektivního výsledku, zadaná doba se měří 3×. Pro kontrolu správnosti PWM modulátoru se vypočítá průměr těchto hodnot. Pro určení, jestli je sepnutá doba správná, byla nastavena tolerance 500 ms.

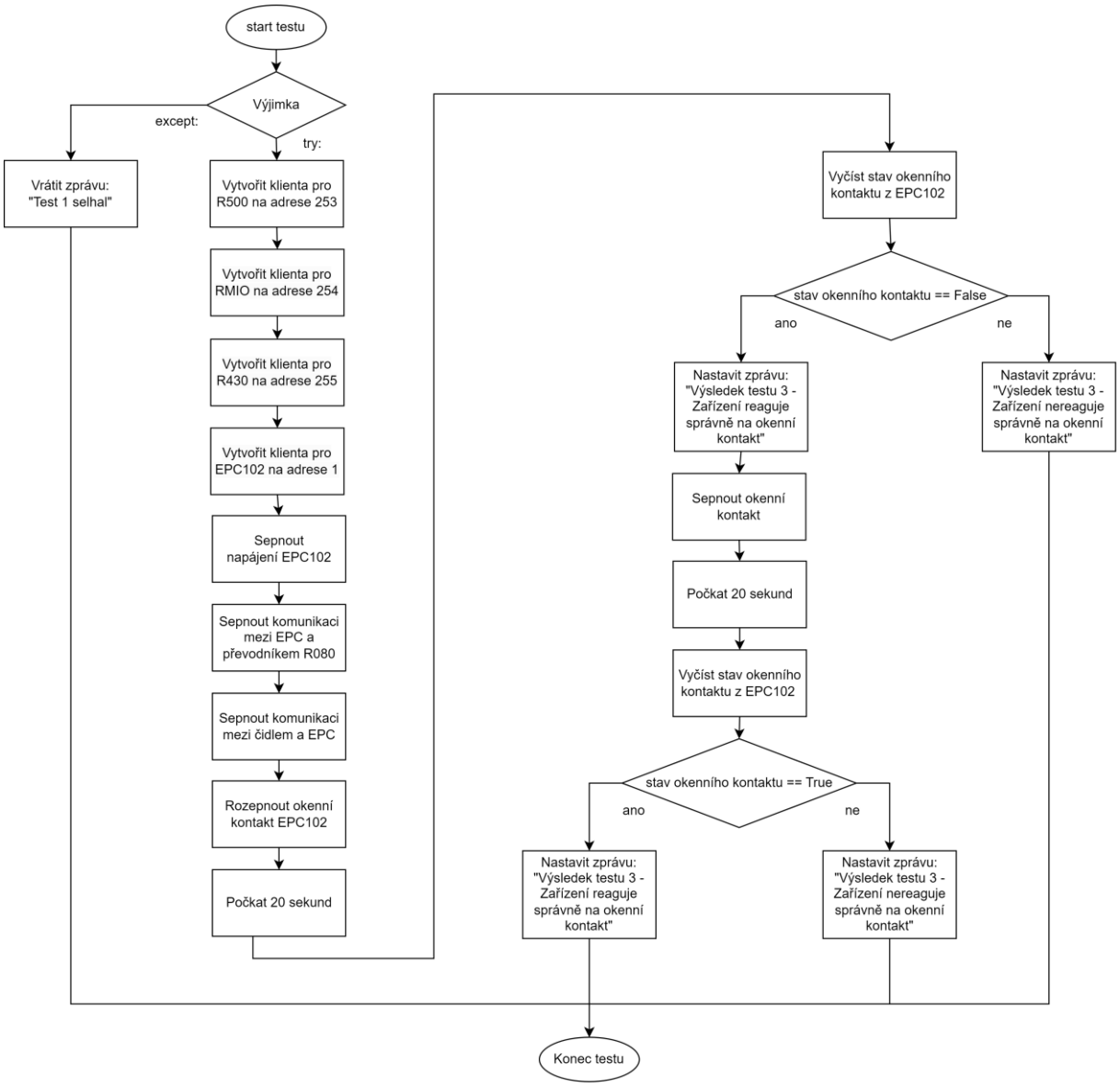
2.9 VÝVOJOVÉ DIAGRAMY TESTŮ



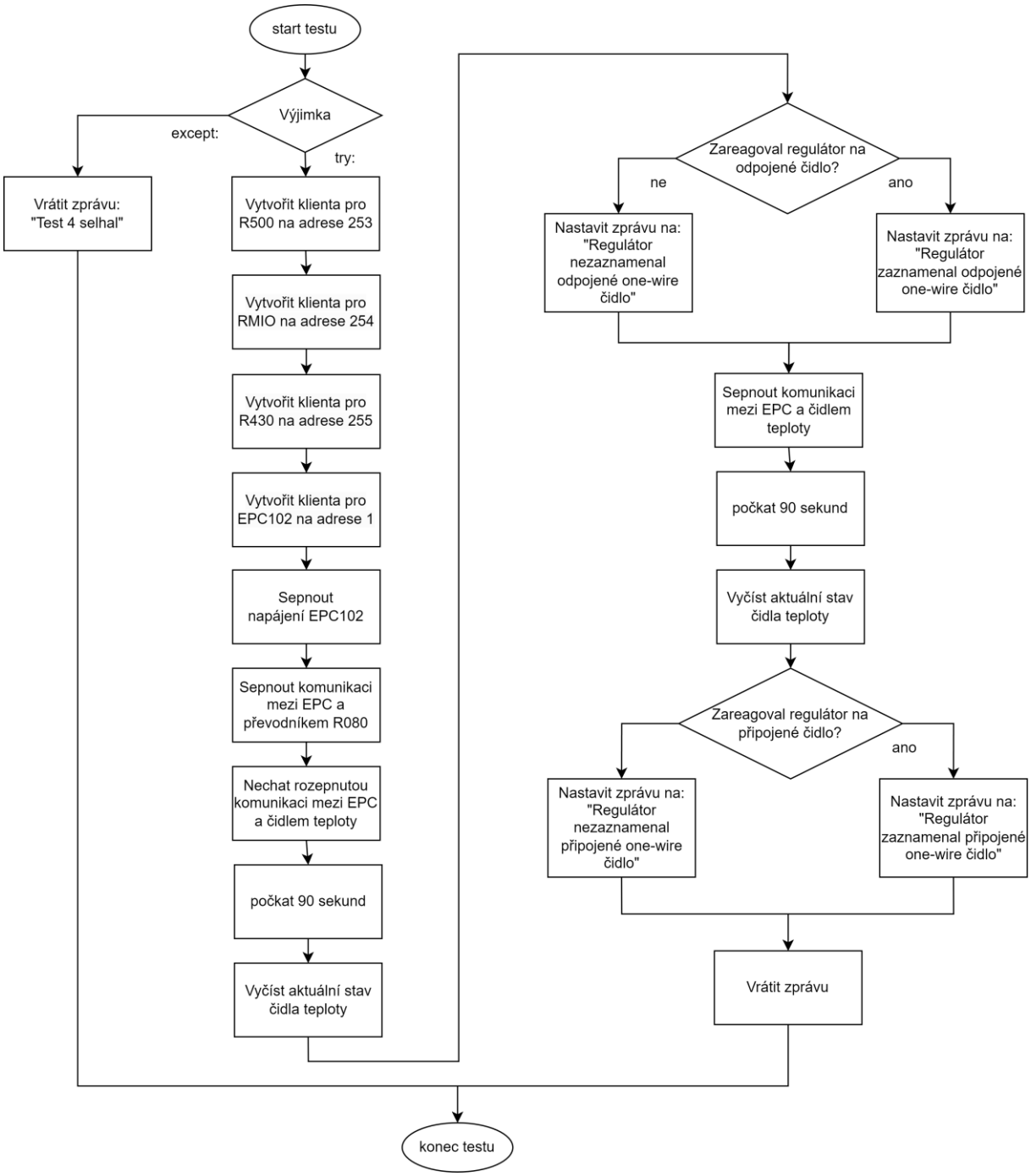
Obr. 2.37 – Vývojový diagram testu: Kontrola správně vybraného zařízení



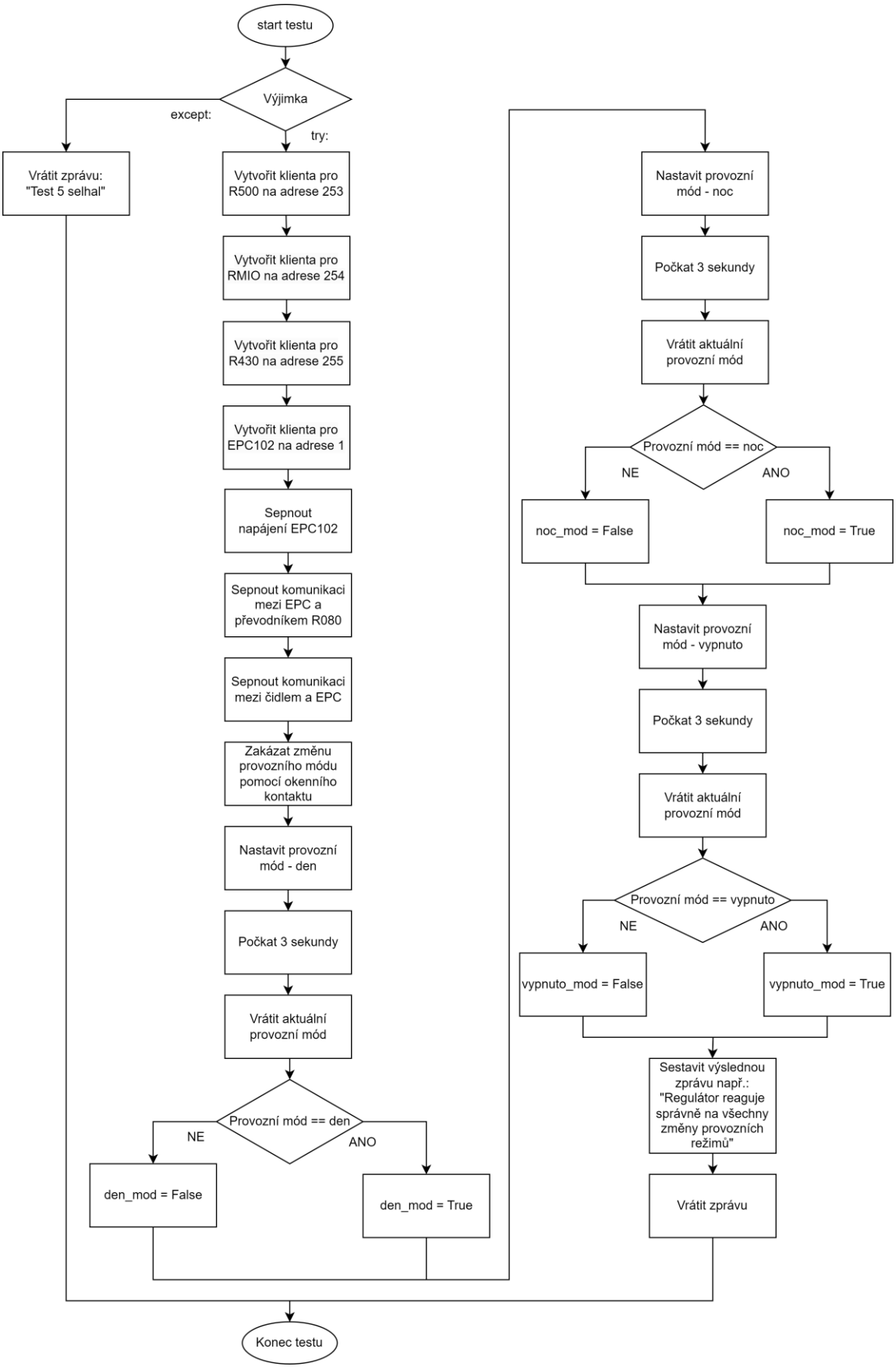
Obr. 2.38 – Vývojový diagram testu: Test Modbusových adres – SW reset



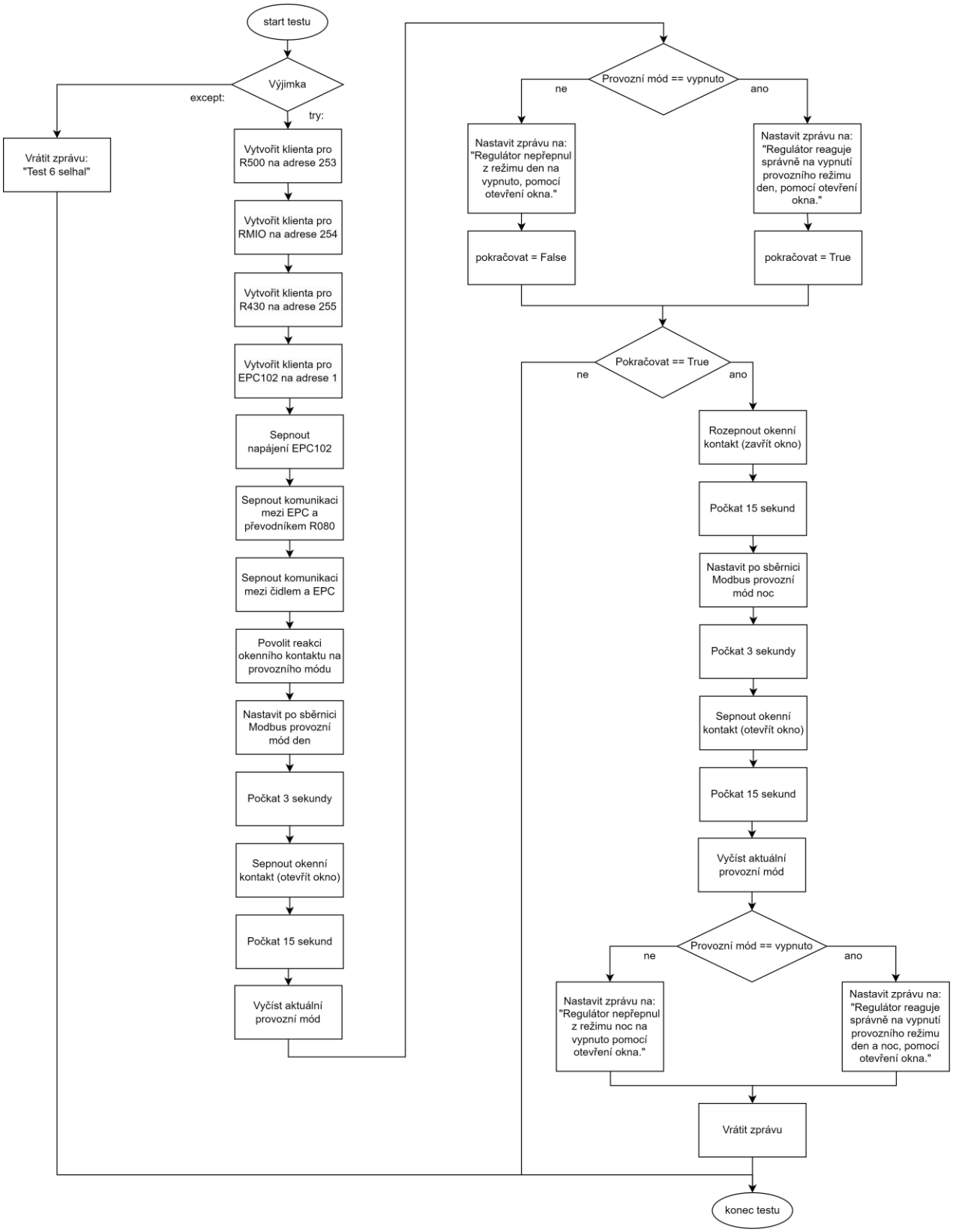
Obr. 2.39 – Vývojový diagram testu: Test okenního kontaktu



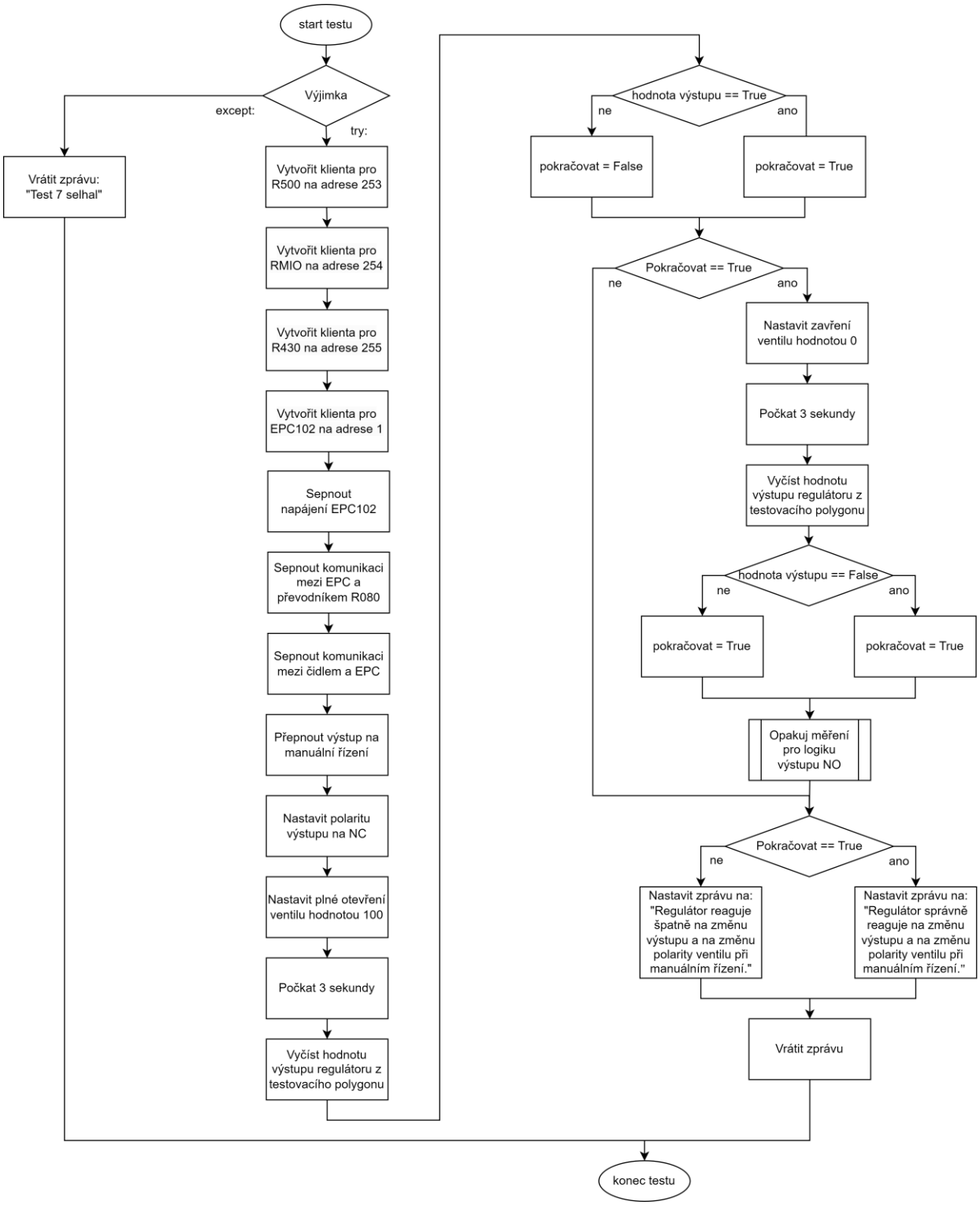
Obr. 2.40 – Vývojový diagram testu: Přerušené 1-Wire čidlo



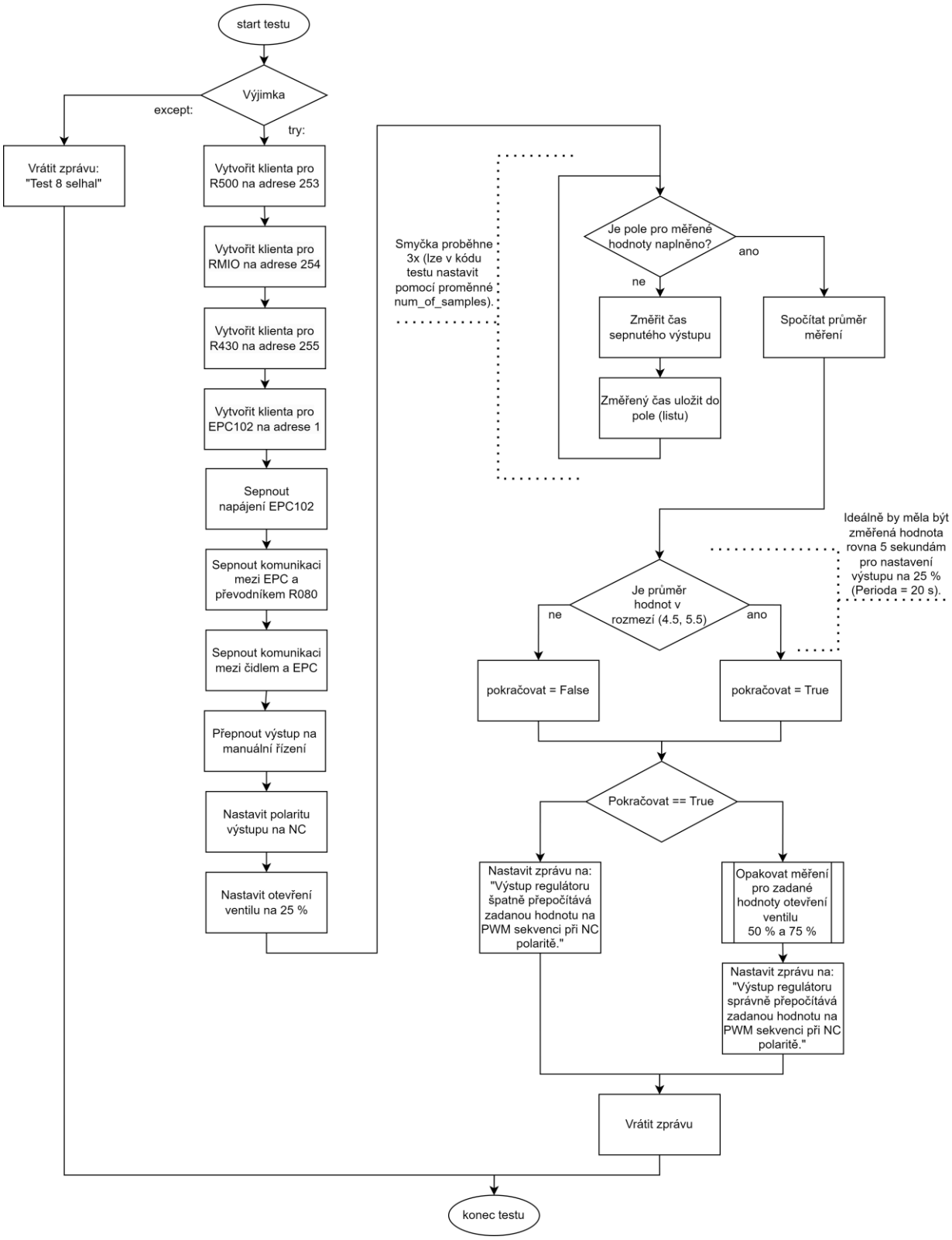
Obr. 2.41 – Vývojový diagram testu: Provozní módy – Modbus



Obr. 2.42 – Vývojový diagram testu: Provozní módy – okenní kontakt



Obr. 2.43 – Vývojový diagram testu: Manuální sepnutí výstupu



Obr. 2.44 – Vývojový diagram testu: Manuální sepnutí výstupu – PWM, polarita NC

ZÁVĚR

Závěrem této diplomové práce lze konstatovat, že vytvořený software pro automatizované testování firmware přináší významné výhody v oblasti efektivity a finanční úspory ve vývojové fázi návrhu zařízení. Aplikace umožňuje automatické provedení testů dle definovaných scénářů, což vede k eliminaci manuálních úkonů a významnému zkrácení času potřebného k jejich provádění. Automatické generování výsledků testů a jejich ukládání do SQL databáze přispívá k rychlé a efektivní analýze výsledků.

Příkladem úspory času je provedení testu ověřujícího správnost Modbusových adres zařízení. Manuální kontrola by vyžadovala značné množství času a práce, zatímco prostřednictvím navržené aplikace je tento proces zautomatizován a trvá pouze přibližně 8,75 minuty. Tímto způsobem se nejen ušetří čas operátora, ale také se minimalizuje riziko lidských chyb a zvyšuje se spolehlivost testovacího procesu. Výsledkem je efektivnější a kvalitnější vývoj firmware, což přispívá k celkové konkurenceschopnosti a úspěchu společnosti.

Ukázkové testy, které byly pro zařízení definovány skončily s pozitivním výsledkem, což je především důsledkem kvality samotného zařízení, které bylo testováno – regulátoru vyvinutého s důrazem na produkční standardy. Tyto testy pokrývají základní logiku regulátoru a přispívají k ověření jeho správné funkce.

Pro další zvýšení kvality testování je možné rozšířit aplikaci a databázi o systém kombinací jednotlivých testových scénářů. Tím by se dosáhlo vyšší robustnosti testování zařízení. Například by bylo možné prověřit Modbusové adresy za různých nastavení Modbusového klienta, čímž by se ještě komplexněji otestovala funkčnost a spolehlivost zařízení při různé konfiguraci.

I když je práce hodně zaměřena na modularitu a snadné rozšíření v budoucnu. Bylo by možné ještě tuto problematiku posunout o úroveň výše. Konkrétně samotné skripty, které jsou aplikací spouštěny by bylo dobré od kódu aplikace oddělit.

Tento krok by zahrnoval uložení skriptů do samotné složky na FTP serveru, a při každém spuštění aplikace by se vytvořila lokální kopie těchto skriptů na PC operátora testu, obdobně jako je tomu např. u schémat zařízení.

POUŽITÁ LITERATURA

Black box vs white box testing, 2024. In: Practitest [online]. [cit. 2024-05-21]. Dostupné z: https://www.practitest.com/resource-center/article/black-box-vs-white-box-testing/

BOHUSLAV, Daniel, 2020. Vývoj aplikace očima zákazníka. In: Memos software [online]. [cit. 2024-04-13]. Dostupné z: https://www.memos.cz/vyvoj-aplikace-ocima-zakaznika/

Dokumentace JSON [online], 2023. Sydney [cit. 5. 9. 2024]. Dostupné z: https://docs.fileformat.com/cs/web/json/

Domat Control System s.r.o., 2017. EPC102 Regulátor topení, komunikativní [PDF]. Pardubice, 4 s. Dostupné z: https://www.domat-int.com/cs/katalogove-listy

Domat Control System s.r.o., 2017. R430 Modul 32 digitálních vstupů [PDF]. Pardubice, 6 s. Dostupné z: https://www.domat-int.com/cs/katalogove-listy

Domat Control System s.r.o., 2018. RMIO Kompaktní I/O modul [PDF]. Pardubice 7 s. Dostupné z: https://www.domat-int.com/cs/katalogove-listy

Domat Control System s.r.o., 2019. R500 Modul analogových vstupů [PDF]. Pardubice, 6 s. Dostupné z: https://www.domat-int.com/cs/katalogove-listy

Domat Control System s.r.o., 2020. R080 Převodník USB–RS485 [PDF]. Pardubice, 4 s. Dostupné z: https://www.domat-int.com/cs/katalogove-listy

HARTINGER, David, 2021. Lekce 2 - Singleton (Jedináček). In: Itnetwork.cz [online]. [cit. 10. 5. 2024]. Dostupné z: https://www.itnetwork.cz/navrh/navrhove-vzory/gof/singleton-navrhovy-vzor

HARTINGER, David, 2022. Úvod do práce se soubory [online]. [cit. 9. 5. 2024]. Dostupné z: https://www.itnetwork.cz/csharp/soubory-a-sit/c-sharp-tutorial-uvod-do-prace-se-soubory

CHERNYAK, Alex Zap, 2024. Statické testování v testování softwaru – co to je, typy, proces, přístupy, nástroje a další!. ZPTEST unlimited software automation [online]. [cit. 2024-05-14]. Dostupné z: https://www.zaptest.com/cs/staticke-testovani-v-testovani-softwaru-co-to-je-typy-proces-pristupy-nastroje-a-dalsi

JANEBA, Mark, 2011. The Pentium problem. In: DevTopics [online]. 2011 [cit. 2024-04-12]. Dostupné z: https://willamette.edu/~mjaneba/pentprob.html

KOMUNIKAČNÍ PROTOKOL MODBUS, 2021. In: Domat člen ČEZ ESCO [online]. [cit. 2024-05-14]. Dostupné z: https://www.domat-int.com/cs/komunikacni-protokol-modbus

PATTON, Ron, 2002. Testování softwaru. 2002. Praha: Computer Press. Programování. ISBN 80-722-6636-5.

PECINOVSKÝ, Rudolf, 2020. Python: kompletní příručka jazyka pro verzi 3.9. Praha: Grada Publishing. Knihovna programátora (Grada). ISBN 978-80-271-1269-2.

PECINOVSKÝ, Rudolf, 2020. Začínáme programovat v jazyku Python. Praha: Grada Publishing. Začínáme s.. ISBN 978-80-271-1237-1.

RUSCICA, Tim, 2019. PyQt5 Python 3 Tutorial. Youtube.com [online]. [cit. 11. 4. 2024]. Dostupné z: https://www.youtube.com/playlist?list=PLzMcBGfZo4-lB8MZfHPLTEHO9zJDDLpYj

RUTLEDGE, Kim a Melissa MCDANIEL, 2022. Y2K bug. National Geographic [online]. [cit. 2024-04-13]. Dostupné z: https://education.nationalgeographic.org/resource/Y2K-bug/

PŘÍLOHY

A – CD

Příloha A

Příloha k diplomové práci

Software pro automatizované testování firmware prvků průmyslové automatizace

Bc. Martin Poledno

CD

OBSAH