FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název: Automatické testování infotainment jednotek
Student: Bc. Jan Kubát
Vedoucí: Ing. Martin Daňhel, Ph.D.
Studijní program: Informatika
Studijní obor: Návrh a programování vestavných systémů
Katedra: Katedra číslicového návrhu
Platnost zadání: Do konce letního semestru 2020/21

Pokyny pro vypracování

Cílem práce je vytvořit sadu testů pro prototyp průmyslového testovacího zařízení (tzv. testovací stav), kterým se testuje grafika infotainment jednotek automobilů. Stav obsahuje kromě jednotky také další komponenty, které s ní komunikují prostřednictvím CAN sběrnic. Testovací prostředí navíc dokáže simulovat přítomnost ostatních jednotek vozidla, aby infotainment zobrazoval korektní a úplné údaje.

Proveďte:
1) Seznamte se s dodaným rozhraním infotainment jednotky a testovacím zařízením (stavem).
2) Popište vývojový cyklus grafického rozhraní infotainment jednotky v automobilu a vysvětlete důležitost automatického testování.
3) Zdokumentujte prototyp testovacího stavu.
4) Seznamte se programovacím jazykem TCL, který zde slouží pro psaní automatických testů.
5) Rozšiřte stávající knihovnu funkcí v jazyce TCL a s její pomocí vytvořte sadu testů pro automatické testování grafiky infotainmentu.

Seznam odborné literatury

Dodá vedoucí práce.

doc. Ing. Hana Kubátová, CSc.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 27. ledna 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Automatické testování infotainment jednotek

Bc. Jan Kubát

Katedra číslicového návrhu
Vedoucí práce: Ing. Martin Daňhel, Ph.D.

29. července 2020

Poděkování

Rád bych touto cestou poděkoval Ing. Martinovi Daňhelovi, Ph.D. za aktivitu a ochotu při vedení mé práce, za nabídku spolupráce nejen na diplomové práci, ale i za poskytnutý kontakt, který mi otevřel dveře do pracovního sektoru.

Dále můj dík patří firmě Digiteq Automotive s.r.o., která mi dala příležitost na tomto zajímavém tématu pracovat a využít všechen potřebný hardware a software. Konkrétně bych chtěl poděkovat celému oddělení DQFG pod vedením Bc. Martina Šamana a projektové vedoucí Ing. Lucie Hradecké za vlídné přijetí do kolektivu a vytvoření nadstandardních pracovních podmínek.

V neposlední řadě patří dík Ing. Tomášovi Zimmerhaklovi, který mi domluvil účast na tomto projektu, následně se mnou úzce spolupracoval a tím mi velmi ulehčil a zpříjemnil přechod ze studentského do pracovního života.

Moc děkuji i mé přítelkyni Bc. Lucii Trnkové za její trpělivost a pomoc při opravě a korekci již napsaného textu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu) licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 29. července 2020 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2020 Jan Kubát. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Kubát, Jan. *Automatické testování infotainment jednotek*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2020.

Abstrakt

Tato diplomová práce se zabývá automatickým testováním infotainment jednotky automobilu. Práce podává teoretické informace o skriptovacím jazyce TCL a sběrnici CAN, které představují základ pro vytvoření automatických testů. Seznámení s procesem vývoje grafického rozhraní doprovází motivace pro jeho zlepšení. Na základě představené motivace je navržen princip a zapojení automatického testování do vývojového procesu grafického rozhraní. Součástí práce je dokumentace testovacího stavu, jež je provedena z hlediska použitého hardware i software. Pomocí jazyka TCL jsou vytvořeny testovací funkce a sady testů, které slouží k odhalení chyb grafického rozhraní. Práce popisuje celý testovací proces i s reportem výsledků.

Klíčová slova TCL, CAN, automatické testování, infotainment, GUI, HMI

Abstract

This diploma thesis deals with automated testing of the infotainment unit of a car. The thesis provides theoretical information about the scripting language TCL and the CAN bus, which are the basis for creating automated tests. Introduction to the process of developing a graphical interface is accompanied by motivation for improvement. Based on the introduced motivation, the principle and involvement of automated testing in the development process of the graphical interface is designed. The thesis contains the documentation of the test rack, which is performed in terms of used hardware and software. The TCL language is used to create test functions and test cases to detect errors of the grafical interface. This thesis describes the whole testing process with the report of results.

Keywords TCL, CAN, automated testing, infotainment, GUI, HMI

Obsah

Úvod **1**

1 Cíl práce **3**

2 Analýza a návrh **5**
2.1 Jazyk TCL 5
2.1.1 Popis jazyka 5
2.1.2 Syntaxe 8
2.1.3 Strukturované datové typy 21
2.1.4 Práce s řetězci 24
2.1.5 Práce se soubory 26
2.2 Sběrnice CAN 28
2.2.1 Důvody zavedení 29
2.2.2 Základní vlastnosti 29
2.2.3 Fyzická realizace 30
2.2.4 Řízení přístupu a řešení kolizí 32
2.2.5 Zabezpečení přenášených dat 32
2.2.6 Signalizace chyb 33
2.2.7 Základní typy zpráv 34
2.3 Vývojový cyklus grafického rozhraní 37
2.3.1 Životní cyklus obrazovky 37
2.4 Motivace 40
2.5 Zapojení automatického testování do vývoje GUI 42
2.5.1 Princip automatického testování 42
2.6 Testovací stav 45
2.6.1 Hardware 46
2.6.2 Software 55

3 Realizace **57**
3.1 Tracing 58

3.1.1 Python 58
3.2 Automatické testy 60
3.2.1 Kód 60
3.2.2 Spouštění testů 65
3.2.3 Nastavování obrazovek 68
3.2.4 Focení a vyhodnocení 76
3.2.5 Report 81

4 Testování **85**
4.1 Princip 85
4.2 Problémy 85
4.2.1 Úsporný režim 86
4.2.2 Navození obrazovek 86
4.2.3 Uchování testcases ve slovníku 86
4.2.4 Chybějící obrazový podklad 86
4.2.5 Neúspěšné klikání 87
4.2.6 Scrollování 87
4.2.7 Překryv drop-down 88
4.2.8 Generování reportu 89

Závěr **91**

Literatura **93**

A Seznam použitých zkratek **97**

B Obsah přiloženého CD **99**

Seznam obrázků

2.1 Fyzická realizace CAN . . . 30
2.2 Kroucená dvojlinka CAN sběrnice . . . 31
2.3 Zapojení sběrnice CAN s konektorem D-SUB . . . 31
2.4 Datová zpráva dle specifikace CAN 2.0A . . . 34
2.5 Začátek datové zprávy standardního formátu podle 2.0B . . . 35
2.6 Začátek datové zprávy rozšířeného formátu podle 2.0B . . . 36
2.7 Životní cyklus obrazovky . . . 38
2.8 Dokončená obrazovka před aktualizací modelu . . . 41
2.9 Dokončená obrazovka po aktualizaci modelu . . . 41
2.10 Životní cyklus obrazovky s automatickými testy . . . 43
2.11 Testovací stav pro automatické testování . . . 46
2.12 Windows PC . . . 47
2.13 Black Box PC . . . 48
2.14 Zdroj ZUP2020 (ilustrační obrázek) . . . 48
2.15 Zdroj bez možnosti dálkového ovládání . . . 49
2.16 Switch lokální sítě stavu . . . 49
2.17 Switch směrující zprávy na CAN/LAN . . . 50
2.18 Jednotka infotainmentu (ilustrační obrázek) . . . 51
2.19 Gateway (ilustrační obrázek) . . . 51
2.20 Grabber . . . 52
2.21 Displej infotainmentu - platfroma MQB, 9″ . . . 52
2.22 CAN case . . . 53
2.23 Schéma zapojení hardwaru stavu ke zdroji el. energie . . . 53
2.24 Schéma datového propojení testovacího stavu . . . 54

3.1 Testovací stav s 10″ displejem platformy MQB_37W . . . 57
3.2 Ovládání jednotky se zpětnou vazbou . . . 59
3.3 10″ displej platformy MQB_37W . . . 60
3.4 Schéma provázanosti zdrojových souborů . . . 61
3.5 Program TestAut pro ovládání běhu testů . . . 65

3.6 Příklad obrazovky bez simulace (nastavení jednotek systému) . . . 66
3.7 Příklad obrazovky se simulací (nastavení jednotek systému) 66
3.8 Nenastavená obrazovka s přepínači 69
3.9 Nastavená obrazovka s přepínači 69
3.10 Ikona drop-down 70
3.11 Obrázek vytvořený podle zadaného textu 70
3.12 Scrollovací obrazovka v počáteční pozici 71
3.13 Jednou posunutá scrollovací obrazovka 72
3.14 Vybraný přepínač 72
3.15 Nevybraný přepínač 72
3.16 Slider nastavený na maximum 73
3.17 Slider nastavený na minimum 73
3.18 Slider pro nastavení světlometů 73
3.19 Nastavení ekvalizéru 74
3.20 Obrazovka s aktivním drop-down 74
3.21 Odebírání widgetů z domovské obrazovky 75
3.22 Prázdná domovská obrazovka 75
3.23 Domovská obrazovka s widgety o jízdních datech 76
3.24 Celý displej 76
3.25 Celý displej s vyznačenými ořezovými oblastmi 77
3.26 Ořezová oblast `WIDE_SCREEN` 78
3.27 Ukázka výstupu módu `tabbar` 79
3.28 Referenční obrázek 80
3.29 Nový obrázek 80
3.30 Rozdílový obrázek 80
3.31 Adresářová struktura reportu 81
3.32 Ukázka jednotlivých řádků reportu 82
3.33 Ukázka konečného reportu 83

4.1 Databázové předlohy pro switch 87
4.2 Ukázka překryvu obrazovky elementem drop-down 88

Seznam tabulek

2.1 Významy speciálních znaků při substituci v TCL 11
2.2 Matematické funkce v TCL 12
2.3 Logické, bitové a relační operace v TCL 13
2.4 Přepínače příkazu switch 18
2.5 Funkce dict 19
2.6 Příkaz array pro práci s polem 22
2.7 Funkce pro práci se seznamem 23
2.8 Znaky regulárních výrazů prvního typu 25
2.9 Znaky regulárních výrazů druhého typu 25
2.10 Funkce pro práci s řetězci 26
2.11 Funkce string 26
2.12 Práce s daty souborů 27
2.13 Příkaz file pro práci se soubory 28

3.1 Zdrojové soubory 62
3.2 Atributy testcase 64
3.3 Módy pořizování snímků 78

Úvod

Moderní automobily obsahují velké množství systémů a funkcionalit. Dnes je samozřejmostí, že většinu informací o takových systémech a ovládání jejich funkcionalit má uživatel k dispozici. Rozhraní mezi uživatelem a automobilem se nazývá infotainment - spojení dvou anglických slov information (informace) a entertainment (zábava). Infotainment zobrazuje uživateli na displej informace o stavu vozidla a jeho systémech. Umožňuje uživateli nastavit velké množství systémů od jízdních vlastností automobilu, přes komfortní systémy, jako je klimatizace, až po systémy zábavy, kterým je například poslech hudby.

Všechny systémy automobilu jsou samozřejmě testovány z hlediska jejich funkčnosti. Nutné je ovšem také testování HMI[1], tedy přímého rozhraní mezi uživatelem a systémem infotainmentu. Chyba na obrazovce infotainmentu by nejen ovlivnila zážitek uživatele z interakce s vozem, ale mohla by také znemožnit nastavení některého ze systémů automobilu. S rostoucím počtem funkcí automobilu, například i v propojení s IoT[2], roste i počet obrazovek, které se uživateli mohou zobrazit. S tím roste také počet testů, jenž musí být provedeny. V moderních systémech infotainmentu jsou takových obrazovek stovky, což znamená velké množství práce při jejich opakovaném testování.

Jednotka infotainmentu, jež tento systém řídí, zobrazuje na displej obraz konkrétních obrazovek a reaguje na podněty uživatele. Ty mají podobu nejčastěji doteků na konkrétních souřadnicích displeje. Pokud to jednotka umožňuje, může být podnět uživatele v podobě stisknutí tlačítka či hlasového příkazu. Ať už jde o jakýkoli podnět uživatele, příslušná jednotka, která ho zaznamená, odešle zprávu o podnětu do jednotky infotainmentu. Ta na tento podnět následně adekvátně reaguje. Komunikace mezi jednotkami přítomnými ve voze probíhá po konkrétních sběrnicích, nejčastěji se jedná o sběrnice CAN[3].

[1]Human Mahine Interface - rozhraní mezi uživatelem a přístrojem.
[2]Internet of Things - internet věcí, propojení elektronických přístrojů přes internet.
[3]Controller Area Network - komunikační protokol využívaný v automobilech.

Znalost protokolu CAN, jednotlivých CAN sběrnic a konkrétních podob CAN zpráv lze využít pro automatizaci testování. Vývojář má možnost poslat jednotce infotainmentu zprávu o podnětu, který fyzicky nenastal, avšak jednotka na tento podnět zareaguje stejně, jakoby se stalo.

Automobily vždy tvořily velkou oblast mého zájmu a to nejen z hlediska jízdních vlastností. Chtěl jsem také poznat, jak fungují jejich elektronické komponenty a jakým způsobem probíhá vývoj jednotlivých součástí celého automobilu. Téma této diplomové práce mi dalo příležitost se touto oblastí zabývat a podílet se na vývoji grafického rozhraní infotainment jednotky.

Tato diplomová práce se zabývá principem automatizace testování HMI jednotky infotainmentu a vytvořením sady testů pro její automatické testování. Obsahuje šest důležitých kapitol včetně úvodu a závěru. Úvod následuje kapitola *Cíl práce* představující cíle stanovené na začátku tvorby práce. Následují kapitoly *Analýza a návrh*, *Realizace* a *Testování*. První z těchto kapitol popisuje skriptovací jazyk TCL[4] a komunikační protokol CAN. Dále uvádí vývojový cyklus grafického rozhraní a popis testovacího stavu. V této kapitole je také uvedena motivace použití automatických testů ve vývoji HMI a princip testování. Kapitola *Realizace* obsahuje význam využití trace jednotky infotainmentu a popis vytvořených zdrojových souborů, testovacích funkcí a testů. Následně se zabývá výsledky testů a vytvořením reportu. Následující kapitola s názvem *Testování* popisuje princip testování funkčnosti vytvářených funkcí a testcases. Obsahuje také seznámení s problémy, které nastaly v průběhu vývoje a s jejich řešením.

[4]Tool Command Language - skriptovací programovací jazyk.

Kapitola 1

Cíl práce

První z dílčích cílů práce představuje seznámení se skriptovacím jazykem TCL a sběrnicí CAN. Abych byl schopen vytvořit sadu testovacích funkcí a testů, musím nejdříve prozkoumat možnosti jazyka TCL, ve kterém budou funkce a testy vytvářeny. Stejně tak musím porozumět komunikaci jednotek automobilu, jež se uskutečňuje pomocí protokolu CAN. Tyto nabyté znalosti dají základ budoucímu vytvoření testů.

Dalším cílem je seznámení s vývojovým procesem grafického rozhraní a motivací, jak takový proces vylepšit. Aby bylo možné efektivně navrhnout a využít automatické testy, je třeba prozkoumat stávající proces vývoje grafického rozhraní a důvody, proč tento proces vylepšit. Navazující cíl na popis procesu vývoje grafického rozhraní a motivace jeho zlepšení, představuje návrh automatizace testování a jejího zapojení do vývoje grafického rozhraní.

Dílčím cílem je i dokumentace testovacího stavu sloužícího pro ovládání a spouštění automatických testů. Seznámení se s hardwarem i softrwarem testovacího stavu mi umožní využít jeho funkčnosti pro testování v plném rozsahu.

Hlavním cílem je využít všechny znalosti nabyté v analytické části práce a vytvořit sadu testovacích funkcí a testů, které odhalí chyby grafického rozhraní infotainment jednotky.

Kapitola 2

Analýza a návrh

Tato kapitola obsahuje analytickou část diplomové práce. Jsou v ní zpracované teoretické informace o skriptovacím jazyce TCL a sběrnici CAN. Dále popisuje vývoj grafického rozhraní infotainmentu a uvádí motivaci pro jeho zlepšení. Na základě této motivace navrhuje princip automatických testů a jeho zapojení do procesu vývoje grafického rozhraní. Tato část práce také dokumentuje testovací stav pro ovládání automatických testů.

2.1 Jazyk TCL

Úvodním tématem, jenž je důležitým základem pro vytvoření testovacích funkcí a samotných testů je seznámení s jazykem TCL. Následující sekce obsahují popis všech důležitých vlastností a struktur tohoto jazyka.

2.1.1 Popis jazyka

Tool Command Language je interpretovaný programovací jazyk, jenž je specializován na snadnou a rychlou tvorbu aplikací nebo grafických uživatelských rozhraní (GUI[5]), pro které se užívá nadstavba **Tk**. TCL se také používá pro zabudovaní do dalších aplikací, kterým dává možnost jednoduchého a efektivního skriptování. Mimo to, lze také přímo z programu napsaného v jazyce TCL volat jiné programy a utility, které tak mohou vlastní TCL rozšiřovat o další funkce.

TCL ideově a konceptuálně vychází z jazyka Lisp, ale zavádí určitá pravidla, která uživateli zjednodušují práci a na rozdíl od Lispu (a Perlu) je tak kód TCL snáze čitelný a pochopitelný. [1]

[5] Graphical User Interface - grafické uživatelské rozhraní.

Vlastnosti

Jak již bylo zmíněno, TCL je jazykem interpretovaným a tohoto aspektu využívá (podobně jako svůj předchůdce Lisp) co nejvíce. Příkladem je fakt, že samotný jazyk neobsahuje žádné příkazy pro tvorbu smyček, podmínek nebo řešení matematických a logických výrazů. Bez těchto základních kamenů programování si snad nikdo nemůže představit funkční programovací jazyk. V TCL jsou tyto elementy řešeny funkcí, které jako své parametry mohou přijímat programový kód (jenž pak například tvoří tělo smyčky nebo výraz určující podmínku).

Podobně jako u dalších interpretovaných jazyků se i zde setkáme se značně jednoduchou syntaxí. Ta však může v určitých případech být jaksi nepohodlná, například při zapisování matematických výrazů či vyhodnocování proměnných.

Jednou z nejzajímavějších vlastností TCL je fakt, že v podstatě vše se v průběhu programu vyhodnocuje jako textový řetězec, na který se případně číselné hodnoty převádějí - čísla jsou tedy z uživatelského pohledu ukládána jako řetězce. Za řetězec je vlastně považován i vlastní zápis programu a je tedy možné ho dynamicky, takže přímo za běhu programu, upravovat. Tento fakt opět připomíná jeho předchůdce - Lisp. Fakt, že všechny proměnné jsou považovány za řetězec, dělá z TCL beztypový jazyk, tedy u proměnných, parametrů nebo i návratových hodnot není třeba uvádět specifický datový typ. Je to však pochopitelně také zdroj nepříjemností spojených s jazykem, může totiž nastat případ (například při zpracování matematických výrazů), kdy dojde vinou špatné konverze k nesprávné interpretaci parametrů.

Přestože se jedná o interpretovaný jazyk, má TCL k dispozici také překladač. Ten však samozřejmě nedokáže převést program do spustitelného tvaru, právě kvůli dynamické změně programu při běhu. Překlad se nejprve provádí do bytekódu, který je snadněji a rychleji zpracovatelný než původní zdrojový soubor. Překlad do strojového kódu se provádí až přímo při spuštění nebo změně programu. Tomuto překladu se říká *just-in-time* překlad. [1]

Výhody a nevýhody

Mezi bezesporné výhody jazyka TCL patří jednoduchá tvorba aplikací, snadné ladění a také návaznost na další programovací jazyky jako například C, C++ nebo Java. Spojení dynamického TCL a kupříkladu rychlého C je určitě výhodnou kombinací. Jednou z variant je mít program napsaný v TCL a pro složitější nebo časově náročnější části použít jiný programovací jazyk. Nebo naopak lze aplikace vytvořit v jiném jazyce a TCL použít jako skriptovací nástroj či pro práci s grafickým prostředím.

Další velkou výhodou TCL je jeho portabilita. Jak jednotlivé knihovny a napsané programy, tak i vývojové prostředí lze jednoduše portovat na různé operační systémy, mezi které samozřejmě patří Linux, MacOS i Windows. Stejně tak je na těchto systémech dostupný i interpret jazyka.

Výhodou může být i podpora kódování řetězců v Unicode[6], která je součástí TCL od verze 8.1. Do této verze bylo podporováno kódování jen v ASCII[7] a tak představovalo problém zejména v Evropě, kde se běžně používá velké množství speciálních znaků. V době tvorby tohoto textu je aktuální verzí jazyka verze 8.6.10.[2]

Největší nevýhodou TCL je nižší rychlost provádění skriptů, zejména v porovnání s kompilovanými jazyky. Užívá se však většinou v případech, kdy rychlost běhu programu není tak kritická a pokud ano, vždy se dají konkrétní funkce přesunout do kompilovaného jazyka.

K dalším záporům patří omezení jazyka, především při tvorbě objektově orientovaných programů. Opět se při potřebě OOP[8] dá přikročit k C funkcím, návrh jazyka však s objektově orientovaným programováním nepočítal. [1]

Porovnání

TCL bývá porovnáván s dalšími programovacími jazyky, jimiž jsou Perl, Lisp nebo Basic. Další porovnání se nabízejí i s velmi rozšířeným jazykem C. S těmito jazyky je TCL porovnáván i v následujících sekcích.

Nejčastěji je TCL srovnáván s dalším skriptovacím jazykem jménem Perl. Ten je v mnoha směrech velmi jednoduchým jazykem a není třeba znát mnoho, aby bylo možné v něm psát i poměrně složité programy. Na druhou stranu má i své stinné stránky, jako fakt, že neexistuje pevně definovaná syntaxe pro jednu konkrétní úlohu.[3]

Vzhledem k TCL, vítězí Perl s ohledem na podporu regulárních výrazů (v tomto ohledu je snad nepřekonatelný) a také při porovnání rychlosti běhu programu. Na druhou stranu je TCL jednodušší, jeho kód je čitelnější a skýtá mnohem více možností na zakomponování do dalších aplikací. [1]

[6]Technická norma kódování znaků všech abeced.

[7]American Standard Code for Information Interchange - technická norma kódování anglické abecedy.

[8]Object-oriented programming - objektově orientované programování.

2.1.2 Syntaxe

Poznámky

Úvodem bylo popsáno, že v jazyce TCL se všechna slova považují za řetězce, přičemž některé z nich představují příkazy a další představují parametry příkazů. Příkazy samotné jsou od sebe odděleny novým řádkem (newline = \n) nebo středníkem (semicolon = ;). Jinak tomu není ani u poznámek, které začínají specifickým řetězcem - nejedná se tedy o žádnou specializovanou jazykovou konstrukci (s výjimkou, že se v textu poznámky neprovádějí substituce příkazů ani proměnných).

Poznámka v jazyce TCL začíná řetězcem o jednom znaku - křížku (hash = #). Tuto vlastnost sdílí TCL například s jazykem Perl nebo se skripty psanými pro shell. Spuštění slova # tedy znamená, že se všechny následující znaky do konce řádku přečtou, ale ignorují. Ve skutečnosti se tak děje již při načítání programu lexikálním analyzátorem, což přináší rychlejší běh programu. [4]

```
1 # Tohle je poznámka v TCL
```

Při psaní poznámek v TCL je však třeba si dát pozor na onu skutečnost, že znak # je považován za příkaz. Tento znak se tedy musí nacházet na prvním místě řádku. Pokud bychom chtěli napsat poznámku na řádek s jiným příkazem, musí být poznámka, jako každý jiný příkaz navíc, oddělena středníkem od příkazu původního. Mohla by například vypadat takto:[4]

```
1 puts example ;# Příkaz vypíše řetězec "example"
```

Naopak špatně napsaná poznámka, přestože na první pohled vypadá správně, je tato:[4]

```
1 puts example # Příkaz vypíše řetězec "example"
```

Zde si můžeme všimnout, že prostředí vypisující kód v LaTeX správně interpretovalo oba předchozí příklady a při bezchybně napsané poznámce ji vysázelo do dokumentu zelenou barvou, kdežto špatně napsaná poznámka je vysázena klasicky.

Při použití křížku je vždy nutné se omezit na jednořádkový komentář. Pokud je třeba psát komentář víceřádkový, je nutné na začátku každého řádku vždy napsat křížek nebo použít nesplnitelnou podmínku. V druhém případě však vývojové prostředí nebude kód zvýrazňovat jako poznámku.[5]

```
1 if 0 {
2   Víceřádková poznámka
3   pomocí nesplnitelné podmínky
4 }
5 puts "Ahoj světe"
```

Proměnné

S proměnnými se v TCL pracuje velmi jednoduše. Je třeba dodržet pouze jedno pravidlo při práci s nimi a to vždy proměnnou inicializovat před jejím použitím. Tímto se liší od Perlu či Basicu, kde je povoleno použití neinicializované proměnné nebo od C a jemu podobných jazyků, kde se proměnné inicializují na nějakou implicitní hodnotu. Inicializace proměnné se provádí pomocí příkazu **set**, který následuje jméno proměnné a její hodnota. Hodnotu může představovat buď konstanta nebo i jiný příkaz, který se vyhodnotí.[4]

```
1 # Příklady nastavování proměnných
2
3 set num 23
4 set var Ahoj
5 set longVar "Ahoj světe"
```

V předchozí ukázce jsou uvedeny příklady inicializací proměnných. Proměnná **num** obsahuje hodnotu 23, proměnná **var** pak řetězec "Ahoj" a nakonec proměnná **longVar** obsahuje řetězec "Ahoj světe". Uvozovky nejsou součástí obsahu **var** ani **longVar**. Hodnoty všech proměnných jsou fakticky považovány za řetězec - žádný jiný typ TCL ani nezná. Uvozovky v zápise hodnoty proměnných nesmí chybět v případě, že hodnota obsahuje mezery. Pokud tomu tak není, přiřazení může být bez uvozovek.

Pokud se příkaz **set** napíše pouze s názvem proměnné, vrátí hodnotu oné proměnné.

Příkaz **unset** se stará o zrušení proměnné. I jemu se jako parametr předá název proměnné. Tento příkaz také uvolní paměť, na níž byla proměnná uložena, což v případě velkých polí či seznamů může být důležitá operace.[4]

```
1 # Příklad rušení proměnné
2
3 unset num
```

Hodnotu proměnné si lze zobrazit (vytisknout na standardní výstup) pomocí příkazu **puts**, jenž vypisuje své parametry právě na standardní výstup. Příkaz **set** hodnotu vypíše pouze při práci v interaktivním prostředí. U příkazu **puts** se poprvé setkáváme se substitucí - nahrazením pouhého jména proměnné její hodnotou. To je docíleno použitím znaku dolaru (dollar sign = $), jenž se vkládá před jméno proměnné, za které se v programu dosadí její hodnota.[4]

```
1 # Příklad výpisu
2
3 puts var
4 puts $var
```

Výstup předchozího příkladu by při takto zadefinovaných proměnných vypadal následovně:

```
1 var
2 Ahoj
```

Identifikátor neboli jméno proměnné či funkce může v TCL začínat na jakékoli písmeno (velké či malé) nebo na podtržítko (underscore = _). Následovat může žádný nebo více znaků ze skupin písmen, čísel, podtržítek nebo znaků dolaru. Nejsou povoleny znaky jako zavináč (at = @) nebo procento (percent = %). TCL je tzv. case sensitive jazyk, tedy záleží na velikosti písmen. Určitá jména jsou z pochopitelných důvodů rezervována, jako klíčová slova **if**, **while**, **file**, **set** a mnoho dalších.[5]

Substituce při běhu programu

Možnosti substituce byly zmíněny již v předchozích odstavcích. Již bylo řečeno, že použitím znaku $ se nahradí jméno proměnné její hodnotou. Tento fakt by však uživatel jazyka mohl v určitých případech chtít potlačit. To je samozřejmě možné a existují dvě varianty, jak toho docílit. První možností, je uzavřít onu substituci do složených závorek (curly brackets = {, }). Druhou možností je před znak dolaru vložit zpětné lomítko (backslash = \). Tyto vlastnosti demonstruje následující ukázka kódu:[4]

```
1 # Příklad substituce
2
3 set num 23
4 puts "Číslo $num"
5 puts {Číslo $num}
6 puts "Číslo \$num"
```

Výstup předchozího příkladu bude vypadat takto:

```
1 Číslo 23
2 Číslo $num
3 Číslo $num
```

Se složenými závorkami se v TCL lze setkat velmi často, píše se do nich například tělo smyček či podmínek.

Nepostradatelnou součástí programování je možnost vykonávání příkazu a předání jeho návratové hodnoty dalšímu příkazu. O tuto substituci se v TCL starají hranaté závorky (square brackets = [,]). Kód v těchto závorkách se provede nejdříve a jeho výsledek se dále předá příkazu nadřazenému, jedná se tak vlastně o kompozici funkcí. Takto v sobě může být samozřejmě vloženo více příkazů.[4]

```
1 # Příklad substituce pomocí hranatých závorek
2
3 příkaz1 [příkaz2 [příkaz3 arg ..] arg ..] arg ..
```

Zpětné lomítko se obecně užívá ke zrušení speciální funkce následujícího znaku, nejen znaku dolaru.

Složené závorky plní vlastně stejnou funkci jako uvozovky s tím rozdílem, že potlačují všechny substituce. Mají oproti uvozovkám ještě jednu odlišnost, která spočívá v tom, že ve složených závorkách se může objevit znak konce řádku.[4]

Tabulka 2.1: Významy speciálních znaků při substituci v TCL

Znaky	**Význam**
$	substituce proměnných - nahrazení názvu hodnotou
[ ]	vyhodnocení příkazu v závorkách a dosazení návratové hodnoty na původní místo
""	rušení významu mezery jako oddělovače příkazů nebo argumentů
{ }	jako uvozovky, navíc jsou potlačeny všechny substituce uvnitř složených závorek
\	ruší speciální význam následujícího znaku

Matematika

V jazyce TCL neexistuje žádná speciální konstrukce na práci s matematickými funkcemi a operacemi. Místo toho existuje příkaz (tedy vlastně funkce), která jako svůj argument přijímá matematický výraz a jeho výsledek vrací jako návratovou hodnotu. Tento příkaz nese název **expr** (z anglického expession = výraz) a jeho možnosti jsou opravdu značné - k dispozici je například i podmíněný ternární operátor známý z jazyka C.

U příkazu **expr** je možné používat jak konstanty (celočíselné i reálné), tak i proměnné. V jazyce TCL se příkazy a argumenty od sebe vždy oddělují mezerou, avšak příkaz **expr** je v tomto ohledu výjimkou. Je samozřejmě možné mu každou část matematického výrazu předložit jako samostatný argument (tedy klasicky pro jazyk TCL s mezerami), onu výjimku však představuje možnost mu matematický výraz napsat jako jeden argument (tedy v celku - bez mezer) a tento příkaz si již pak se zpracováním výrazu poradí.[4]

```
1 # Příklad použití příkazu expr
2 
3 set a [expr 1+2]
4 set b 20
5 puts "$a + $b se rovná [expr $a + $b]"
```

Výstupem takového programu bude následující řádka:

```
1 3 + 20 se rovná 23
```

V následující tabulce jsou uvedeny podporované matematické operace. Kromě nich však jazyk TCL obsahuje také velké množství jiných matematických funkcí jako **sin**, **cos**, **log**, **pow** a mnoho dalších.[4]

Tabulka 2.2: Matematické funkce v TCL

Operace	Význam
+	unární plus/binární operace sčítání
-	unární mínus/binární operace odčítání
*	součin
/	podíl
%	modulo - zbytek po dělení

Logické, bitové a relační operace

Logické, relační a bitově orientované operace jsou samozřejmě i v jazyce TCL k dispozici a svou podstatou vychází z jazyka C. Jednotlivé symboly a jejich význam popisuje tabulka č. 2.3.

Tabulka 2.3: Logické, bitové a relační operace v TCL

Operace	**Význam**
<<	bitový posun doleva
>>	bitový posun doprava
~	bitová negace
!	logická negace
&	bitový operátor AND
|	bitový operátor OR
^	bitový operátor XOR
&&	logický součin
||	logický součet
?:	ternární podmíněný výraz
<	menší než
>	větší než
<=	menší nebo rovno
>=	větší nebo rovno
==	rovnost
!=	nerovnost

Funkce

Základním stavebním kamenem jazyka jsou funkce (resp. příkazy nebo procedury). Ty lze v TCL rozdělit do tří typů.[4]

1. vestavěné funkce
2. uživatelské funkce vytvořené příkazem **proc**
3. externí funkce (například v jazyce C)

Všechny tyto funkce mohou mít libovolný počet parametrů, který se dokonce může měnit a mohou také vracet návratovou hodnotu.

Některé vestavěné funkce již byly představeny. Jednou z nich je také funkce **proc**, jež umožňuje uživateli si definovat vlastní proceduru. Příkaz **proc** následuje jméno funkce, dále pak parametry psané ve složených závorkách a nakonec tělo funkce, taktéž ve složených závorkách. Parametry funkce se v jejím těle chovají jako zadefinované proměnné. Zápis těla funkce do složených závorek dává smysl, jelikož právě složené závorky zabrání vyhodnocení těla funkce při jejím vytvoření. Navíc je nutné, aby otevírací závorka těla funkce byla na stejném řádku jako příkaz **proc**. Je to z toho důvodu, že příkazy v TCL jsou řádkově orientované a jazyk takto pozná, že oblast ve složených závorkách patří právě k příkazu **proc**. Uvnitř těla se již může kód formátovat libovolně, neboť je povoleno, aby uvnitř složených závorek byl znak nového řádku.[4]

```
# Příklad vytvoření funkce - bez parametrů

proc ahoj {} {
  puts "Ahoj světe"
}
```

Návratová hodnota se předává pomocí příkazu **return**, který zároveň přeruší funkci a případné příkazy za ním se neprovedou. Pokud funkce skončí jiným příkazem než **return**, předána je návratová hodnota posledního příkazu. Návratovou hodnotu lze samozřejmě ignorovat.[4]

```
# Příklad vytvoření funkce - s parametry

proc secti {a b} {
  return [expr $a + $b]
}
```

Opět je možno při předávání parametrů využít faktu, že vše je v TCL vnímáno jako textový řetězec, není tedy třeba předávat pouze hodnoty, ale lze například předat i operátor, či cokoli dalšího, co by se mohlo hodit.[4]

```
# Příklad vytvoření funkce - s parametry

proc mat_operace {a b operator} {
  return [expr $a $operator $b]
}

puts [mat_operace 2 3 +]
puts [mat_operace 2 3 *]
```

Výstup toho příkladu bude vypadat následovně:

```
5
6
```

TCL také podporuje možnost zavedení parametru s defaultní hodnotou. Takový parametr pak není nutné funkci předávat a ta bude počítat s jeho přednastavenou hodnotou, je-li třeba, lze naopak tento parametr specifikovat na potřebnou hodnotu.[6]

```
# Příklad funkce s defaultním parametrem

proc porovnej {a {prah 90}} {
  if { $a < $prah } {
    puts "$a je menší než $prah"
  } else {
    puts "$a není menší než $prah"
  }
}

porovnej 50
porovnej 50 20
```

První volání funkce **porovnej** v předchozí ukázce porovná číslo 50 s defaultně zadaným číslem 90. V druhém případě měníme proměnnou **prah** a číslo 50 se tedy porovnává s číslem 20. Výstup vypadá následovně:

```
50 je menší než 90
50 není menší než 20
```

Podmíněné příkazy

V podstatě každý programovací jazyk má možnost rozvětvení běhu programu podle nějaké podmínky a TCL není žádnou výjimkou. Možnosti, jak program rozvětvit, jsou vlastně dvě:[7]

1. příkazy **if** a **else**,
2. příkaz **switch**.

Podmínka, kterou je většinou nutné psát do složených závorek kvůli předčasnému vyhodnocení, se vyhodnocuje stejným způsobem jako u příkazu **expr**. Je tedy možné použít dostupné matematické, logické i relační operátory. Výsledkem podmínky není nic jiného než řetězec, který se dále interně převede na boolovskou hodnotu. Pokud řetězec obsahuje nenulové číslo, hodnotu *true* nebo hodnotu *yes*, je vyhodnocen jako logická pravda (true). V ostatních případech se řetězec vyhodnotí jako nepravda (false).

V případě kladného vyhodnocení podmínky příkazu **if** se provede ta část kódu, která přímo následuje. V opačném případě je provedena větev **else**, pokud je vůbec přítomna. V jazyce TCL je konstrukce podmínky pomocí **if** a **else** příkazů značně benevolentní. Je možné psát příkaz **if** i s dalšími příkazy, kterými jsou **then** a **else**, povoleno je však tyto slova vůbec neuvádět, i přes to, že jejich větve jsou v příkazu přítomny. Konkrétní ukázka jednoduchého příkazu **if** je v předchozí kapitole.[7]

```
# Možné konstrukce podmínky s if

if podmínka těloPodmínky
if podmínka then těloPodmínky
if podmínka těloPodmínky těloDruhéVětve
if podmínka těloPodmínky else těloDruhéVětve
if podmínka then těloPodmínky těloDruhéVětve
if podmínka then těloPodmínky else těloDruhéVětve
```

Pro vyjádřeních složitějších podmínek existuje k příkazu **if** příkaz **elseif**. V tomto případě není zaručena taková volnost jako u **then** a **else** - pokud je větev **elseif** přítomna, je nutné aby byl příkaz uveden.[7]

```
# Příklad podmínky s if a elseif

if {$a < 10} {
  puts "$a je menší než 10"
} elseif {$a < 100} {
  puts "$a je mezi 10 a 100"
} else {
  puts "$a je větší než 100"
}
```

Jak je vidět i z předchozího příkladu, je nutné opět zajistit, aby celý příkaz **if** byl se všemi svými komponenty na jednom řádku. Tedy symbol nového řádku se smí nacházet pouze uvnitř složených závorek, jinak příkaz nebude správně zapsán a program bude generovat chybu. To platí i při zápisu podmínky beze slova **then** či **else**.[7]

```
# Korektní zápis podmínky

if {$a < 10} {
  puts "$a je menší nez 10"
} {
  puts "$a je větší nebo rovno 10"
}

# Chybný zápis podmínky

if {$a < 10} {
  puts "$a je menší nez 10"
}
{
  puts "$a je větší nebo rovno 10"
}
```

Druhou variantou psaní podmíněného větvení kódu je příkaz **switch**. V následující ukázce je znázorněno, jak se příkaz zapisuje. Lze si všimnout, že oproti například jazyku C se v TCL nepíše ukončovací příkaz typu **break**, vždy se provede pouze jedna větev.[8]

```
1 # Zápis příkazu switch
2
3 switch řetězec vzor1 větev1 vz2 vět2 .. default větN
```

Součástí tohoto příkazu může být i část, označená slovem **default**, která se provede, pokud nebude splněna ani jedna z předchozích možností. Tato sekce u příkazu být nemusí, pokud je však přítomna, musí být poslední ze všech možností.[9]

U příkazu **switch** se nejedná o vyhodnocení klasické boolovské podmínky, ale jelikož jde o porovnávání řetězců, stává se tak tento příkaz mocnějším nástrojem než jak může být vnímán v jiných jazycích. Oproti klasickému porovnávání dvou řetězců na shodu (a tedy v podstatě vyhodnocování boolovské podmínky) může příkaz **switch** v TCL vyhodnocovat i regulární výrazy a tak například hledat částečné shody řetězců. Způsob porovnávání řetězců jde zvolit pomocí přepínače, který se zapíše jako první argument příkazu.[8]

```
1  # Příklad příkazu switch - exactní porovnání
2
3  switch "abc" {
4    aba {
5      puts "první volba"
6    }
7    abd {
8      puts "druhá volba"
9    }
10   default {
11     puts "implicitní volba"
12   }
13 }
14
15 # Příklad příkazu switch - regulární výraz
16
17 switch -glob "abc" {
18   a* {
19     puts "první volba"
20   }
21   ?bb {
22     puts "druhá volba"
23   }
24   default {
25     puts "implicitní volba"
26   }
27 }
```

Zde si lze všimnout, že uvnitř příkazu **switch** není nutné psát všechny výrazy přesně za sebe na řádek, neboť celé tělo příkazu je uzavřeno do složených závorek.

Tabulka 2.4: Přepínače příkazu switch

Přepínač	Význam
-exact	porovnávání na shodu řetězců - implicitní volba, netřeba zadávat
-glob	tzv. glob-style matching - jednodušší forma výrazu (jako v shellu)
-regexp	plnohodnotný regulární výraz

Cykly

Dalším způsobem řízení průběhu programu jsou iterační příkazy. V TCL jsou opět dvě základní varianty, jak cyklus vytvořit. Jsou jimi příkazy **while** a **for**.

Příkaz **while**, stejně jako v jiných programovacích jazycích, znamená, že před prvním (a každým dalším) během smyčky se provede test podmínky (podobně jako u příkazu **if**). Pokud je podmínka splněna (true), pak se provede tělo smyčky, pokud ne (false), pokračuje se kódem za smyčkou.[7]

```
# Příklad smyčky while - výpis čísel 1 až 10

set i 1
while {$i<=10} {
  puts $i
  incr i
}
```

V cyklech, ale i v jiných případech, se často může hodit zvýšit hodnotu číselné proměnné o 1. K tomu účelu je v TCL přímo vytvořen příkaz **incr**, viz. předchozí ukázka. Funkce **incr** však umí k číselné hodně přičítat číslo dle uživatelovi volby, například i záporné a tedy fakticky odčítat. Toho lze docílit předáním požadované hodnoty jako argumentu příkazu **incr**.[7]

```
# Příklad smyčky while - výpis čísel od 10 do 1

set i 10
while {$i} {
  puts $i
  incr i -1
}
```

Další možností pro vytvoření cyklu je příkaz **for**.

```
# Zápis příkazu for

for úvod podmínka iteračníPříkaz smyčka
```

Zápis této funkce je intuitivní. Prvním argumentem je kód, který se provede před první kontrolou podmínky, obvykle zde bývá nastavení iterační proměnné. Dalším parametrem je podmínka, která se vyhodnocuje vždy před spuštěním smyčky. Následuje parametr s iteračním příkazem, těch může být dokonce víc. Posledním parametrem je samotné tělo smyčky.

Všechny tyto parametry se většinou zapisují do složených závorek, opět aby se předešlo předčasnému vyhodnocení kódu, který obsahují. Oproti příkazu **while** je příkaz **for** přehlednější, neboť všechny pro cyklus podstatné části kódu se zapisují jako argumenty vedle sebe na řádek. V zásadě však obě funkce plní stejný účel.[7]

```
1 # Příklad smyčky for - výpis čísel od 10 do 1
2
3 for {set i 10} {$i} {incr i -1} {
4   puts $i
5 }
```

Jak u příkazu **while**, tak u příkazu **for** je možné použít příkazy pro okamžité přerušení běhu cyklu (**break**) a nebo pro okamžité pokračování na následující smyčku (**continue**).[7]

Slovník

Příkaz **dict** (z anglického dictionary = slovník) slouží k mapování hodnot na klíče. Na rozdíl od vytvoření slovníku pomocí asociativního pole jako v ukázce na straně 22, nabízí **dict** mnoho různých funkcí pro práci s klíči i hodnotami. Vybrané z nich jsou uvedeny v tabulce 2.5.[10]

Tabulka 2.5: Funkce dict

Funkce	**Význam**
dict create	vrací nově vytvořený slovník se zadanými hodnotami namapovanými na klíče
dict exists	vrací boolovskou hodnotu, zda klíč existuje ve slovníku
dict filter	vrací nově vytvořený slovník, který obsahuje pouze hodnoty odpovídající zadanému filtru
dict get	vrací hodnotu namapovanou na zadaný klíč
dict keys	vrací všechny klíče slovníku
dict remove	vrací kopii slovníku, která neobsahuje zadaný klíč
dict set	vrací slovník s nastavenými klíči a hodnotami
dict size	vrací počet párů klíč - hodnota ve slovníku
dict values	vrací všechny hodnoty slovníku

[11]

Při práci se slovníkem je třeba si dát pozor na fakt, že funkce **dict** nepracuje přímo se slovníkem zadaným, ale vytváří jeho kopii, se kterou provede požadovaný úkon. Nově vytvořený slovník je pak předáván skrze návratovou hodnotu. Z tohoto důvodu může být práce se slovníkem méně intuitivní.

```
1  # Práce se slovníkem
2
3  set překladač [dict create auto "car" kolo "wheel"]
4  puts $překladač
5
6  dict remove $překladač auto
7  puts $překladač
8
9  set překladač [dict remove $překladač auto]
10 puts $překladač
```

Výsledek bude následující:

```
1 auto car kolo wheel
2 auto car kolo wheel
3 kolo wheel
```

V předchozí ukázce je uveden příklad práce se slovníkem. Je z ní patrné, že první příkaz **remove** neodstraní požadovaný klíč s hodnotou ze slovníku `překladač`. Udělá to ve slovníku novém, který vrací jako návratovou hodnotu. Ta však v tomto případě není využita.

V případě druhém je nový slovník přiřazen do původní proměnné `překladač` a tím starý slovník nahradí. Projeví se tedy i ona funkce odstranění klíče a hodnoty.

Namespace

Namespace, česky jmenný prostor, je kontejner, ve kterém je možno sdružit sadu proměnných a funkcí. V TCL je k dispozici od verze 8.0. Před jeho představením bylo vše ukládáno a reprezentováno globálně. Výhodou namespace je možnost, užít stejná jména proměnných a funkcí v různých jmenných prostorech a také možnost logického rozdělení kódu.

K práci se jmenným prostorem slouží funkce přímo pojmenovaná **namespace**. Pro přístup k funkcím či proměnným z daného prostoru se užívá operátor, který má podobu dvojité dvojtečky (double colon - `::`).

Následující ukázka demonstruje použití jmenného prostoru a přístup k jeho obsahu. TCL dokonce umožňuje vnořování více namespace, ke kterým se dále přistupuje pomocí dalších dvojitých dvojteček.[12]

```
1  # Ukázka práce s namespace
2
3  namespace eval Prostor {
4    variable vysledek
5  }
6
7  proc Prostor::Secti { a b } {
8    set Prostor::vysledek [expr $a + $b]
9  }
10
11 Prostor::Secti 10 13
12 puts $Prostor::vysledek
```

Výsledek předchozí ukázky je následující:

```
1  23
```

Mezi nejdůležitější funkce namespace patří v ukázce znázorněný **eval**, jenž jmenný prostor vytvoří a vyhodnotí jeho kód. Opakem této funkce je **delete**, který jmenný prostor smaže i s obsahem.[13]

2.1.3 Strukturované datové typy

Asociativní pole

Prvním představeným strukturovaným datovým typem je **asociativní pole**. Jako další imperativní programovací jazyky, umožňuje TCL práci s poli. Tento jazyk však nemá pole indexované klasicky číselnými hodnotami, ale (jak název napovídá) jde o pole, kde je klíčem obecný řetězec. Za zmínku stojí, že asociativní pole byla dlouho dobu známá jako *hashe* - viz. například jazyk Perl.[7]

```
1  # Ukázka pole s číselnými klíči
2  set pole(0) 10
3  set pole(1) 20
4
5  puts $pole(0)
6  puts $pole(1)
```

V předchozí ukázce je příklad pole s klasickými číselnými hodnotami indexů. Je však potřeba mít stále na paměti, že tyto číselné hodnoty jsou jako všechny hodnoty v TCL vnímány jako řetězce.[7] Výsledek bude následující:

```
1  10
2  20
```

Pro demonstraci vlastností asociativního pole se například hodí definice jednoduchého slovníku.[7]

```
set slovnik(auto) car
set slovnik(motorka) motorbike
set slovnik(vznasedlo) hovercraft
```

Práci se samotným polem je možné provádět pomocí příkazu **array**. Ta jako svůj první parametr přijímá řetězec, jenž určuje, jaká operace se s polem má vykonat.[7] Konkrétní případy jsou ukázány v tabulce 2.6.

Tabulka 2.6: Příkaz array pro práci s polem

Funkce	**Význam**
array get *pole*	vrací všechny hodnoty klíčů i prvků pole
array get *pole vzor*	vrací všechny hodnoty klíčů i prvků pole, kde klíče odpovídají zadanému vzoru
array names *pole*	vrací všechny klíče (indexy) pole
array names *pole vzor*	vrací všechny klíče pole, které odpovídají zadanému vzoru
array set *pole seznam*	vytváří pole ze seznamu (indexuje se automaticky)
array exists *pole*	provede ověření (predikát), zda existuje pole o zadaném názvu – vrací řetězec 0 nebo 1
array size *pole*	vrátí počet prvků v poli – vhodné pro počítané smyčky

[7]

Procházení polem je možné realizovat pomocí parametrů *startsearch*, *donesearch* a *nextelement*. V některých případech však může být výhodnější převést pole na seznam a ten procházet pomocí smyčky **foreach**. Převedení pole na seznam fakticky provede příkaz **array get pole**.[7]

Seznam

Dalším strukturovaným datovým typem, který je v TCL k dispozici je **seznam**. Ten se od pole liší především tím, že jeho prvky mohou být libovolného typu - například i další seznamy. Tímto způsobem tak lze z původně lineární datové struktury vytvořit například n-ární strom.

Na následujícím příkladu je vidět, že vytvoření seznamu je velmi jednoduché - není k tomu potřeba žádná speciální jazyková konstrukce. Pro zápis seznamu je třeba jen složených závorek. Při zápisu kódu do bloků jsou tak pomocí složených závorek vlastně vytvářeny seznamy.[7]

Pro práci se seznamy existuje mnoho funkcí, některé z nich jsou uvedeny v tabulce 2.7.

```
1 # Ukázka definice seznamů
2
3 set seznam1 { 1 2 3 }
4 set seznam2 { jedna dve tri }
5 set seznam3 { 1 {2 3} {4 5} }
```

Tabulka 2.7: Funkce pro práci se seznamem

Funkce	Význam
list	vytvoření seznamu z argumentů, které jsou tomuto příkazu zadány
concat	spojení dvou a více seznamů
llength	získání počtu prvků v seznamu
split	rozložení řetězce na seznam buď podle bílých znaků nebo podle specifikovaného oddělovače
join	vytvoření řetězce spojením prvků seznamu, mezi něž se může volitelně vložit oddělovač
lappend	přidání jednoho či více prvků do seznamu
linsert	vložení jednoho či více prvků na danou pozici (index)
lindex	získání prvku ze seznamu na dané pozici (indexu)
lreplace	nahrazení prvků v seznamu
lrange	vyjmutí více prvků ze seznamu (souvislá oblast)
lsearch	hledání prvků v seznamu (možné i podle regulárních výrazů)
lsort	setřídění prvků v seznamu podle zadaných kritérií

[7]

Pro průchod seznamem je možné použít standardní smyčky a příkazy **llength** a **lindex**. To však není příliš efektivní a přehledný způsob procházení seznamu. Z tohoto důvodu v TCL existuje speciální forma smyčky. Je to již dříve zmíněný příkaz **foreach**. Tento příkaz má následující formu:[7]

```
1 # Zápis příkazu foreach
2
3 foreach proměnná seznam těloSmyčky
```

Jmenná podobnost se smyčkou **for** je na místě, neboť i funkce **foreach** používá řídící proměnnou. Tato proměnná však nenabývá číselných hodnot, ale jsou do ní postupně dosazovány jednotlivé prvky seznamu. Navíc smyčka **foreach** může pracovat s více než jednou řídící proměnnou.[7]

```
# Ukázka smyčky foreach

# Jednoduchá smyčka
foreach i {jedna dve tri} {
  puts $i
}

# Seznam jako proměnná
foreach {i j} {1 2 3 4 5 6} {
    puts "i = $i"
    puts "j = $j"
}

# Více proměnných
foreach i {1 2 3} j {4 5 6} {
    puts "i = $i"
    puts "j = $j"
}
```

Výstup předchozího příkladu bude vypadat následovně:

```
jedna
dve
tri

i = 1
j = 2
i = 3
j = 4
i = 5
j = 6

i = 1
j = 4
i = 2
j = 5
i = 3
j = 6
```

2.1.4 Práce s řetězci

Není divu, že programovací jazyk TCL má velkou podporu pro práci s řetězci, když všechny proměnné i příkazy jsou interně reprezentovány jako řetězce. Proto existuje velké množství sofistikovaných funkcí pro práci s nimi. Mnoho takových funkcí se opírá o regulární výrazy.[8]

Regulární výrazy

TCL podporuje dva typy regulárních výrazů. Prvním z nich je tzv. **glob matching**, který v mnohém odpovídá regulárním výrazům známým ze světa unixového shellu. Typicky takové regulární výrazy používají příkazy jako **ls**, **mv**, **cp** apod.

Druhým typem je obsáhlejší varianta regulárních výrazů, která koresponduje s regulárními výrazy použitými ve vyhledávací utilitě **grep** či textovém editoru **Vim**. V tabulkách 2.8 a 2.9 jsou vyobrazeny podporované znaky.[8]

Tabulka 2.8: Znaky regulárních výrazů prvního typu

Znak	**Význam**
`*`	libovolně dlouhá sekvence znaků
`?`	libovolný znak (pouze jeden)
`[...]`	libovolný znak ze zadané množiny (lze použít i interval)
znak	odstranění speciálního významu znaku
ostatní znaky	další znaky se porovnávají bez dalšího zpracování

Tabulka 2.9: Znaky regulárních výrazů druhého typu

Znak	**Význam**
`.`	libovolný znak
`*`	nula či více výskytů předchozí položky
`+`	jeden či více výskytů předchozí položky
`|`	volba mezi dvěma výrazy (or)
`[...]`	libovolný znak ze zadané množiny (lze použít i interval)
`^`	začátek řetězce (uvnitř závorek funguje jako negace)
`$`	konec řetězce
znak	odstranění speciálního významu znaku
ostatní znaky	další znaky se porovnávají bez dalšího zpracování

Funkce pro práci s řetězci

Tabulka 2.10 představuje vybrané funkce pro práci s řetězcem. Nejzajímavější z nich je funkce **string**, která podobně jako příkaz **array** při práci s poli, změnou svého prvního parametru vykoná různé manipulace s řetězcem. Některé její funkce jsou popsané v tabulce 2.11.[8]

Tabulka 2.10: Funkce pro práci s řetězci

Funkce	Význam
append	přidání znaků na konec řetězce
string	manipulace s řetězcem s mnoha dalšími volbami
regexp	vyhledání shody v řetězci podle zadaného regulárního výrazu (dle obsáhlejšího - druhého typu)
regsub	vyhledání shody v řetězci podle zadaného regulárního výrazu a zápis nového řetězce do zadané proměnné
format	formátování řetězce ve stylu funkce **sprintf()** jazyka C
scan	čtení hodnot z řetězce ve stylu funkce **sscanf()** jazyka C
binary	formátování a získávání informací z binárních řetězců

Tabulka 2.11: Funkce string

Funkce	Význam
string length	vrací délku řetězce
string index	vrací konkrétní znak na určitém indexu (první znak je na pozici 0, jako v C)
string first	vyhledání podřetězce v řetězci, vrací první pozici od začátku nebo -1, pokud podřetězec není nalezen
string last	vyhledání podřetězce v řetězci, vrací první pozici od konce nebo -1, pokud podřetězec není nalezen
string compare	lexikografické porovnání dvou řetězců, vrací -1, 0 nebo 1
string match	porovnání řetězce se vzorem, je možné použít jednodušší regulární výrazy, vrací 0 nebo 1
string range	vrací podřetězec zvolený dvěma indexy
string tolower	převod všech písmen řetězce na malá
string toupper	převod všech písmen řetězce na velká
string trim	vrací podřetězec, ze kterého jsou odstraněny vybrané počáteční a koncové znaky
string trimleft	jako trim, ale odstraněny jsou pouze počáteční znaky
string trimright	jako trim, ale odstraněny jsou pouze koncové znaky

2.1.5 Práce se soubory

Práce s daty

Práce se soubory je ve své podstatě velmi přímočará a jednoduchá. K dispozici jsou příkazy pro otevření souboru, čtení a zápis dat a uzavření souboru. K souborům se přistupuje stejně jako k datovým tokům. V dokumentaci TCL se vyskytuje také výraz kanál.

Při spuštění programu jsou stejně jako v jiných programovacích jazycích otevřeny tři základní kanály: **stdin** (standardní vstup), **stdout** (standardní výstup) a **stderr** (chybový výstup).

K otevření nového kanálu se využívá příkaz **open**. Při úspěšném otevření souboru vrátí jako svou návratovou hodnotu identifikátor kanálu, se kterým dále pracují další funkce. Příkaz open má následující formu:[8]

```
# Zápis příkazu open

open jménoSouboru přístup práva
```

Je možné zadat jméno otevíraného souboru s absolutní nebo relativní cestou. Řetězec *přístup* specifikuje, zda je soubor otevřen pro čtení (**read**), zápis (**write**) nebo přidávání na konec souboru (**append**). Parametr *práva* určuje, jaká práva budou přiřazena nově vytvořenému souboru, tedy při otevření neexistujícího souboru. Nastavují se podobně jako je tomu u příkazu shellu **chmod**.

Pro zavření kanálu slouží příkaz **close**, který jako svůj jediný parametr přijímá identifikátor kanálu, který vrací právě funkce open.[8]

```
# Zápis příkazu close

close kanál
```

Otevření a zavření souboru je pouze základ, hlavním smyslem práce se soubory je přístup k datům uvnitř. K tomuto účelu slouží další funkce, které pracují s otevřeným kanálem. Jejich popis se nachází v tabulce 2.12.[8]

Tabulka 2.12: Práce s daty souborů

Funkce	Význam
read	načte zadaný počet znaků, pokud není počet zadán, načte celý soubor, lze použít i pro binární soubory
gets	načte jeden řádek souboru
puts	zapíše řetězec do souboru
format	formátovaný zápis do řetězce, většinou užíván spolu s funkcí **puts**
scan	čtení hodnot z řetězce, většinou užíván spolu s funkcí **gets**

Práce se soubory samotnými

Příkaz, který pracuje se soubory samotnými namísto s jejich obsahem, se nazývá **file**. Podobně jako u příkazů **array** a **string** se jeho funkce liší podle prvního parametru, který mu je předán. V tabulce 2.13 jsou uvedené vybrané funkce příkazu.[8]

Tabulka 2.13: Příkaz file pro práci se soubory

Funkce	**Význam**
file size *soubor*	vrací délku souboru v bytech
file mtime *soubor*	vrací čas poslední úpravy souboru
file extension *soubor*	vrací příponu souboru
file exists *soubor*	vrací hodnotu 1, pokud soubor existuje
file isdirectory *soubor*	vrací hodnotu 1, pokud soubor existuje a je adresář
file isfile *soubor*	vrací hodnotu 1, pokud soubor existuje a je běžný soubor (ne adresář ani link)
file readable *soubor*	vrací hodnotu 1, pokud soubor lze číst
file writable *soubor*	vrací hodnotu 1, pokud do souboru lze zapisovat
file executable *soubor*	vrací hodnotu 1, pokud je soubor spustitelný
file mkdir *soubor*	vytvoří neexistující adresáře
file rename *soubor novýSoubor*	přejmenuje soubor, pokud je v novém jméně jiná cesta, pak ho také přesune
file copy *soubor novýSoubor*	podobně jako **rename**, ale soubor kopíruje
file delete *soubor*	smaže soubor

[14]

2.2 Sběrnice CAN

Controller Area Network je sériový komunikační protokol původně vyvinutý firmou Bosch pro použití v automobilech. Využití tohoto protokolu se však neomezuje pouze na automobilový průmysl. Se sběrnicí CAN se lze setkat i v integrovaných obvodech, například v různých průmyslových aplikacích. Hlavními důvody rozšíření protokolu je nízká cena, snadné nasazení, spolehlivost, vysoká přenosová rychlost a snadná rozšiřitelnost.

Protokol CAN je definován normou ISO 11898, který popisuje fyzickou vrstvu a specifikaci **CAN 2.0A**. Později byla ještě vytvořena specifikace **CAN 2.0B**, která zavádí dva druhy formátu zpráv - standardní a rozšířený. V normě je definována pouze fyzická a linková vrstva referenčního modelu ISO/OSI[9]. Aplikační vrstva protokolu CAN je definována několika různými a vzájemně nekompatibilními standardy, jako je CAL, CANopen, DeviceNet nebo CAN Kingdom.[15]

[9]ISO/OSI - referenční model pro standardizaci počítačových sítí.

2.2.1 Důvody zavedení

Všechna moderní vozidla jsou vybavena celou řadou elektronických zařízení a řídících systémů. V průběhu vývoje moderních vozidel se zvyšoval nárok uživatelů a tím pádem také počet elektronických zařízení. S rostoucím počtem těchto zařízení se však zvyšuje i komplexnost funkcí a nutnost vzájemné komunikace mezi těmito systémy. Propojení, kde pro každý signál je potřeba jedna přenosová linka, se z důvodu velkého množství signálů stalo neúnosné.

Veškeré jednotky, jenž mezi sebou ve vozidle komunikují nebo sbírají data z jednotlivých senzorů jsou propojeny právě pomocí sběrnice CAN. Účelem použití této sběrnice v automobilovém průmyslu je zajištění komunikace mnoha jednotek aniž by docházelo v velkému zatížení centrální řídící jednotky.[15]

2.2.2 Základní vlastnosti

Sériový komunikační protokol CAN umožňuje distribuované řízení systémů v reálném čase a s vysokou mírou zabezpečení vůči chybám.

Protokol je typu **multi-master**, tedy každý uzel na sběrnici může být **master** a řídit tak chování ostatních uzlů. Tento fakt zvyšuje spolehlivost systému, neboť není zapotřebí přítomnost řídícího uzlu a při poruše jednoho uzlu probíhá komunikace ve zbytku systému dál.

Řízení přístupu k médiu je realizováno náhodným přístupem a kolize jsou řešeny prioritním rozhodováním. Komunikace po sběrnici CAN probíhá pomocí zpráv. Zprávy mohou mít různý tvar, jako např. datová zpráva nebo žádost o data. Management komunikace je zajištěn pomocí jiných zpráv, jako jsou chybové zprávy a zprávy o přetížení.

Zprávy, které jsou vysílány na sběrnici jedním uzlem neobsahují žádnou informaci o uzlu cílovém. Zprávy jsou přijímány všemi uzly připojenými na sběrnici. Každá zpráva je uvozena identifikátorem, jenž udává její význam a prioritu. Protokol CAN zajišťuje, že zpráva s vyšší prioritou je v případě kolize doručena přednostně. Dále je podle identifikátoru možné určit, aby uzel přijímal pouze zprávy, které jsou pro něj relevantní.

Sběrnice CAN je rozdělena do tří vrstev z důvodu transparentnosti návrhu a flexibility implementace.

- vrstva objektů
- transportní vrstva
- fyzická vrstva

Vrstva objektů a transportní vrstva zahrnuje veškeré služby a funkce v rámci linkové vrstvy, tak jak ji definuje model ISO/OSI.

Vrstva objektů je zodpovědná za nalezení zprávy, která má být odeslána, rozhodnutí, zda přijatá zpráva od transportní vrstvy má být použita a za poskytnutí rozhraní pro aplikační vrstvu související s hardwarem.

Transportní vrstva má za úkol především realizovat přenosový protokol. Její funkcí je například řízení rámců, kontrola a signalizace chyb. Uvnitř této vrstvy je také rozhodnuto, zda je sběrnice volná pro nový přenos či příjem dat.

Úkolem **fyzické vrstvy** je vlastní přenos bitů mezi uzly. Uvnitř jedné sítě má fyzická vrstva stejné vlastnosti pro všechny přítomné uzly, je však možné si její parametry zvolit tak, aby vyhovovaly dané aplikaci.[15]

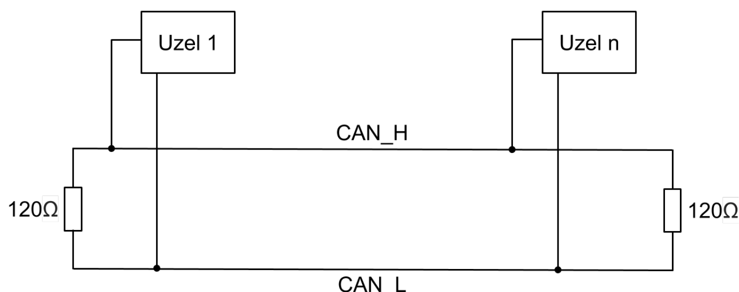
2.2.3 Fyzická realizace

Protokol CAN má definované vlastní rozhraní fyzického přenosového média. Tento fakt ho odlišuje od modelu ISO/OSI. Vlastnosti jeho fyzické vrstvy jsou však velkou předností protokolu CAN.

Standard protokolu CAN definuje dvě vzájemně komplementární hodnoty, kterých může sběrnice nabývat - **dominant** a **recessive**. Ty v podstatě přestavují zobecněný ekvivalent logických úrovní. Určení stavu sběrnice je jednoduché a jednoznačné. Pokud všechny uzly vysílají *recessive* bit, pak se na sběrnici nachází úroveň *recessive*. Vysílá-li alespoň jeden uzel *dominant* bit, pak je na sběrnici úroveň *dominant*.

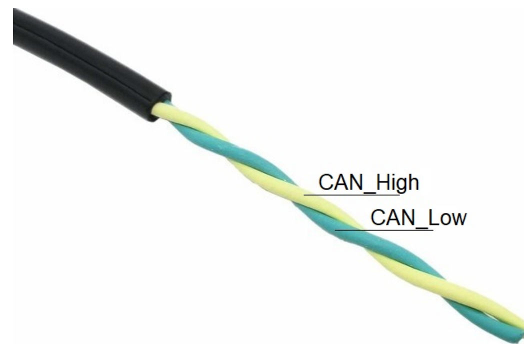
Jako fyzické přenosové médium je nejčastěji užívána diferenciální sběrnice definovaná normou ISO 11898, vyobrazena na obrázku 2.1. V této normě jsou definovány elektrické vlastnosti vysílače a přijímače i principy časování, synchronizace a kódování jednotlivých bitů.

Sběrnici tvoří dva vodiče označované jako **CAN_H** (CAN_High) a **CAN_L** (CAN_Low). V automobilech jsou tyto vodiče rozváděny jako kroucená dvojlinka, aby se dosáhlo co nejlepších fyzikálních vlastností přenosu signálu (obr. 2.2). Úrovně *dominant* a *recessive* jsou definované jako rozdílové napětí na těchto vodičích. Velikost rozdílového napětí úrovně *recessive* je $V_{diff} = 0$V a úrovně *dominant* je $V_{diff} = 2$V. Na obou koncích je sběrnice zakončená odporem, který slouží k eliminaci odrazů na vedení. V situaci, kdy je vyžadováno manuální připojení uzlu ke sběrnici, bez nutnosti pájení vodičů, se jednotlivá zařízení na sběrnici připojují nejčastěji pomocí konektoru D-SUB. Zapojení konektoru je popsáno obrázkem 2.3.[15]

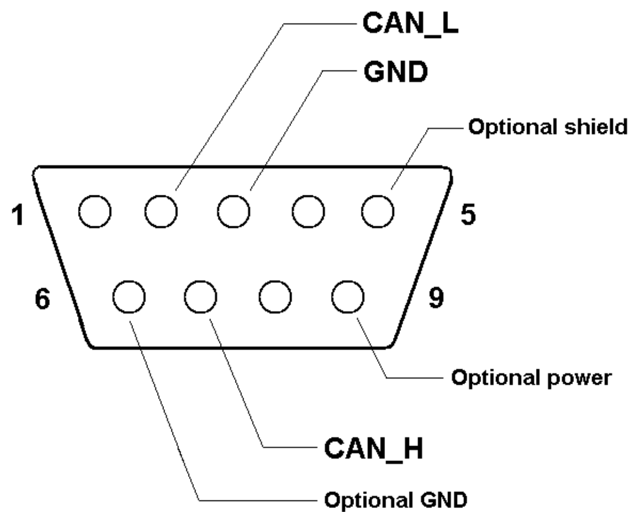


Obrázek 2.1: Fyzická realizace CAN dle normy ISO 11898

Počet uzlů připojených ke sběrnici je teoreticky neomezený, prakticky se však uvádí omezení s ohledem na zatížení sběrnice kolem 64 na segment. Maximální přenosová rychlost 1Mbit/s je dosažitelná pouze na krátkou vzdálenost, tedy do 40m. Se vyšší vzdáleností rychlost přenosu prudce klesá. Je to dáno původním posláním sběrnice CAN, která byla určena pro komunikaci na malé vzdálenosti v automobilech.



Obrázek 2.2: Kroucená dvojlinka CAN sběrnice
[16]



Obrázek 2.3: Zapojení sběrnice CAN s konektorem D-SUB
[17]

2.2.4 Řízení přístupu a řešení kolizí

Každý z uzlů může zahájit vysílání, jakmile je síť v klidovém stavu, což vyplývá z faktu, že protokol CAN je typu *multimaster*. Tedy kdo začne první, ten vysílá a ostatní čekají opět na klidový stav. Pouze chybové zprávy jsou výjimkou tomuto pravidlu. Ty se dají vysílat okamžitě po zjištění chyby kterýmkoli účastníkem.

Pokud nastane situace, že více uzlů zahájí vysílání zároveň, pak se rozhodne, kdo bude pokračovat na základě priority. Vyšší prioritu má uzel, který má nižší identifikátor. Identifikátor je přenášen vždy na začátku zprávy. Každý uzel při vysílání sleduje skutečnou hodnotu na sběrnici. Pokud zjistí, že vysílá jinou hodnotu, než která na sběrnici ve skutečnosti je přítomna, pak přeruší své vysílání a čeká na další možnost. Jediný případ, kdy hodnota na sběrnici může být jiná, něž hodnota vysílaná, je když uzel vysílá hodnotu *recessive* a na sběrnici je přítomna hodnota *dominant*. Tato situace znamená, že zároveň vysílá uzel, který má nižší prioritu a tedy přednost. Tímto způsobem je docíleno faktu, že po odvysílání identifikátoru již vysílá pouze uzel s jeho nejnižší hodnotou a data tedy nikdo nepoškodí.[15]

2.2.5 Zabezpečení přenášených dat

Velkou výhodou protokolu CAN je silný mechanismus pro zabezpečení přenosu dat. Současně působí následující mechanismy:[15]

- monitoring,
- CRC kód,
- vkládání bitu,
- kontrola zprávy,
- potvrzení přijaté zprávy.

Monitoring

Vysílač porovnává hodnotu vysílaného bitu s úrovní přítomnou na sběrnici. Pokud se tato hodnota liší v průběhu vysílání identifikátoru, pak to znamená, že zároveň vysílá uzel s vyšší prioritou. Vysílač tedy přestane a počká na volnou sběrnici, kdy se zprávu pokusí odeslat znovu. Pokud rozdíl zjistí v jiné části zprávy je vygenerována chyba bitu.[15]

CRC kód

Cyclic Redundancy Check je poslední pole vysílané zprávy o délce 15 bitů. Generuje se ze všech dosud odvysílaných bitů podle definovaného polynomu. Detekuje-li libovolný uzel na sběrnici CRC chybu, je vygenerována chyba CRC.[15]

Vkládání bitu

Pokud uzel vysílá ve zprávě pět po sobě jdoucích bitů stejné hodnoty, pak do zprávy navíc vloží bit hodnoty opačné. Toto opatření slouží nejen k detekci chyb, ale také ke správnému časovému synchronizování přijímačů jednotlivých uzlů. Při detekci chyby se generuje chyba vkládání bitu.[15]

Kontrola zprávy

Zpráva je kontrolována dle specifikace a pokud je na nějaké pozici nepovolená hodnota, je vygenerována chyba rámce.[15]

Potvrzení přijetí zprávy

Každé zařízení připojené na sběrnici musí správně potvrdit přijatou zprávu. Potvrzení spočívá ve změně bitu ACK z hodnoty *recessive*, kterou vysílá vysílač na hodnotu *dominant*.[15]

2.2.6 Signalizace chyb

Každý uzel na sběrnici se z hlediska hlášení chyb může nacházet ve třech různých stavech: aktivní, pasivní a odpojený.[15]

Aktivní

Uzel, který se nachází z hlediska chyb v aktivním stavu se může aktivně podílet na komunikaci na sběrnici. Zároveň pokud detekuje libovolnou chybu v právě přenášené zprávě, vyšle na sběrnici aktivní příznak chyby. Aktivní příznak chyby je tvořen šesti po sobě jdoucími bity *dominant*. Tím dojde k poškození zprávy a porušení pravidla o vkládání bitu.[15]

Pasivní

Uzel v pasivní stavu hlášení chyb se podobně jako uzel aktivní účastní komunikace na sběrnici. Při detekování chyb však vysílá pouze pasivní příznak chyby. Ten se skládá z šesti po sobě jdoucích bitů *recessive*, takže nedojde ke znehodnocení vysílané zprávy.[15]

2.2.7 Základní typy zpráv

Ve specifikaci protokolu CAN jsou definovány čtyři typy zpráv:[15]

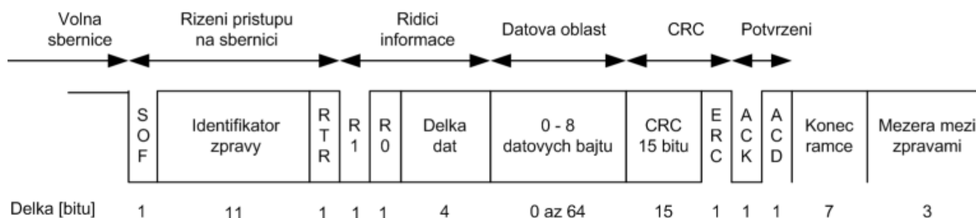
- datová zpráva,
- žádost o data,
- zpráva o chybě,
- zpráva o přetížení.

Datová zpráva tvoří základ komunikace a umožňuje vyslat zprávu dlouhou až 8 Bytů. Pro krátké zprávy, jako jsou například povely zapni/vypni však není třeba posílat žádná data. Takové binární příkazy mohou být obsaženy už v identifikátoru zprávy. Tímto principem se zvyšuje rychlost přenosu dat protokolu.[15]

Datová zpráva

Datové zprávy protokolu CAN jsou dvou různých typů. První z nich je definovaný ve specifikaci 2.0A a je označován jako **standardní formát**. Specifikace 2.0B definuje typ druhý, který je označován názvem **rozšířený formát** zprávy. V podstatě jediný rozdíl mezi oběma variantami je v délce identifikátoru, který je dlouhý 11 bitů ve standardním formátu zprávy, zatímco v rozšířeném formátu je dlouhý 29 bitů. Oba typy datové zprávy mohou být použity na stejné sběrnici, pokud je použitým řadičem podporován protokol CAN 2.0B.[15]

Struktura datové zprávy dle specifikace 2.0A je zobrazena na obrázku 2.4.

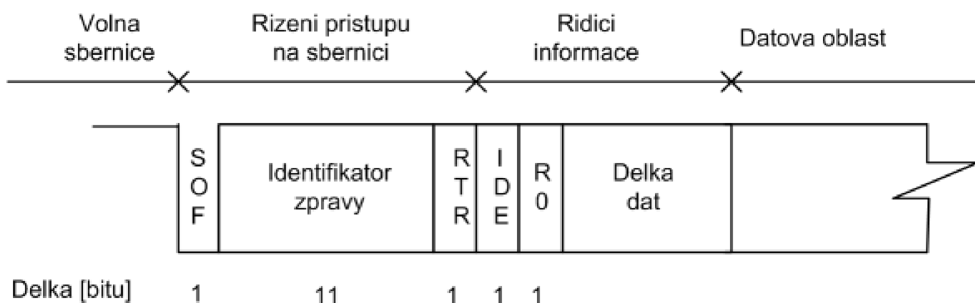


Obrázek 2.4: Datová zpráva dle specifikace CAN 2.0A
[18]

Význam jednotlivých částí datové zprávy:[15]

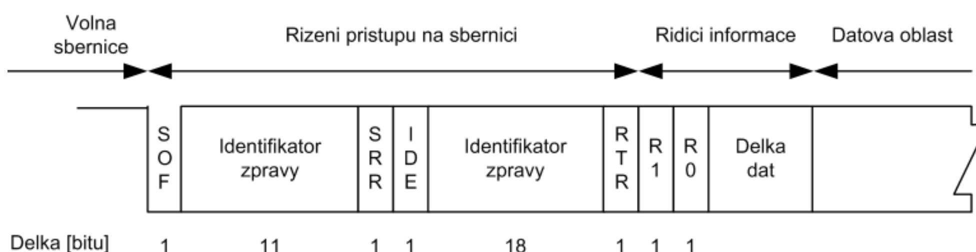
- (1b) SOF - *Start of Frame*, začátek zprávy, *dominant*
- (11b) identifikátor, určuje prioritu a význam zprávy
- (1b) RTR - *Remote Request*, odlišení, zda jde o datovou zprávu (*dominant*) nebo žádost o data (*recessive*)
- (2b) řídící pole *Control Field*, rezervováno
- (4b) délka datové zprávy
- (max. 8B) datová oblast *Data Field*
- (15b) CRC - *Cyclic Redundancy Check*, zabezpečovací kód
- (1b) ERC - oddělovač, *dominant*
- (2b) ACK - potvrzení, ACD - oddělovač
- (7b) konec zprávy *End of Frame*, *recessive*
- (3b) mezera mezi zprávami, *recessive*

Dle specifikace 2.0B jsou definovány dva typy datové zprávy - standardní a rozšířený. Standardní zpráva je převzata ze specifikace 2.0A. Jediným rozdílem je použití bitu *R1* pro indikaci, zda se jedná o rámec standardní nebo rozšířený. Podle CAN 2.0B se tomuto bitu říká *IDE - Identifier Extended.* Tento bit je v úrovni *dominant*, pokud se jedná o standardní rámec a v úrovni *recessive*, pokud se jedná o rámec rozšířený. Změna oproti specifikaci 2.0A standardního rámce je ilustrována na obrázku 2.5.[15]



Obrázek 2.5: Začátek datové zprávy standardního formátu podle 2.0B [19]

Rozšířený formát zprávy používá celkem 29 bitů jako identifikátor. Ten je rozdělen do dvou částí: 11 bitů jako u specifikace 2.0A a zbylých 18 bitů. Bit *RTR* je v rozšířeném formátu nahrazen bitem *SRR - Substitute Remote Request*, který má hodnotu *recessive*. Díky tomu je zajištěno, že při vzájemné kolizi standardního a rozšířeného formátu bude mít přednost formát standardní. Bit *RTR* samotný je přesunut na konec druhé části identifikátoru. Obrázek 2.6 ukazuje odlišnosti rozšířeného rámce od standardního.[15]



Obrázek 2.6: Začátek datové zprávy rozšířeného formátu podle 2.0B [20]

Žádost o data

Formát žádosti o data je velmi podobný datové zprávě. Jediným rozdílem je bit *RTR*, který je nastaven na hodnotu *recessive* a chybějící datová oblast. Uzel, jenž žádá o data nastaví identifikátor takový, který bude mít zpráva s daty, které požaduje. Pokud tedy zároveň bude jeden uzel žádat data a jiný je bude vysílat, pak se konflikt projeví až v bitu *RTR*, kde přednost získá uzel, jenž data vysílá, neboť tento bit nastaví na hodnotu *dominant*.[15]

Zpráva o chybě

Chybová zpráva signalizuje přítomnost chyby na sběrnici. Jakmile jí libovolný uzel odhalí, začne na sběrnici generovat chybový rámec. Ten se liší podle toho, v jakém stavu hlášení chyb se onen uzel nachází. Generuje tedy buď aktivní chybový rámec (šest bitů *dominant*) nebo pasivní (šest bitů *recessive*). Při generování aktivního chybového rámce je porušeno pravidlo o vkládání bitu a chybu začnou generovat i ostatní uzly. Hlášení chyb je pak indikováno superpozicí všech chybových příznaků, které uzly generují. Délka toho úseku je tedy 6 - 12 bitů. Po odeslání chybového příznaku začne uzel generovat úroveň *recessive* a zároveň kontrolovat stav sběrnice. Jakmile se na sběrnici objeví hodnota *recessive*, je chybová zpráva u konce a uzel vygeneruje 7 bitů *recessive*, který plní roli ukončení chybové zprávy. Následně může pokračovat běžná komunikace.[15]

Zpráva o přetížení

Posledním typem zprávy je zpráva o přetížení. Tato zpráva slouží k oddálení vyslání další datové zprávy či žádosti o data. Většinou je tato zpráva využita uzlem, jenž není schopen kvůli svému vytížení zpracovat další zprávy. Její struktura je podobná zprávě o chybě, ale vysílání může být zahájeno až po konci předchozí zprávy.[15]

2.3 Vývojový cyklus grafického rozhraní

Samotný software nahrávaný do jednotek je produkován externí koncernovou firmou, která ho v pravidelných intervalech zprostředkuje celému koncernu Volkswagen. Tomuto softwaru se také říká model. Dceřiné firmy, jakou je i Škoda Auto, si vlastní design grafického rozhraní (HMI) upravují sami. Tato práce probíhá po jednotlivých obrazovkách a nazývá se skinning. Právě pro Škoda Auto provádí skinning firma Digiteq Automotive. Skinneři, pracovníci vykonávající úpravu koncernem dodaného softwaru, se nestarají o funkčnost systému, ale o jeho vzhled. Jejich úkolem tedy je umístit správně všechny ikony, texty, tlačítka, check-boxy[10] nebo slidery[11] a zajistit, aby všechny tyto elementy správně reagovaly na dotek uživatele. Ten pak může mít ničím nerušenou interakci s infotainment systémem vozidla. Výsledkem práce skinnerů je tzv. skin, tedy konkrétní vzhled modelu.

Zda skinner správně odvedl svou práci, potvrdí testování obrazovky. Obrazovka je testována ve dvou kolech dvěma různými testery. Pokud někdo z nich zjistí chybu, nahlásí ji zpět skinnerovi a obě kola testovaní po opravě proběhnou znovu. Projde-li obrazovka oběma koly testování, její životní cyklus končí a práce na ní je hotova.

2.3.1 Životní cyklus obrazovky

Na obrázku 2.7 je zobrazen proces, ve kterém vzniká finální podoba konkrétní obrazovky.

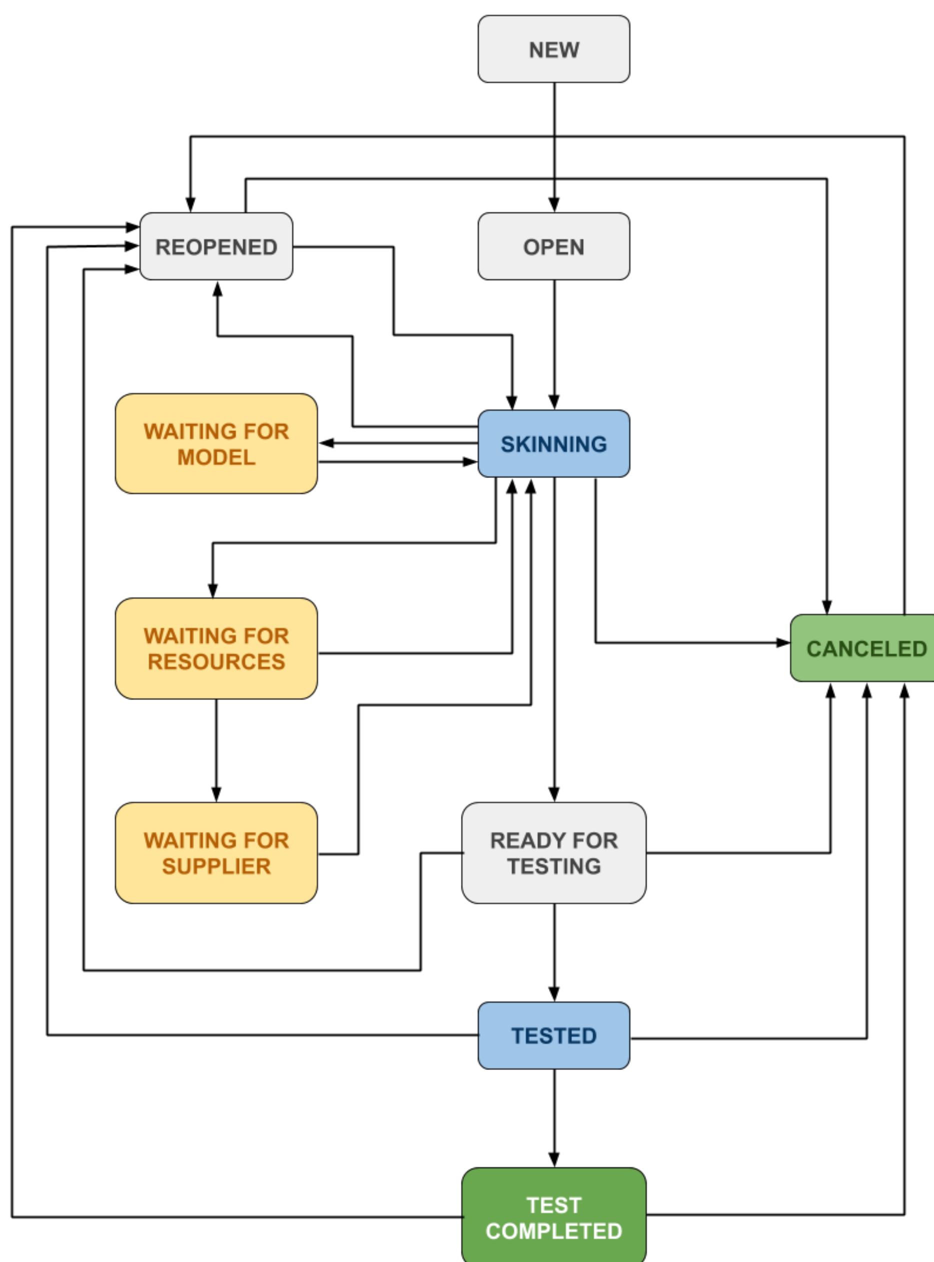
New

Na začátku svého životního cyklu se obrazovka nachází ve stavu **new**. Nová obrazovka často neobsahuje vůbec nic, kromě prázdných kontejnerů, jenž nejsou nijak prostorově umístěné.

Open

Ve stavu **open** čeká obrazovka na svého řešitele. Obrazovku v tomto stavu si může vyzvednout skinner a začít pracovat na její úpravě.

[10]Přepínače, viz kapitola 3.2.3 na straně 72.
[11]Posuvníky, viz kapitola 3.2.3 na straně 73.



Obrázek 2.7: Životní cyklus obrazovky

Skinning

Skinning je z pohledu obrazovky nejdůležitější část jejího životního cyklu. V tomto stavu na ní pracuje skinner, který jí upraví podle zadané specifikace do finální podoby.

Wainting for model

Ne vždy však skinner může obrazovku dokončit pouze vlastními prostředky. Jedním z případů, kdy to možné není, je špatně vytvořený model. Nevyhovující částí modelu může například být chybějící kontejner na ikonu či text, přestože ve specifikaci obrazovky tento element nesmí chybět. V tomto případě pošle skinner žádost externí koncernové firmě vytvářející software, aby chybějící element doplnila. V mezidobí přesune obrazovku do stavu **waiting for model**.

Wainting for resources

Waiting for resources je velmi podobným případem jako **waiting for model**, s tím rozdílem, že v tomto případě skinner nemá k dispozici zdroje, které jsou pro obrazovku určeny. To může být například text či ikona.

Wainting for supplier

Ve speciálním případě, kdy zdroj vytváří externí firma, se obrazovka dostává do stavu **waiting for supplier**. Takovým zdrojem muže například být design ikony. Tento stav je z pohledu obrazovky totožný s předchozím, neboť je třeba počkat na chybějící zdroj.

Ready for testing

V případě, že skinner dokončil svou práci, předá obrazovku do stavu **ready for testing**. V tomto stavu začíná obrazovka druhou důležitou část svého životní cyklu a čeká, než se jí ujme tester.

Tested

Úkolem testera je obrazovku otestovat, zda odpovídá zadané specifikaci a zda skinnerovi neunikla ani drobná chyba. Tester má dvě možnosti, jak obrazovku otestovat.

První z nich je navození obrazovky na **simulátoru** jednotky v počítači. Tento způsob přináší velkou výhodu ve snadném navození jakékoli obrazovky, avšak má svá úskalí. V simulátoru neběží stejný software jako na jednotce a není tedy samozřejmé, že funkční obrazovka v simulátoru bude funkční i ve skutečné jednotce.

Druhou možností testování obrazovky je její navození na **skutečné jednotce**, jež je připojena k tzv. testovacímu stavu. Tato situace je mnohem složitější, neboť skutečná jednotka pro svou správnou funkčnost vyžaduje velké množství jednotek, které automobil obsahuje. V testovacím stavu tak musejí být tyto jednotky přítomny nebo je třeba zprovoznit složitou simulaci, která jednotky zastupuje. V tomto případě je však jisté, že obrazovka vypadá přesně tak, jak bude vypadat i v produkčním automobilu.

Pokud tester nezjistí žádnou chybu na obrazovce, změní její stav na **tested**. V opačném případě její stav změní na **reopened** a přidá popis chyby, kterou objevil.

Test completed

Obrazovku ve stavu tested čeká ještě druhé kolo testování. To provádí jiný tester, než prováděl kolo první, aby se zabránilo případnému stejnému přehlédnutí chyby. Ve druhém kole testovaní musí být obrazovka testována pouze na skutečné jednotce. Pokud ani v tomto kole testování není zjištěna žádná chyba, obrazovka je prohlášena za hotovou a její stav se změní na **test completed**.

Reopened

V případě, že skinner na obrazovce z jakéhokoli důvodu přestane pracovat nebo na ní tester objeví chybu, je obrazovka přesunuta do stavu **reopened**, kde čeká, než se jí ujme nějaký skinner a bude pokračovat na její úpravě.

Canceled

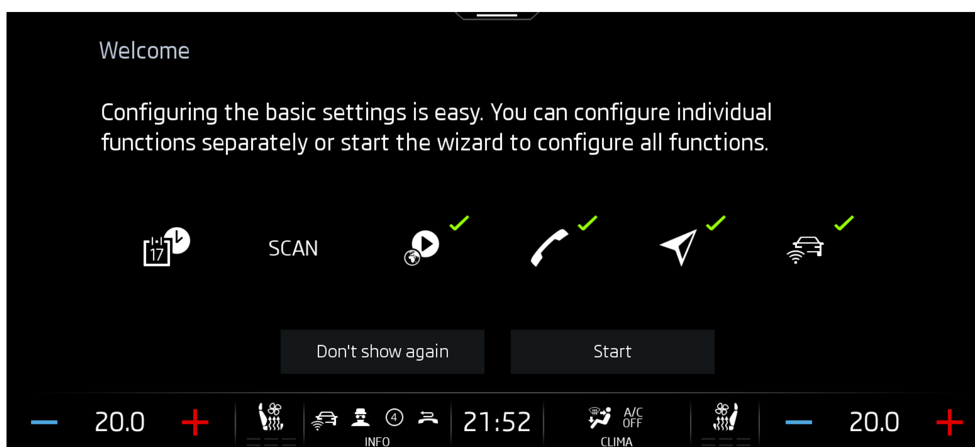
Posledním možným stavem obrazovky je stav **canceled**. Pokud je v průběhu vývoje rozhodnuto, že obrazovka se již implementovat nebude, pak se ocitne ve zrušeném stavu.

2.4 Motivace

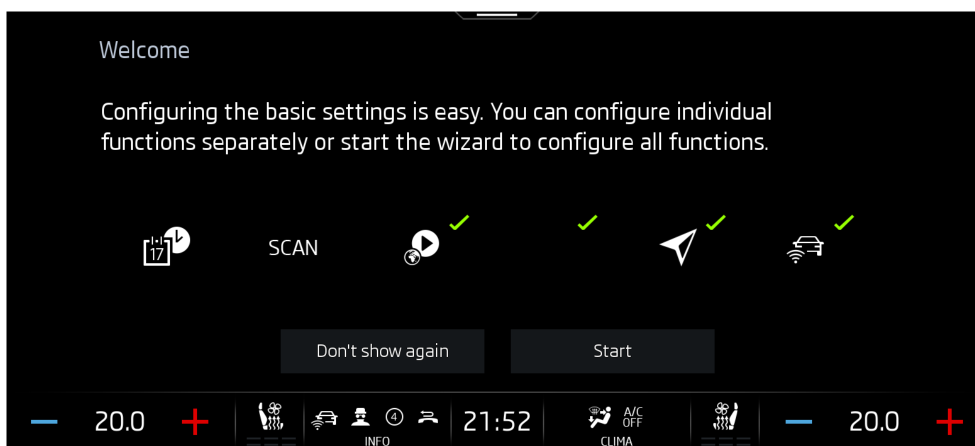
Samotný proces vytváření a testování jednotlivých obrazovek funguje spolehlivě. Obrazovka se postupnými kroky dostane ze stavu *new* do stavu *test completed*. Problém však nastává až po jejím vytvoření a dokončení testovacího procesu. V tuto chvíli má obrazovka status *test completed* a z pohledu skinnerů i testerů je práce na ní dokončena. Práce na celém grafickém rozhraní však pokračují dále a na pravidelné bázi se aktualizuje nová verze softwaru a HMI.

Kámen úrazu nastává ve chvíli, kdy z externí firmy VW[12] koncernu dorazí nový model softwaru. Stejně jako skinneři posílají do této firmy požadavky na změnu modelu, aby vyhovoval potřebám Škoda Auto, žádají i ostatní koncernové firmy jako např. Seat nebo samotný Volkswagen o změnu modelu. Právě například změny ostatních koncernových značek mohou přinést chybu v modelu pro Škoda Auto. V nejhorším případě se tato chyba projeví na obrazovce, která je již ve stavu *test completed*. Obrázky 2.8 a 2.9 ilustrují nechtěnou změnu nastalou po povýšení HMI na novou verzi.

[12] VW - Volkswagen.



Obrázek 2.8: Dokončená obrazovka před aktualizací modelu



Obrázek 2.9: Dokončená obrazovka po aktualizaci modelu

Objevit chybu na dokončené obrazovce není jednoduché. Na ukázce č. 2.9 je jasně patrné, že chybí ikona telefonu, ale chyba může nastat i v menším měřítku, například o pár pixelů špatně odsazený text nebo špatně zarovnaná ikona. Tato chyba je pak na první pohled jen těžko odhalitelná. Pokud je z pohledu všech pracovníků obrazovka v pořádku, pak pouze opětovné testování nebo náhoda může odhalit, že takováto chyba nastala. Vzhledem k počtu obrazovek a různým kombinacím jejich nastavení je velice časově náročné pravidelně manuálně testovat již dokončené obrazovky.

2.5 Zapojení automatického testování do vývoje GUI

Účelem automatického testování infotainment jednotky je odhalení chyb na již vytvořených obrazovkách. Zapojení automatických testů do životního cyklu obrazovky je znázorněno na obrázku 2.10. Obrazovka se tak dostane do nového stavu *automated testing*, který odhalí, zda na ní ve fázi jejího konečného stavu nenastala chyba.

Díky tomuto procesu je možné otestovat stovky obrazovek bez nutnosti fyzické přítomnosti testera. Ten po skončení testu pouze zkontroluje výsledek a popřípadě rozhodne, že chyba, kterou test objevil opravdu chybou je a změní stav obrazovky na *reopened*. V tomto stavu bude obrazovka čekat na opravu skinnerem.

2.5.1 Princip automatického testování

Již bylo zmíněno, že počet obrazovek v systému infotainmentu se pohybuje v řádu stovek. Většina obrazovek však nemá pouze jednu konečnou podobu, ale obsahuje prvky, které jsou nastavitelné uživatelem, nebo se dynamicky mění. To přináší tisíce různých kombinací, které se na displeji infotainmentu mohou zobrazit.

Jednotlivé obrazovky umí tester zkontrolovat komplexně. Po jejím navození na jednotce či v simulátoru, projde její možnosti a manuálně zkusí nastavit obrazovku do různých podob. Pro automatizovaný proces testování je však mnohem složitější obrazovku vyhodnotit jako správnou, je-li možné, že její konkrétní podoba se může měnit.

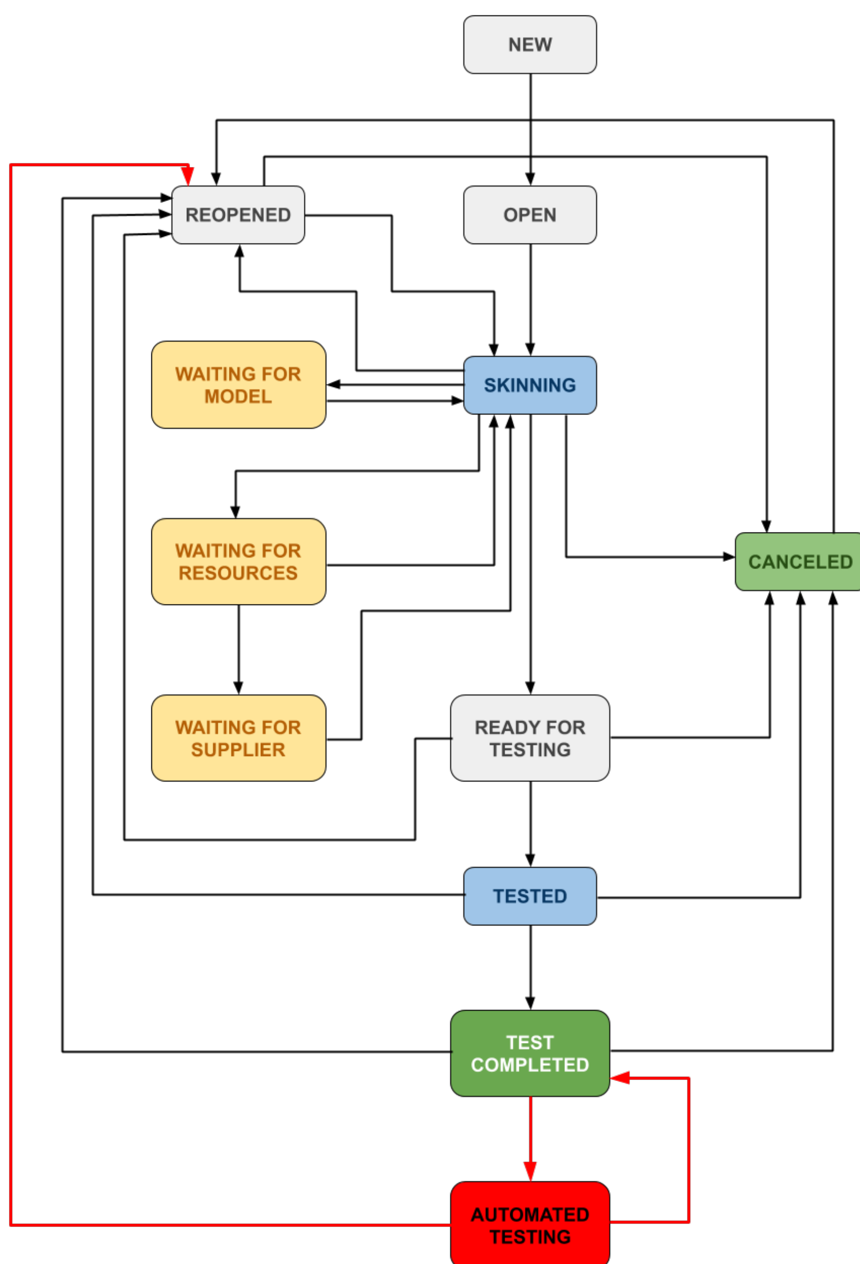
Proces průběhu celého testu vypadá následovně:

1. inicializace prostředí,
2. provedení jednotlivých testů,
3. generování HTML[13] reportu.

1. Inicializace prostředí

Prvním krokem celého testovacího procesu je úvodní inicializace. V ní proběhne nastavení globálních proměnných, jakými jsou např. cesta k archivu snímků, verze jednotky a HMI atd. Dalším důležitým bodem tohoto kroku je definování testovacích funkcí, které následně využívají konkrétní testcases.

[13] Hypertext Markup Language - značkovací programovací jazyk pro tvorbu webových stránek.



Obrázek 2.10: Životní cyklus obrazovky s automatickými testy

2. Provedení jednotlivých testů

Provedení samotných testů probíhá po tzv. kontextech. Jeden kontext obsahuje velké množství testcases, které představují jednotlivé kroky testu. Rozdělení do kontextů je zavedeno z důvodu přehlednosti výsledků.

3. Generování HTML reportu

Poslední fází testovacího procesu je generování výsledku. Pro snadné zobrazení je z výsledků všech testů vytvořen HTML report, kde jsou obsaženy všechny testované snímky a barevně označen výsledek jejich testu. Tuto zprávu s výsledky testu následně projde tester a musí rozhodnout, jak se bude dále postupovat s obrazovkou, kde nastala chyba.

Jeden testcase se skládá z následujících kroků:

1. navození obrazovky,
2. nastavení obrazovky,
3. uložení snímku obrazovky,
4. vyhodnocení správnosti,
5. uložení výsledku.

1. Navození obrazovky

Prvním krokem k otestování obrazovky je její zobrazení na displeji. V této fázi testovací prostředí vysílá jednotce signály po sběrnici CAN, které jednotka vyhodnotí jako doteky na displeji. Na tyto signály pak reaguje jednotka změnou svého stavu a tedy i zobrazením příslušné obrazovky. Tento princip testování je tedy tzv. uživatelský - jednotka je testována stejným způsobem, jako by ji testoval člověk.

2. Nastavení obrazovky

Po úspěšném navození obrazovky nastává fáze nastavení jejich prvků tak, aby bylo možné správně vyhodnotit výsledek. To znamená například zapnutí přepínačů a sliderů do konkrétní pozice, nastavení drop-down[14] na konkrétní hodnotu a podobně. Princip nastavování obrazovek je stejný jako při navození obrazovky - pomocí simulovaných doteků, na které jednotka adekvátně reaguje, jsou nastaveny potřebné elementy obrazovky.

3. Uložení snímku obrazovky

Ve chvíli, kdy je rozhraní infotainmentu na správné obrazovce a obrazovka správně nastavena, je možné pořídit její snímek. Aby se předešlo zbytečným chybám, není nutné pořizovat snímek celého displeje, ale pouze jeho konkrétního výřezu. Obrazovka se totiž dělí na určité části, které spolu přímo nesouvisí a jejich společné nastavování by bylo příliš zdlouhavé.

[14]Seznam možností k vybrání, podrobně v sekci 3.2.3 na straně 74.

Příkladem takového výřezu je tabbar (viz obr. 3.25), který se nachází u okraje obrazovky a zůstává stále stejný, ačkoli se obrazovka mění. Při testu takové obrazovky se tak pořizuje snímek bez tabbaru a ten je testován pomocí samostatného testcase.

Podle typu obrazovky je možné, že se snímků bude pořizovat více. Například obrazovky obsahující seznam položek je nutné fotit celé, pomocí scrollovací funkce. Tím se zjistí, zda je celý seznam zobrazován správně.

4. Vyhodnocení správnosti

Vyhodnocení správnosti snímku probíhá porovnáním s referenční verzí. Obrázky se porovnávají principem *pixel perfect*, tedy jejich shoda musí být na pixel přesná, aby bylo docíleno správného výsledku. Podle procentuálního počtu rozdílných pixelů se pak vyhodnotí, jaký výsledek má konkrétní snímek dané obrazovky. Pokud je rozdíl 0 %, je snímek vyhodnocen za správný. Pokud je rozdíl menší než 1 % je ve výsledné zprávě uvedeno, že na snímku je přítomna malá chyba. Rozdíl nad 1 % je vyhodnocen jako závažná chyba.

5. Uložení výsledku

Posledním krokem každého testcase je uložení výsledku, aby mohl být na konci testu zahrnut do finálního HTML reportu.

2.6 Testovací stav

Pojmem testovací stav se rozumí skříň, neboli rack, ve kterém je zapojena jednotka infotainmentu. Aby však jednotka mohla fungovat v alespoň omezeném provozu, musí rack obsahovat další jednotky automobilu, které s jednotkou infotainmentu komunikují přes sběrnici CAN. Rack, který tímto způsobem umožňuje spuštění a provoz infotainment jednotky se nazývá **testovací stav**.

Není vhodné a ani fyzicky možné, aby testovací stav obsahoval všechny jednotky reálného automobilu. Některé chybějící jednotky se dají nahradit pomocí simulace. Ta komunikuje s jednotkou také přes sběrnici CAN a vysílá zprávy, které by za normálních okolností produkovaly nahrazované jednotky.

Specialitu testovacího stavu pro automatické testování představuje přítomnost dvou počítačů. Ty se starají o simulaci chybějících jednotek, komunikaci s jednotkou infotainmentu a řízení celého testovacího stavu. V běžném stavu pro testování manuální není přítomen žádný počítač.

Obrázek 2.11 zobrazuje podobu testovacího stavu pro automatické testování.

Obrázek 2.11: Testovací stav pro automatické testování

2.6.1 Hardware

Jak již bylo zmíněno, testovací stav pro automatické testování obsahuje dva stolní počítače. Ty však nejsou zdaleka jedinými komponentami celého stavu. V této části jsou popsány všechny složky testovacího stavu a jejich datové propojení ke zdroji napájení, které je zobrazeno na obrázku 2.23. Obrázek 2.24 následně zobrazuje datové propojení jednotlivých komponent stavu.

Windows PC

Obrázek 2.12: Windows PC
[21]

Prvním z počítačů a zároveň mozkem celého testovacího stavu je počítač s operačním systémem Windows. Jeho hlavním úkolem je řízení celého stavu a spouštění jednotlivých testů.

Propojení počítače s okolním světem zabezpečují tři různé ethernetové kabely:

1. připojení k firemní síti, a skrze ní také k internetu (WAN[15]);
2. připojení k lokální síti, sloužící pro komunikaci mezi komponentami testovacího stavu (LAN[16]);
3. připojení k USB[17] hub, pomocí kterého jednotka posílá trace, tedy nepřetržitý proud zpráv o své aktivitě.

Kromě těchto síťových prvků je k počítači připojen přímo i zdroj ZUP2020, umožnující dálkové ovládání. Samotný počítač standardně ovládá klávesnice, myš a jako jeho grafický výstup slouží monitor.

Black Box PC

Druhým počítačem testovacího stavu je tzv. **Black Box PC**. Pod tímto názvem si lze představit černou skříňku, u níž není zřejmé jak svou práci vykonává. V případě Black Box PC tomu tak ale není. Název počítač dostal podle barvy a svého výrobce. Na Black Box PC běží operační systém Linux s velkým počtem implementovaných skriptů, starajících se o posílání CAN zpráv infotainment jednotce.

[15]Wide Area Network - globální počítačová síť.
[16]Local Area Network - lokální počítačová síť.
[17]Universal Serial Bus - univerzální sériová sběrnice.

Obrázek 2.13: Black Box PC
[22]

Propojení počítače a stavu zajišťují opět ethernetové kabely, tentokrát dva:

1. připojení k lokální síti, která slouží pro komunikaci mezi komponentami testovacího stavu (LAN);
2. připojení ke CAN/LAN switch, skrze který probíhá komunikace s jednotkou pomocí CAN protokolu.

Black Box PC je ovládán pomocí příkazů, které mu vysílá druhý počítač v lokální síti. Není k němu připojena myš s klávesnicí ani monitor. Je-li třeba, do počítače se lze připojit vzdáleně SSH[18] klientem

ZUP2020

Obrázek 2.14: Zdroj ZUP2020 (ilustrační obrázek)
[23]

Nastavitelným zdrojem elektrického napětí je **ZUP2020**. Jeho velkou přednost představuje možnost dálkového ovládání. ZUP2020 je přímo propojený pomocí USB s Windows PC, které se stará o jeho řízení. Zdroj napájí výhradně jednotky automobilu, což přináší výhodu v podobě možnosti je vzdáleně restartovat pomocí přerušení dodávky elektrické energie.

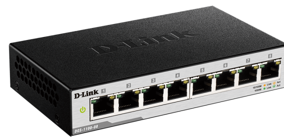
[18]Secure Shell - zabezpečený komunikační protokol.

Zdroj

Obrázek 2.15: Zdroj bez možnosti dálkového ovládání
[24]

ZUP2020 není jediným zdrojem elektrického napětí integrovaným ve stavu. Druhý zdroj však není možné dálkově ovládat a ten tak stabilně dodává napětí 12V komponentám, které takové provozní napětí vyžadují. Mezi ně patří například převodníky CAN/LAN nebo Grabber, jenž jsou popsány na stranách 49, respektive 52 .

LAN switch



Obrázek 2.16: Switch lokální sítě stavu
[25]

V celém testovacím stavu se nachází dva switche. První z nich se stará o propojení lokální sítě celého stavu a tím pádem o správnou komunikaci mezi jednotlivými počítači a Grabberem.

CAN/LAN převodníky

Zprávy posílané Black Boxem po ethernetovém kabelu je nutné na sběrnici CAN přeložit. O to se starají tzv. **CAN/LAN převodníky**. Ve stavu se nacházejí dva a každý z nich má na starost převod zpráv pro různé sběrnice CAN. Jeden z převodníků má na starost převod zpráv a jejich odeslání na Private CAN a Infotainment CAN, které slouží pro komunikaci jednotek poskytujících uživateli ovládací a zábavní funkce, jakou je například poslech hudby. Druhý z převodníků překládá zprávy na Komfort CAN, kde probíhá komunikace jednotek komfortních systémů, jakou je například klimatizace.

CAN/LAN převodníky byly vyvinuty ve firmě Digiteq Automotive, ale jejich konkrétní implementace není předmětem této práce.

CAN/LAN switch

Obrázek 2.17: Switch směrující zprávy na CAN/LAN
[26]

Druhým switchem je tzv. CAN/LAN switch. Tento real-time switch má za úkol přesměrovat zprávy, které posílá Black Box PC na správné CAN/LAN převodníky.

Propojovací panel

Z CAN/LAN převodníků putuje zpráva již v podobě CAN protokolu směrem ke Gateway a do jednotky infotainmentu. Než se tam dostane, projde ještě skrze rozbočku CAN sběrnic, která se nachází na propojovacím panelu. Toto zařízení je čistě fyzické rozvětvení jednotlivých CAN sběrnic do více přípojek. Z tohoto místa lze ke sběrnicím CAN připojit externí zařízení pro monitoring komunikace a pro simulaci.

Propojovací panel také rozvětvuje napájení, které jednotce poskytuje zdroj ZUP2020. V tomto místě jsou připojeny i další jednotky související s činností infotainmentu, a sice Gateway (viz str. 51).

Posledním konektorem, který se nachází na propojovacím panelu je diagnostická zásuvka. Přes tuto zásuvku se pomocí speciálního zařízení a softwaru lze jednotku tzv. *nakódovat*. Kódování jednotky nastaví funkce, které jsou uživateli zpřístupněny. Pro účely testování je vhodné mít možnost kódování dostupnou v testovacím stavu. Pomocí stejného principu a diagnostické zásuvky se jednotky kódují i v produkčních vozech.

Jednotka infotainmentu

Nejdůležitější součástí stavu na automatické testování je pochopitelně testovaná **jednotka infotainmentu**. Jejím úkolem v automobilu je poskytnout uživateli snadné a komfortní nastavení celého vozu a zobrazení všech potřebných a užitečných informací.

S testovacím stavem je propojena svazkem všech CAN sběrnic a napájení. Tomuto svazku se v koncernu VW říká Quadlock. Dále komunikuje pomocí dvou datových kabelů. Jedním z nich posílá obraz na displej infotainmentu. Druhým je propojena s USB hubem, ke kterému lze například připojit externí zařízení s hudbou. Sama přes tento datový kanál posílá trace v reálném čase.

Obrázek 2.18: Jednotka infotainmentu (ilustrační obrázek) [27]

Testovací stav pro automatické testy je uzpůsoben pro dvě různé platformy jednotek infotainmentu. První, straší z nich, je jednotka s označením MQB, jež může mít připojený 8″ nebo 9″ displej. Tuto jednotku je možné najít například ve vozech Kamiq, Karoq nebo Scala. Druhou jednotkou je systém spojený s 10″ displejem s označením MQB_37W. Tato jednotka poskytuje infotainment služby uživatelům nové Škody Octavie 4. generace. Obě tyto jednotky mají stejný quadlock konektor, dva datové výstupy pro obraz a USB. V jednu chvíli však může být připojena pouze jedna jednotka.

Gateway

Obrázek 2.19: Gateway (ilustrační obrázek) [28]

Jednotka, které se v automobilu říká **Gateway**, slouží k přeposílání zpráv mezi jednotlivými sběrnicemi. Toto zařízení zajišťuje přenos mezi různými komunikačními kanály a tudíž i mezi jednotkami, které spolu nejsou přímo propojeny. Gateway přeposílá zprávy nejen mezi různými sběrnicemi CAN, ale také mezi různými komunikačními protokoly, jakými jsou CAN, LIN[19] nebo Ethernet.

Pro každou platformu infotainment jednotky je určena konkrétní jednotka Gateway. Ta ovšem, na rozdíl od jednotky infotainmentu, nemá skrze platformy stejný konektor. V testovacím stavu tak je pro obě platformy jeden kabelový svazek pro připojení Gateway. Ve schématech zapojení 2.23 a 2.24 je znázorněna pouze jedna jednotka Gateway. Pro zajištění správné funkčnosti konkrétní infotainment jednotky může být připojena pouze jedna adekvátní Gateway.

[19]Local Interconnect Network - sériový komunikační protokol používaný v automobilech.

USB hub

Rozbočka signálu USB, skrze níž jednotka komunikuje s externími zařízeními a posílá záznam o své aktivitě a stavu, se nazývá **USB hub**. Pomocí redukce je k tomuto zařízení připojen ethernetový kabel zachycující ono tracování. Tento stream dat putuje skrze lokální síť do Windows PC.

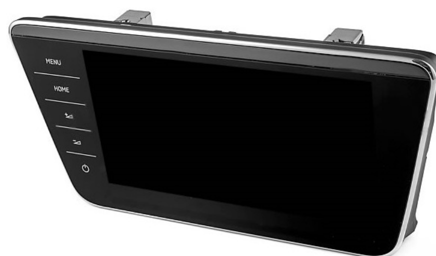
Grabber

Obrázek 2.20: Grabber
[29]

Další velmi důležitou součástí testovacího stavu je **Grabber**. Toto zařízení vyvíjené firmou Digiteq Automotive slouží k zachytávání obrazu vysílaného do displeje jednotkou infotainmentu. Díky tomuto zařízení je možné vyfotit konkrétní obrazovku, jež se právě zobrazuje na displeji jednotky a následně softwarově vyhodnotit, zda se na ní nenachází žádná chyba.

Grabber je posledním zařízením, připojeným pomocí ethernetového kabelu do lokální sítě stavu. Tímto způsobem posílá data do Windows PC, ze kterého lze Grabber vzdáleně nastavit.

ABT



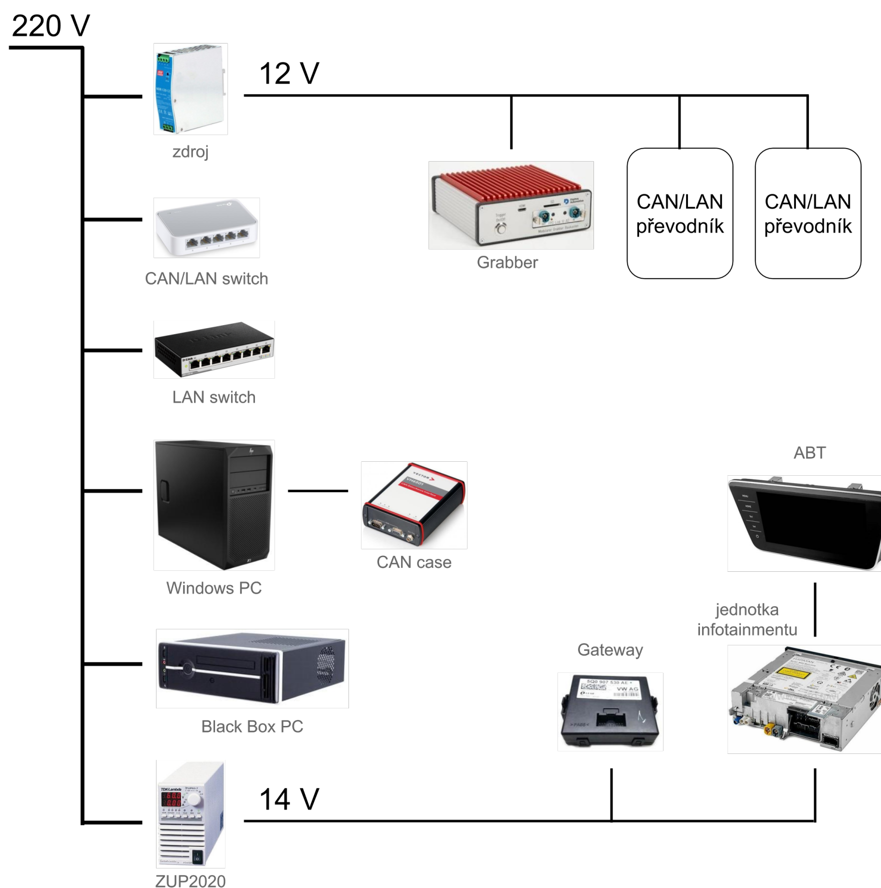
Obrázek 2.21: Displej infotainmentu - platfroma MQB, 9″
[30]

Anzeigebedienteil neboli ABT je německý výraz pro zobrazovací a ovládací panel. Tímto názvem se označuje displej, který slouží jako přímé rozhraní mezi jednotkou infotainmentu a uživatelem. Pomocí obrazu mu jednotka předává informace a uživatel doteky systém ovládá.

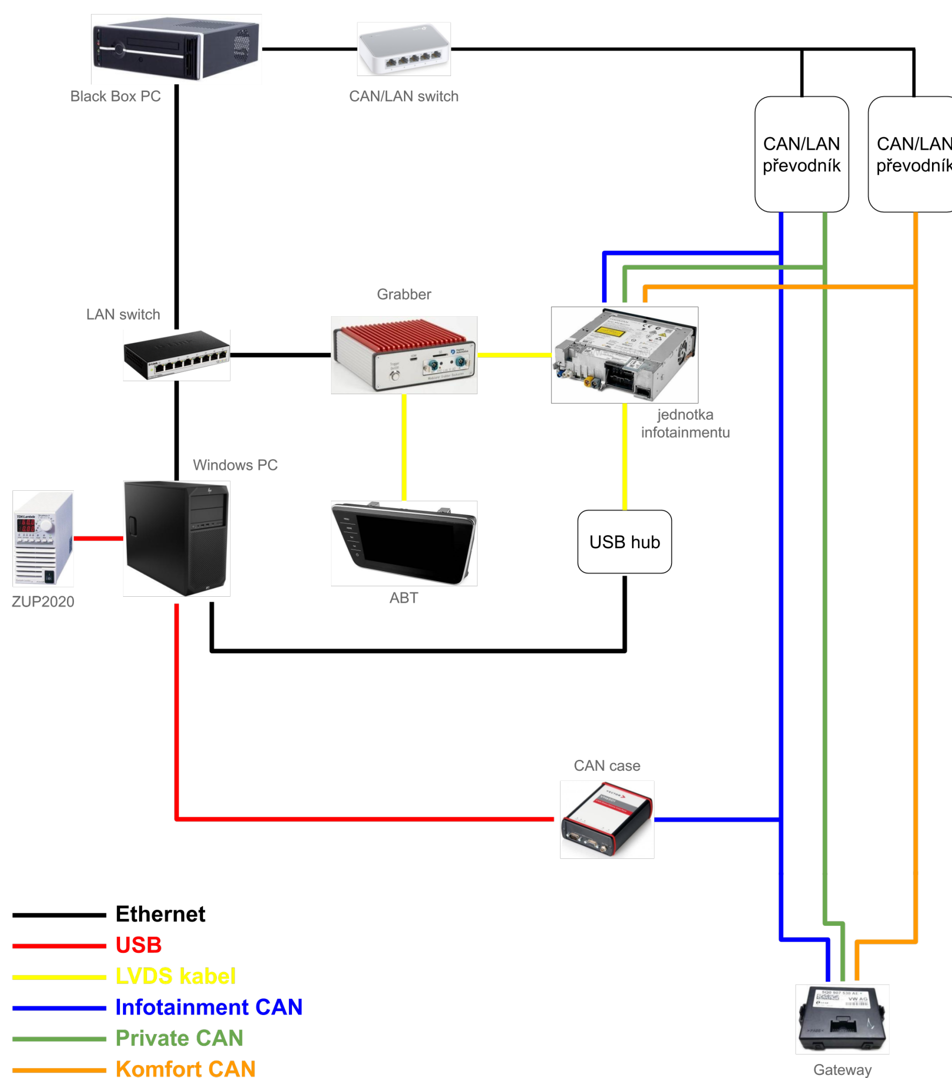
CAN case

Obrázek 2.22: CAN case
[31]

Firma Vector vyrábí speciální produkt s názvem **CAN case**, který spolu se softwarem CANoe poskytuje možnost simulovat různé jednotky automobilu a jejich CAN komunikaci. Toto zařízení je připojené ke CAN sběrnici skrze rozbočovač CAN sběrnic. CAN case ovládá software CANoe běžící na Windows PC, se kterým je CAN case propojen pomocí USB.



Obrázek 2.23: Schéma zapojení hardwaru stavu ke zdroji el. energie



Obrázek 2.24: Schéma datového propojení testovacího stavu

2.6.2 Software

Tato sekce popisuje softwarové vybavení jednotlivých komponent testovacího stavu.

Black Box PC

Na operačním systému Linux v Black Box PC jsou implementovány skripty na posílání zpráv ve správném tvaru a na správnou CAN sběrnici. Pomocí druhého počítače lze tomuto stroji zadat příkaz k odeslání konkrétní zprávy, ale i k posílání nějakých zpráv periodicky. Všechny tyto úkony zvládá Black Box PC v reálném čase.

Tyto skripty byly vytvořeny firmou Digiteq Automotive, ale jejich konkrétní podoba není předmětem této práce.

Jednotka infotainmentu

Software běžící na jednotce infotainmentu se stará o uživatelské nastavení vozu. Účelem celého procesu testování jednotky je odhalení chyb v jejím grafickém výstupu.

Software jednotky infotainmentu je vyvíjen v koncernu VW a jeho konkrétní podoba není předmětem této práce.

Windows PC

Většina softwarového vybavení testovacího stavu se nachází na Windows PC. Tento počítač slouží k celkovému ovládání testovacího stavu. Následující odstavce popisují pět nejdůležitějších programů tohoto počítače.

1. GIT

Pro verzování kódu a dalších zdrojových souborů je využit standardní verzovací nástroj GIT. Spolu s uživatelským rozhraním Git Extensions poskytuje snadné verzování, popřípadě řešení kolizí. Verzovány jsou všechny zdrojové soubory funkcí a testcases, ale i databáze obrázků.

2. Grimmer

Jednoduchým programem zobrazující aktuální obraz, procházející skrze Grabber do ABT, je Grimmer. Pomocí tohoto programu lze manuálně vytvořit snímek obrazovky nebo definovat novou ořezovou oblast. Pro samotný běh testu není tento program důležitý.

Grimmer je vyvíjen firmou Digiteq Automotive.

3. CANoe

Díky softwaru produkovaným firmou Vector, je možné testovat jednotku infotainmentu mnohem komplexněji. Bez simulace jednotek, které ve stavu nejsou fyzicky připojeny, ale v automobilu ano, nezobrazuje jednotka zdaleka tolik obrazovek. Navíc je velmi běžné, že je-li obrazovka bez simulace viditelná, pak nemá zobrazeny zdaleka všechny elementy, nebo nejde nastavit do potřebné podoby. Simulace tedy markantně zvýší počet obrazovek, které je možno testovat v jejich konečné podobě.

Konkrétní podoba programu CANoe a nastavování simulace není předmětem této práce.

4. TestAut

Hlavní program řídící celý proces testování se nazývá TestAut. Tento program spustí konzoli jazyka TCL a podle vybrané platformy nastaví a spustí správné skripty, kterými Black Box PC generuje CAN zprávy. Dále při prvním spuštění nastaví TestAut zdroj ZUP2020, aby dodával jednotkou požadované elektrické napětí. Intuitivním ovládáním následně program umožňuje spouštění testů.

TestAut vyvinula firma Digiteq Automotive.

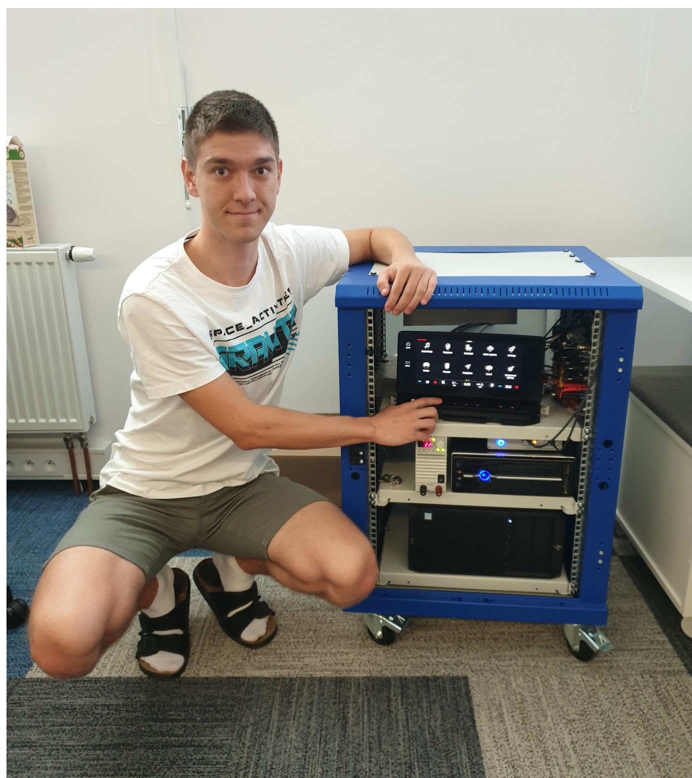
5. ImageMagick

Volně dostupný software pro práci s obrázky využívaným automatickými testy nese název ImageMagick. Pomocí tohoto programu je možné obrázky porovnávat, vyhledávat v jiných obrázcích nebo měnit jejich velikost. Další funkčností programu představuje například vytvoření obrázku podle zadaného fontu o konkrétní velikosti a barvě.[32] To všechno jsou velmi důležité funkčnosti, které jsou hojně využívány při automatickém testování, neboť je testována grafická stránka jednotky infotainmentu.

Kapitola 3

Realizace

Tato kapitola představuje způsob, jakým byly testy realizovány a prováděny. Vývoj celého projektu v oddělení grafického vývoje firmy Digiteq Automotive začal na platformě MQB_37W do nové Škody Octavie 4. generace. Z tohoto důvodu jsou konkrétní ukázky a výsledky automatických testů uvedeny právě s výstupy z platformy MQB_37W. V současné době je rozpracováno implementování funkcí i pro platformu MQB, jež je kompatibilní s testovacím stavem.



Obrázek 3.1: Testovací stav s 10″ displejem platformy MQB_37W

3.1 Tracing

Testovaná jednotka infotainmentu se ovládána pomocí zpráv, které simulují doteky na konkrétních souřadnicích displeje. Tyto zprávy jsou jednotce posílány přes adekvátní sběrnici CAN. Jednotka na tyto zprávy reaguje stejně, jako by se uživatel skutečně displeje dotkl a podle toho adekvátně změní obraz, který na displeji zobrazuje. Grabber tento obraz dokáže zachytit a zpracovat. Obraz samotný však nestačí jako zpětná vazba, která by určila aktuální obrazovku, na které se systém infotainmentu nachází.

Jedinou informací jinou, než je obrazový výstup, kterou jednotka infotainmentu poskytuje je **trace**. Tracing, nebo také česky trasování, je způsob, jakým jednotka poskytuje informace o svém stavu a všech podstatných událostech. Jedná se o nepřetržitý stream dat, které jednotka posílá pomocí svého druhého datového kabelu, tedy do USB hubu. K tomu je připojena pomocí ethernetového kabelu síťová karta počítače Windows PC, starající se o příjem tohoto proudu dat.

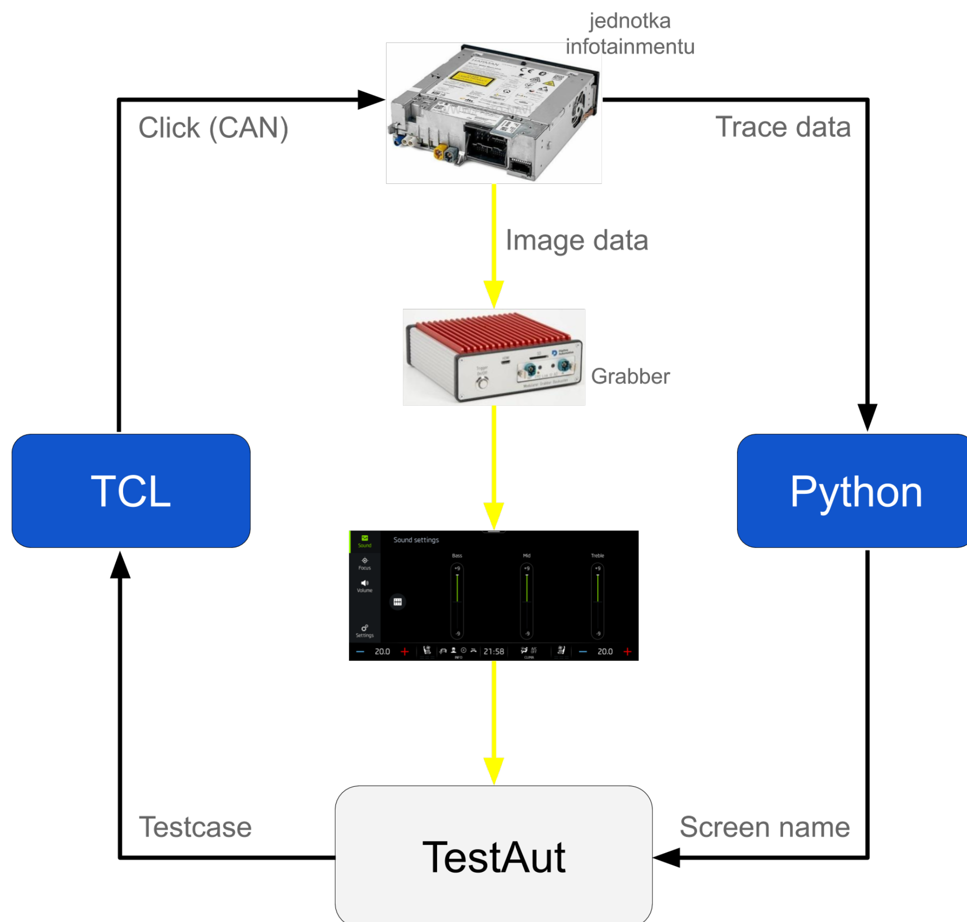
Díky tomuto kanálu je možné z jednotky získat zpětnou vazbu o jejím stavu. To je fakt, který lze s výhodou vyžít ve prospěch automatického testování. Jednou z informací, které jednotka v rámci trace posílá, je totiž i jméno obrazovky, kterou právě zobrazuje. Díky této informaci tak může testovací prostředí kontrolovat, zda po vykonání příkazů, které obrazovku na infotainmentu navodí, se jednotka skutečně na dané obrazovce nachází. Tato kontrola se při provádění automatických testů zavedla z důvodu náhodných chyb při vykonávání příkazů dotyků. Ve výjimečných případech se může stát, že se funkci, ač provede dotek na správné souřadnici, nepodaří projít do chtěného kontextu.

Obrázek 3.2 ilustruje proces získání názvu obrazovky z trace. Jednotka reaguje na povely, jež obdrží jako zprávu na sběrnici CAN. Adekvátně změní obraz, který zachytí Grabber a v trace odešle informaci o změně obrazovky. Název obrazovky z trace získá program napsaný v jazyce Python.

3.1.1 Python

Zpracování trace je na testovacím stavu prováděno pomocí skriptu napsaném v jazyce Python. Tento skript čte stream přicházejících dat z jednotky infotainmentu a zpracovává jej. Pokud narazí na informaci o názvu obrazovky, pak ho uloží do textového souboru `log_screenname.txt`. Čtením tohoto souboru uvnitř funkcí implementovaných v TCL lze real-time zjistit, na které obrazovce se jednotka infotainmentu nachází.

Konkrétní podoba tohoto skriptu není stejně jako konkrétní podoba trace důležitá pro tuto práci. V principu se jedná o parsing přicházejících dat.



Obrázek 3.2: Ovládání jednotky se zpětnou vazbou

3.2 Automatické testy

Obrázek 3.3: 10″ displej platformy MQB_37W
[33]

3.2.1 Kód

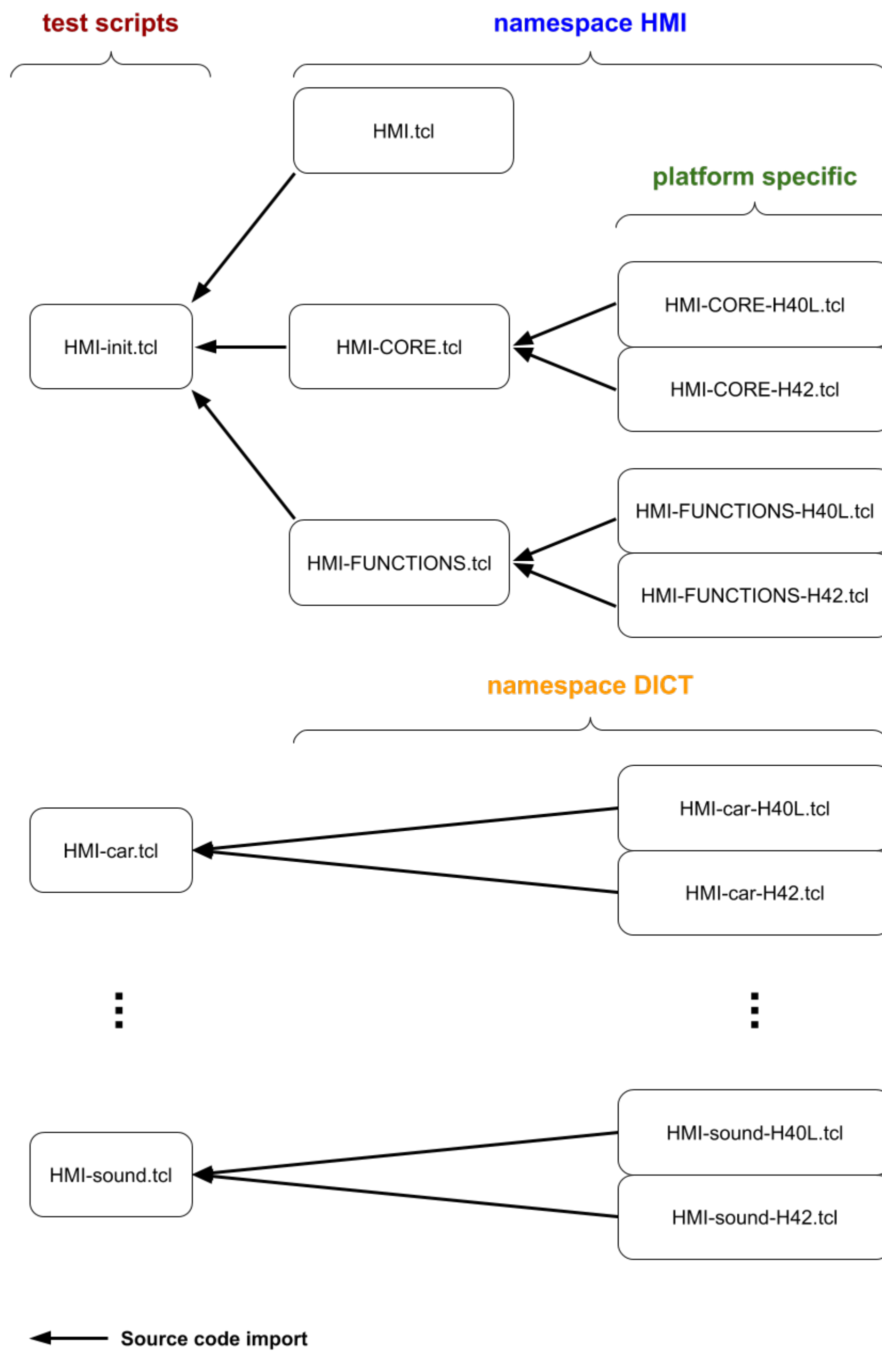
Zdrojový kód knihoven a testů pro automatické testy HMI je uložen v souborech s příponou `.tcl`, což je koncovka běžná pro soubory psané v jazyce TCL. Rovněž pro snadnou orientaci začínají jména souborů zkratkou `HMI`. Schéma na obrázku 3.4 zobrazuje provázanost zdrojových souborů.

Namespaces

Strukturu kódu tvoří dva hlavní jmenné celky, nebo-li namespaces. Teoretické informace o namespaces se nacházejí v kapitole 2.1.2 na staně 20.

HMI

Namespace HMI obsahuje všechny důležité proměnné a funkce pro testování HMI infotainment jednotky a pro tvorbu jednotlivých testcases. Jeho definice je rozdělena do různých souborů, podle jejich významu.



Obrázek 3.4: Schéma provázanosti zdrojových souborů

Tabulka 3.1: Zdrojové soubory

Soubor	**popis**
HMI.tcl	hlavní soubor s definicí namespace HMI, globálními proměnnými, funkcemi na focení obrazovek a vytváření reportu
HMI-CORE.tcl	soubor se základními funkcemi jako Press nebo Click
HMI-FUNCTIONS.tcl	soubor s funkcemi pro nastavení HMI, využívá CORE funkce
HMI-init.tcl	inicializační test, musí proběhnout před prvním spuštěným testem
HMI-*kontext*.tcl	soubory s definicí testu pro daný kontext
HMI-CORE-*platforma*.tcl	platformně specifické CORE funkce
HMI-FUNCTIONS-*platforma*.tcl	platformně specifické funkce
HMI-*kontext-platforma*.tcl	platformně specifické testcases

HMI.tcl je hlavním souborem obsahujícím definici jmenného prostoru HMI. V tomto souboru jsou definovány všechny globální proměnné souboru, které pak následně využívají funkce ze stejného prostoru. Dále tento soubor obsahuje definici funkcí pro tvorbu snímků obrazovky. Funkcí pro pořizování snímků obrazovky je více, neboť různé typy obrazovek vyžadují různé zacházení při pořizování snímků.

Základní funkcí pro focení obrazovky je `TakeScreen`, která uloží snímek obrazovky a připojí ho k reportu. Ostatní funkce pořizující snímky obrazovky tuto funkci využívají. Takovou funkcí je i `TakeClassic`, která vytvoří jeden snímek obrazovky a je-li třeba, tak i snímek drop-down, kterých může na obrazovce být více. Stejně tak funkce `TakeScrollScreens` vytváří snímky drop-downs, má-li to parametrem určeno. Na rozdíl od `TakeClassic` však funkce `TakeScrollScreens` pořídí všechny snímky scrollovatelné obrazovky. Dalšími funkcemi, které pořizují snímky obrazovek jsou například `TakeTabbar`, která fotí všechny pozice tabbaru nebo `TakeDropDowns`, jenž využívají ostatní funkce pro pořízení snímků všech drop-downs. Každý testcase má určený mód, podle kterého se vybere adekvátní funkce pro pořízení snímků.

Tyto funkce jsou volány funkcí `MasterWorker`, která podle parametrů testcase zavolá adekvátní funkci. Další velmi důležitou funkcí definovanou v souboru HMI.tcl je `Report`. Podle názvu je zřejmé, že tato funkce obstará přidání aktuálního pořízeného snímku do výsledného reportu testu. Kromě toho, do reportu přidá také referenční verzi této obrazovky, rozdílový obrázek a vyhodnotí, zda se od sebe liší.

Soubory **HMI-CORE.tcl** a **HMI-FUNCTIONS.tcl** obsahují definice funkcí, jenž jsou součástí namespace HMI. HMI-CORE.tcl obsahuje základní funkce, které jsou potřeba pro stavbu složitějších funkcí definovaných v HMI-FUNCTIONS.tcl a samotném HMI.tcl. V HMI-CORE.tcl jsou tak definované funkce jako `ClickAt` pro kliknutí na konkrétní souřadnice nebo `ImgCompare` pro porovnání dvou snímků obrazovky. Naopak HMI-FUNCTIONS.tcl obsahuje složitější funkce jako `SetAllCheckBoxes`, která nastaví všechny checkboxy na celé scrollovatelné obrazovce nebo `DropDownSetAll`, jež nastaví všechny drop-downs přítomné na displeji.

Posledními soubory s definicemi funkcí jmenného prostoru HMI jsou **HMI-CORE-H40L.tcl**, **HMI-FUNCTIONS-H40L.tcl**, **HMI-CORE-H42.tcl** a **HMI-FUNCTIONS-H42.tcl**. Tyto soubory obsahují platformně specifické funkce pro konkrétní verzi jednotky infotainmentu. Soubory s označením `H40L` odpovídají platformě MQB s displejem o rozměru 9″, kdežto soubory s označením `H42` odpovídají platformě MQB_37W. V obou verzích souboru HMI-CORE-*platforma*.tcl jsou tak definovány stejné funkce, které se však liší svým tělem. Např. funkce `PressAt` vytvoří správnou podobu press zprávy a pošle jí jednotce po sběrnici CAN. Konkrétní podoba zprávy se však napříč platformami liší, a proto je třeba pro konkrétní platformu použít správnou podobu této zprávy. To zajišťuje platformně nezávislý soubor HMI-CORE.tcl, který při svém provádění naimportuje správnou platformně závislou knihovnu funkcí HMI-CORE. Stejný princip je použit u knihoven funkcí.

DICT

Druhým ze jmenných prostorů je namespace s názvem **DICT**. Tento jmenný prostor sdružuje jednotlivé testcases, které se prochází a vykonávají postupně po kontextech. Jedním z takových kontextů je například klimatizace. Test definovaný v souboru HMI-climate.tcl projde všechny testcases, které patří do kontextu klimatizace. Rozdělení testů obrazovek do kontextů je učiněno z důvodu přehlednosti výsledků. Ve výsledkové zprávě budou pohromadě všechny snímky jednoho testu. Každý spuštěný test v reportu představuje jednou tabulku s obrázky a výsledky.

Než test začne procházet jednotlivé testcases daného kontextu, naimportuje tyto testcases do jmenného prostoru DICT. Z důvodu odlišnosti platforem jsou testcases opět importovány z platformě závislých souborů, pro kontext klimatizace to je buď **HMI-climate-H40L.tcl** nebo **HMI-climate-H42.tcl**.

V kódu je využito i dalších jmenných prostorů, jako **V4T**, **OCR** nebo **CANSIMUL**. Tyto namespaces mají implementované funkce obstarávající zachycení obrazu, práci s uloženými obrázky a ovládání Blax Box PC. Jsou součástí již implementovaných knihoven programu TestAut a byly vyvinuty ve firmě Digiteq Automotive.

Testcases

```
#Ukázka testcase

dict set views CARSETUP_MAIN_INTERIOR context "car"
dict set views CARSETUP_MAIN_INTERIOR area "SCREEN_INSIDE"
dict set views CARSETUP_MAIN_INTERIOR mode "classic"
dict set views CARSETUP_MAIN_INTERIOR screenName "CARSETUP_MAIN"
dict set views CARSETUP_MAIN_INTERIOR path {
  {HMI::EnterStageArea "CAR"}
  {HMI::EnterTab "1"}
}
dict set views CARSETUP_MAIN_INTERIOR prepare {
  {HMI::TextClick "DISPLAY" "Interior" 27 70}
  {after 1000}
}
```

Pro jednotlivé testcases je využita struktura slovníku TCL. Teoretické informace o slovníku lze najít v kapitole 2.1.2 na straně 19. Slovník s testcases nese název `views`. Slovník views obsahuje každý testcase a zároveň má každý konkrétní testcase přiřazeno několik důležitých proměnných. Významy jednotlivých proměnných jsou popsány v tabulce 3.2, jež některé proměnné uvádí v uvozovkách. To je z důvodu, že tyto proměnné nemusí být uvedeny u každého testcase. Např. při neuvedení proměnné `prepare` se obrazovka vyfotí, aniž by na ní bylo potřeba cokoli nastavit.

Tabulka 3.2: Atributy testcase

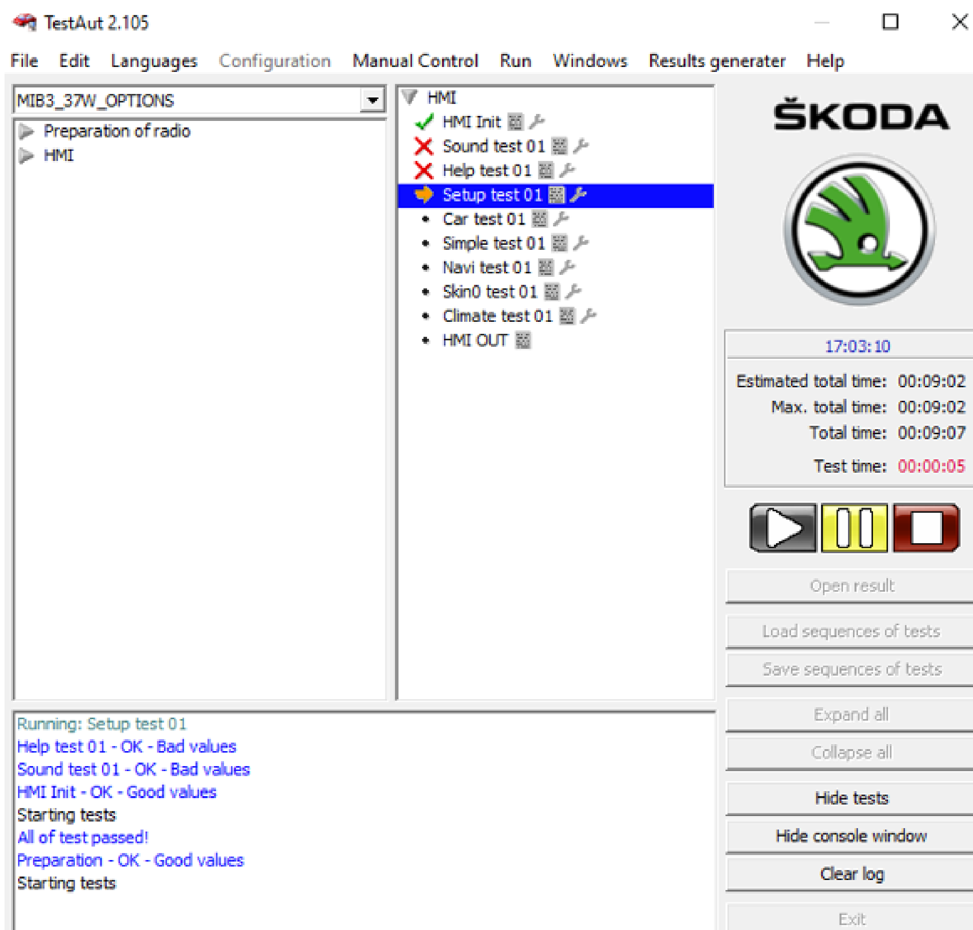
Proměnná	**Popis**
context	kontext, do kterého testcase patří
area	výřez obrazovky, jenž se bude ukládat
mode	mód, kterým bude obrazovka vyfocena (podrobný popis v kapitole 3.2.4 na straně 78)
'screenName'	skutečné jméno obrazovky, jak jí má uloženou jednotka infotainmentu
path	kroky, jenž vedou k obrazovce
'prepare'	kroky, které je třeba vykonat pro přípravu obrazovky na focení
'endCondition'	koncová podmínka pro mód sidescroll
'refVersion'	verze referenční obrazovky

Takovýto testcase vyprodukuje jeden či více obrázků, což odpovídá jednomu či více řádku ve výsledném reportu. Pro každou obrazovku je potřeba alespoň jeden testcase. V případě nastavování vícepoložkových elementů, to však zpravidla bývají testcases alespoň dva. Například na obrazovce, na které se nachází check-boxy, se jeden testcase zaměří na check-boxy v poloze ON a druhý na check-boxy v poloze OFF.

3.2.2 Spouštění testů

TestAut

Již bylo zmíněno, že hlavním ovládacím programem celého procesu automatického testování je **TestAut**, zobrazený na obr. 3.5. Ten se při svém spuštění zeptá uživatele na název platformy, která je připojena k testovacímu stavu. Bez této informace by program nevěděl, jak s jednotkou správně komunikovat. Na základě tohoto vstupu pak nastaví příslušné proměnné a spustí správné skripty Black Boxu, které jednotce simulují jednoduché zprávy. Dalším krokem po prvním spuštění systému, je zprovoznění jednotky infotainmentu. TestAut nastaví výstup ze zdroje ZUP2020 na požadovanou hodnotu a odešle pokyn Black Box PC, aby začal posílat pravidelnou zprávu s informací, že je auto nastartováno. Tuto zprávu běžně posílá řídící jednotka automobilu, která v testovacím stavu není přítomna, proto je simulována pomocí CAN zprávy generované Black Boxem.

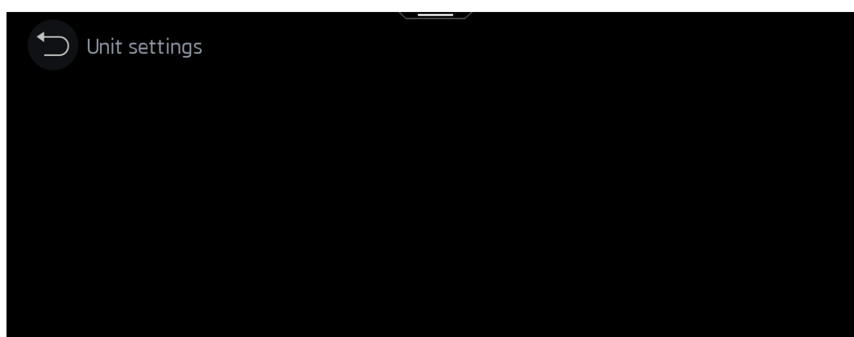


Obrázek 3.5: Program TestAut pro ovládání běhu testů

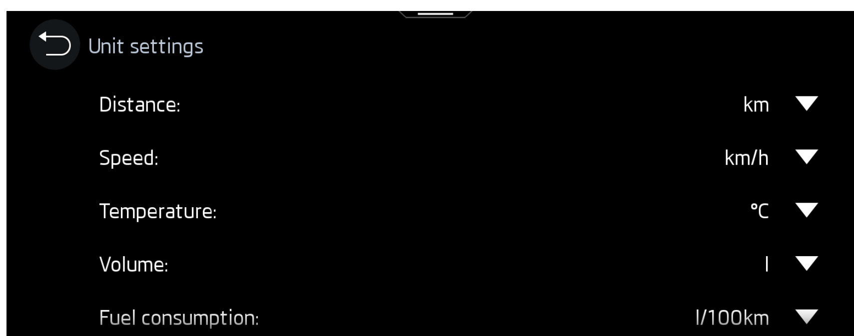
TestAut při nastavování systému využívá XML[20] soubor, ve kterém má definovány všechny potřebné údaje. Tento soubor se jmenuje **Test.xml** a mimo jiné obsahuje definici IP adres obou počítačů a Grabberu, komunikační port zdroje ZUP2020 nebo správné cesty ke knihovním a testovacím souborům. Při spuštění sekvence testů vytvoří TestAut kopii souboru Test.xml a ukládá do něj výsledky testů.

Simulace

Dalším krokem je spuštění softwaru **CANoe** a správné nastavení simulace ostatních jednotek automobilu. Tento krok je zdlouhavý a v současné chvíli není testovacím stavem automatizovaný. Po správném nastavení simulace však software CANoe funguje sám a nepřetržitě, takže do jeho vypnutí je možné jednotku infotainmentu testovat v její plné funkčnosti. Simulace, stejně jako Black Box PC, funguje na principu vysílání zpráv na CAN sběrnici. Ani restart jednotky nezpůsobí přerušení simulace. Jak je simulace pro testování důležitá je patrné z obrázků 3.6 a 3.7 na kterých je zobrazena stejná obrazovka, ale se zapnutou a vypnutou simulací.



Obrázek 3.6: Příklad obrazovky bez simulace (nastavení jednotek systému)



Obrázek 3.7: Příklad obrazovky se simulací (nastavení jednotek systému)

[20]eXtensible Markup Language - obecný značkovací jazyk.

Testy

Jsou-li všechna zařízení testovacího stavu spuštěna a běží programy TestAut a CANoe, může začít samotné testování. Soubory se samotnými testy jsou pojmenovány `HMI-`***`kontext`***`.tcl`.

1. HMI-init

Prvním testem, který musí být spuštěn před všemi ostatními je `HMI-init.tcl`. Tento soubor, přísně vzato, neobsahuje test jako takový, nýbrž inicializaci prostředí před spuštěním prvního testu. jeho součástí je:

- import knihovny HMI.tcl s globálními proměnnými a hlavními funkcemi
- nastavení globálních proměnných:
  - `HMI::unit` - připojená platforma infotainment jednotky
  - `HMI::hmiversion`, `HMI::lastversion` - aktuální a poslední verze HMI
  - `HMI::archiveDirectory` - správné cesty do archivu obrázků
  - `HMI::width`, `HMI::height` - rozlišení obrazovky
  - `HMI::action` - určuje, zda se vytváří report či nikoli
  - `HMI::python` - určuje, zda se pomocí trace kontroluje přítomnost na správné obrazovce
- import knihoven HMI-CORE a HMI-FUNCTIONS, které si podle proměnné `HMI::unit` naimportují platformně specifické soubory CORE a FUNCTIONS
- nastavení černého pozadí a angličtiny

Provedení importu knihovních funkcí vždy před spuštěním prvního testu zaručí, že pokud vývojář změní jakoukoli funkci, tak bude při běhu testu vždy aktuální.

2. HMI-*kontext*

Po provedení inicializace je na řadě samotný test, či spuštění více testů najednou. Soubor s testem je ve své podstatě velice jednoduchý. Dle `HMI::unit` naimportuje do namespace `DICT` konkrétní testcases. Ty pak postupně prochází podle jejich kontextu a vykoná příkazy zadané v proměnných `path` a `prepare`. Funkce, jež prochází slovník podle kontextu se jmenuje `GoThroughContext` a je v podstatě smyčkou `foreach` (viz. sekci 2.1.3 na straně 23) přes všechny testcases daného kontextu.

Speciálním testem je HMI-skin0.tcl, který testuje obrázky automobilů nahrané v jednotce. Každá jednotka obsahuje obrázek podle konkrétního vozu, ve kterém se nachází. Jednotka infotainmentu v testovacím módu umožňuje přepnutí právě zobrazovaného vozu. Aby každý testcase obrazovky, která obsahuje obrázek automobilu, nemusel být duplikován podle počtu automobilů, provádí vždy test HMI-skin0 změnu vozu a následně projde všechny obrazovky kontextu skin0. Pro jeden typ vozu se tak otestují všechny obrazovky obsahující obrázek automobilu a teprve potom se změní typ vozu. Následně se všechny obrazovky opět otestují s novým vzhledem, čímž se zabrání zdlouhavému přepínaní vzhledu vozu při všech testcases, které to kontrolují.

3. HMI-out

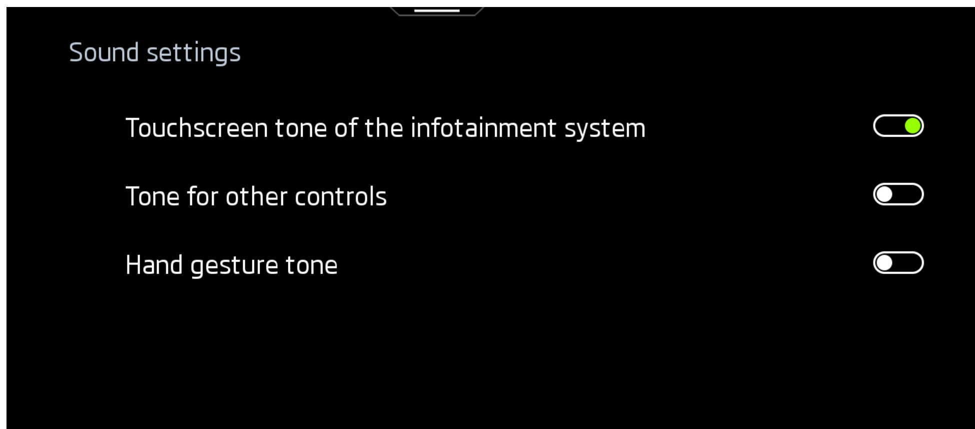
Jakmile jsou všechny testy dokončeny, je nutné provést poslední test pojmenovaný HMI-out. Opět se nejedná o test jako takový, HMI-out dokonce není definovaný ve svém vlastním souboru. Jeho kód je zapsaný přímo v souboru Test.xml, neboť není složitý ani dlouhý. Tento skript se stará o vytvoření HTML struktury výsledku. Funkce `Report` totiž výsledek ukládá také do souboru Test.xml, který následně skript HMI-out zpracuje do formy zobrazitelné HTML stránky.

3.2.3 Nastavování obrazovek

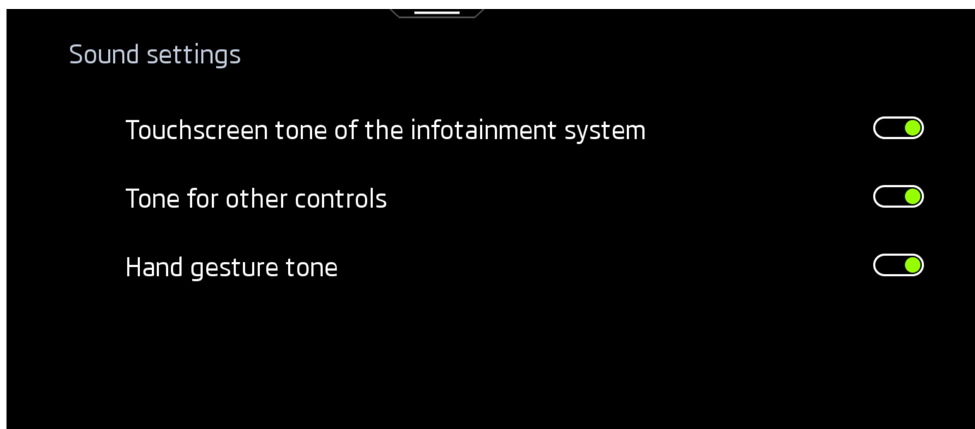
V této části jsou popsány možnosti knihovny z hlediska nastavování jednotlivých obrazovek. Nastavení obrazovky do konkrétní podoby je pro testování velmi důležité. Na obrázcích 3.8 a 3.9 je stejná obrazovka, avšak jinak nastavená. Porovnání těchto dvou obrazovek by samozřejmě vykázalo chybu, neboť obrazovky nejsou stejné. Avšak nastavit jednotku předem do konkrétní varianty a dále ji neměnit také není vhodné, neboť cílem testů je pokrytí co nejvíce stavů obrazovky. Je tedy potřeba testovat obrazovku například se zapnutými i vypnutými přepínači.

Mezi nejdůležitější funkce patří funkce, jež simulují dotek na obrazovce. Systém jednotky infotainmentu rozeznává různé druhy dotekových zpráv na CAN sběrnici. Tyto zprávy mají význam doteku (`Press`), posunu prstu po obrazovce (`Drag`) a odebrání prstu z obrazovky (`Release`). Těchto zpráv, definovaných v souborech HMI-CORE.tcl, využívají všechny ostatní implementované funkce a simulují tak doteky uživatele.

Další z velmi důležitých částí kódu jsou funkce pro nastavení obrazovky do požadovaného stavu. Tomuto účelu slouží funkce definované v souboru HMI-FUNCTIONS.tcl a jeho platformně závislých verzích.



Obrázek 3.8: Nenastavená obrazovka s přepínači

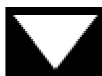


Obrázek 3.9: Nastavená obrazovka s přepínači

Principy

Hlavním principem celého testování je ovládání jednotky pomocí zpráv, které simulují dotek na určitých souřadnicích. Pomocí dalších funkcí lze ale klikat na ikonu nebo text, aniž by vývojář věděl, na jakých souřadnicích se nachází.

Pro takovéto klikání se využívá softwaru ImageMgick, který umožňuje vyhledání ikony či obrázku v jiném obrázku. Za tímto účelem je vytvořen archiv obrázků, jako je např. ikona drop-down (obr. 3.10). Podle takových obrázků vyhledávají konkrétní funkce dané elementy, které nastavují na požadovanou hodnotu. Tato funkčnost softwaru ImageMagick vrací nejen souřadnice nalezeného obrázku uvnitř vzoru, ale také procentuální shodu pixelů. Podle velikosti této shody se funkce rozhodnou, zda se hledaný element na obrazovce skutečně nachází nebo ne.



Obrázek 3.10: Ikona drop-down

Stejným principem je implementováno hledání textu v obrazu. Software ImageMagick umožňuje vytvoření libovolného textu, v zadaném fontu, velikosti, barvě a na konkrétním pozadí. Text je tedy vytvořen do podoby obrázku a dále se postupuje stejně jako při hledání obrázku ve vzoru. Příklad vytvořeného textu je znázorněn na obrázku 3.11.

Obrázek 3.11: Obrázek vytvořený podle zadaného textu

V mnoha případech se využívá principu, kdy jedna funkce slouží k nastavení jednoho elementu v určité oblasti. Této funkce následně využívá další, která nastavuje požadované elementy všechny na aktuální obrazovce. Nakonec je využita i tato funkce pro nastavení všech požadovaných elementů ve všech částech posuvné obrazovky. Následující sekce představují vždy základní nastavení elementů. Jejich přenos do funkcí, jež tento element nastavují po celé obrazovce či dokonce po celé posuvné obrazovce, je v principu vždy stejný. Základní funkce je použita vícekrát za sebou.

Klikání

Základní princip ovládání jednotky infotainmentu z hlediska uživatele vozu spočívá v přímé interakci s displejem pomocí dotyků. Nejjednodušší funkcí simulující dotyk je funkce `ClickAt`, která pošle zprávu `Press` a následně `Release` na zadaných souřadnicích displeje.

V algoritmickém pojetí testování však často může být výhodou neklikat na konkrétní souřadnice. Například pokud se obrazovka může dynamicky měnit, pak nelze zaručit, že požadovaný element bude vždy na jedněch souřadnicích. Pro tyto případy jsou implementovány funkce kliknutí na obrázek (`ImgClick`) nebo na text (`TextClick`). Tyto funkce vyhledají daný element na displeji (nebo jeho části) a následně na něj kliknou.

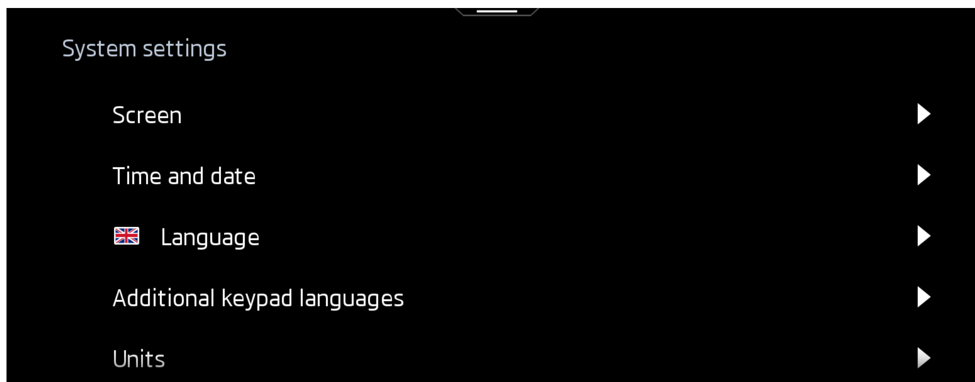
Další požadovanou variantou je i možnost přidržení prstu na jednom bodě po určitý čas. Na dlouhé podržení může různý element reagovat jinak, než na běžné kliknutí. Za tímto účelem je implementována funkce `LongPress`, která jako svůj argument kromě souřadnic přijímá i délku dotyku.

Posledním využitelným typem doteku implementovaným ve funkcích HMI je posun prstu po obrazovce. Tento případ využívá zprávu typu `Drag` pro simulaci tažení prstu po obrazovce. Funkce, jež tuto možnost vykonává je `SwipeFromTo`.

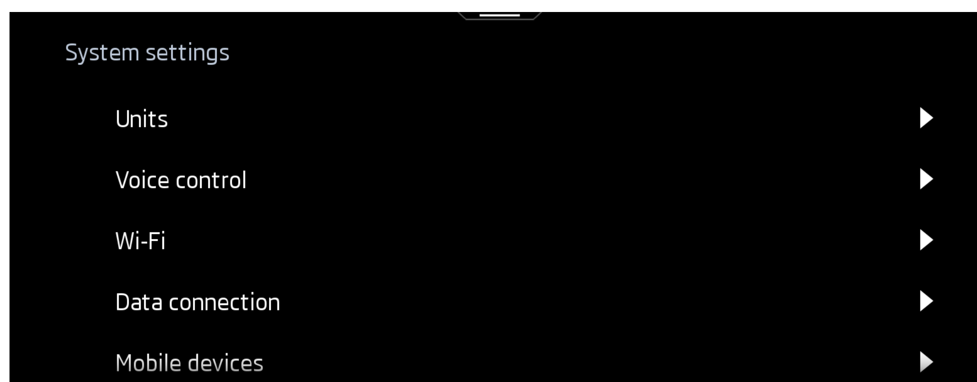
Pro případy, kdy jedno kliknutí nezpůsobí požadovaný efekt, jsou navrženy funkce, kterými jsou například funkce `FoolProofClick`, `FoolProofImgClick` a `FoolProofTextClick`. Tyto funkce odešlou zprávu o kliknutí na určitých souřadnicích jako obyčejné klikací zprávy. Poté však zkontrolují, zda se na displeji, či v nějaké oblasti displeje něco změnilo. Pokud ano, indikuje to úspěšné provedení kliknutí. Pokud ne, zkouší takové funkce kliknout znovu, dokud se jim to nepodaří nebo dokud nevyčerpají zadaný počet pokusů.

Scrollování

Pro možnost posouvání se po scrollovací obrazovce jsou definovány funkce `ScrollDownOnePage` a funkce `ScrollUpOnePage`. Tyto funkce posunou obrazovku přesně o jednu viditelnou část. Pomocí nich je tak docíleno průchodu celé takto specifické obrazovky, aniž by byla nějaká její část vynechána. Pro algoritmizaci je důležité, aby funkce procházející obrazovku začínala vždy na stejné pozici oné obrazovky. K tomuto účelu slouží funkce `ScrollToTop`, která obrazovku posune vždy na její úplný začátek. Toto chování je zajištěno kliknutím na horní kraj scrollbaru (indikátor posunu scrollovatelné obrazovky). Obrázek 3.12 ukazuje scrollovací obrazovku na svém začátku a 3.13 zobrazuje stejnou obrazovku posunutou o jednu viditelnou část níže - poslední řádek se posunul na první pozici.



Obrázek 3.12: Scrollovací obrazovka v počáteční pozici



Obrázek 3.13: Jednou posunutá scrollovací obrazovka

Toto scrollovaní je použito v řadě funkcí, které něco nastavují nebo i ve funkci, která scrollovatelnou obrazovku fotí po částech. V takových případech bylo nutné vyřešit, jak tyto funkce poznají, že se dostaly na konec takové obrazovky. Implementovány jsou zároveň dva způsoby. První z nich porovnává, zda scrollbar dosáhl svého spodního konce. Druhý způsob kontroluje průběžně jednotlivé obrázky scrollovatelné obrazovky. Pokud se stane, že nový obrázek je shodný s předchozím, je to indikátor konce obrazovky.

Switch

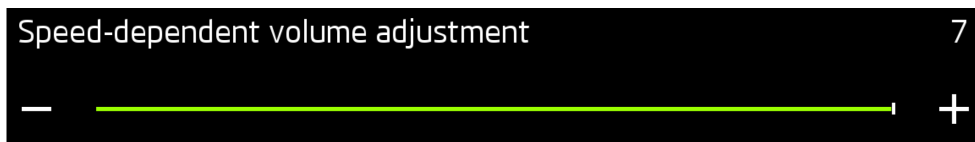
Dalším elementem obrazovky jsou přepínače, neboli switche. Tyto elementy slouží uživateli k indikaci, zda je jistá funkcionalita zapnutá či nikoli. Pro nastavení přepínačů jsou implementovány funkce nesoucí název `CheckAllBoxes` a `UncheckAllBoxes`, jež nastaví přepínače na celé obrazovce displeje. Funkce `SetAllCheckBoxes` pak slouží k nastavení přepínačů na celé scrollovatelné obrazovce. `CheckAllBoxes` hledá na displeji všechny nevybrané přepínače a kliknutím na ně je zapne. Stejně tak funguje `UncheckAllBoxes`, pouze s opačným efektem.

Obrázek 3.14: Vybraný přepínač

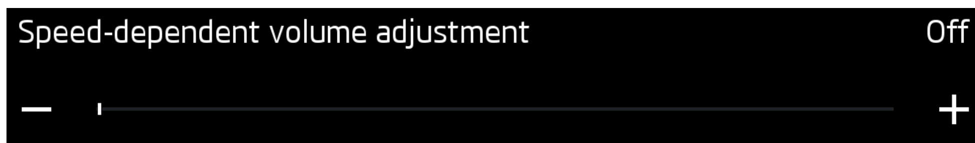
Obrázek 3.15: Nevybraný přepínač

Slider

Pomocí posuvníků, neboli sliderů, může uživatel nastavovat hodnotu nějaké proměnné v určitém rozmezí. Pro účely testování je důležité kontrolovat hlavně krajní případy nastavení sliderů. Funkce `SlideBarDetect` detekuje přítomnost slidebaru v určité části obrazovky. Učiní tak pomocí vyhledání obrázku *mínus*, kterým slidebar začíná na levém kraji, a který slouží pro ubírání hodnoty posuvníku. Po úspěšném nalezení levého kraje, hledá funkce ve stejném řádku symbol *plus*. Podle něj pak dokáže určit, kde je začátek, prostředek a konec slidebaru. Tyto souřadnice pak vrací jako svou návratovou hodnotu a funkce `SlideBarReset` může slider nastavit na požadovanou hodnotu pomocí kliknutí. Hledání symbolu plus je prováděno z důvodu existence sliderů, které nejsou dlouhé přes celou obrazovku.



Obrázek 3.16: Slider nastavený na maximum

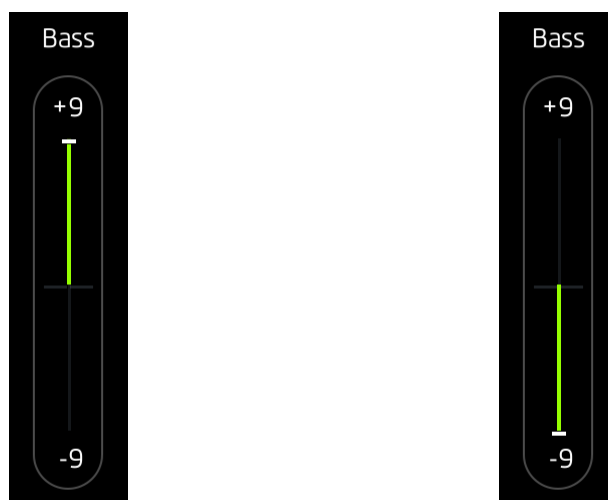


Obrázek 3.17: Slider nastavený na minimum

V systému existují slidebary s jinými symboly než je *mínus*, s tím si však tato funkce dokáže poradit. Příklad takového slideru je na obrázku 3.18, sloužící na nastavení sklonu světlometů.

Obrázek 3.18: Slider pro nastavení světlometů

Dalším specifickým sliderem jsou ekvalizéry v nastavení zvuku. O přesnou polohu těchto specifických sliderů se stará samostatná funkce s názvem `EqualiserSlidersReset`, která se při nastavování jednotlivých posuvníků orientuje podle symbolu +9 na jejich horním okraji.

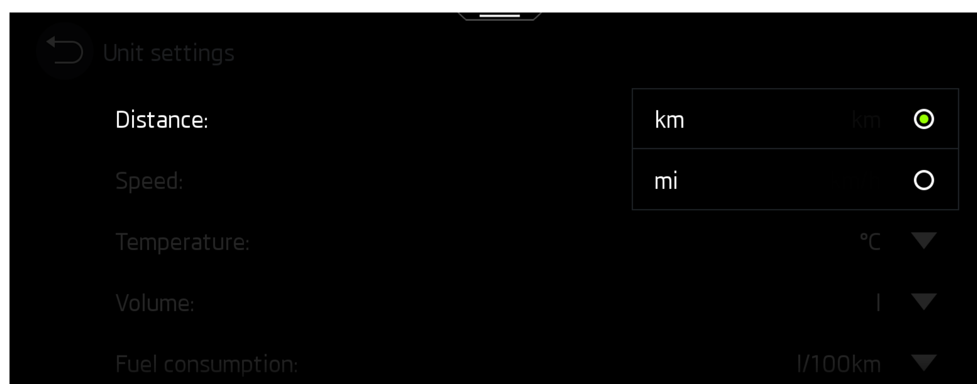


(a) Ekvalizér nastavený na maxmimum (b) Ekvalizér nastavený na minimum

Obrázek 3.19: Nastavení ekvalizéru

Drop-down

Dalším specifickým elementem obrazovek je drop-down. Tento element dovoluje uživateli nastavit proměnnou na konkrétní hodnotu z nabízeného seznamu. Tento seznam se zobrazí jako vrstva překrývající stávající obrazovku a i toto zobrazení je třeba umět nastavit a pořídit jeho snímek. Funkce jménem `DropDownSet` nastaví drop-down v určité části obrazovky do požadované pozice. Implementováno je nastavení na první volbu seznamu a na poslední. Obrázek 3.20 ukazuje, jak vypadá obrazovka 3.7 s aktivním výběrem prvního drop-down.

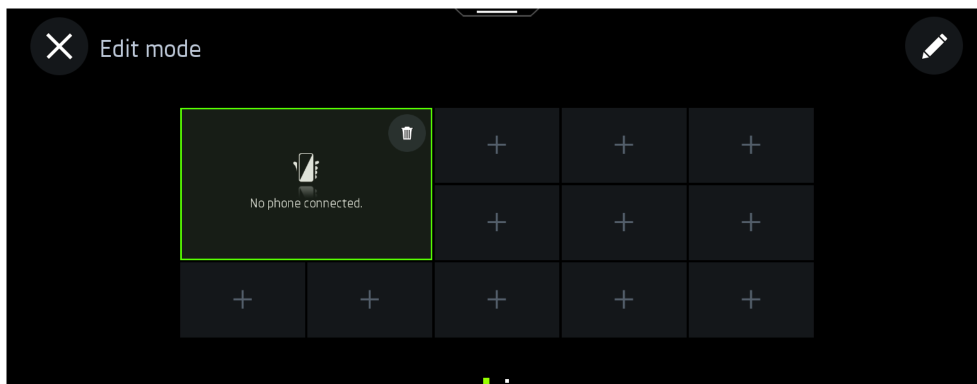


Obrázek 3.20: Obrazovka s aktivním drop-down

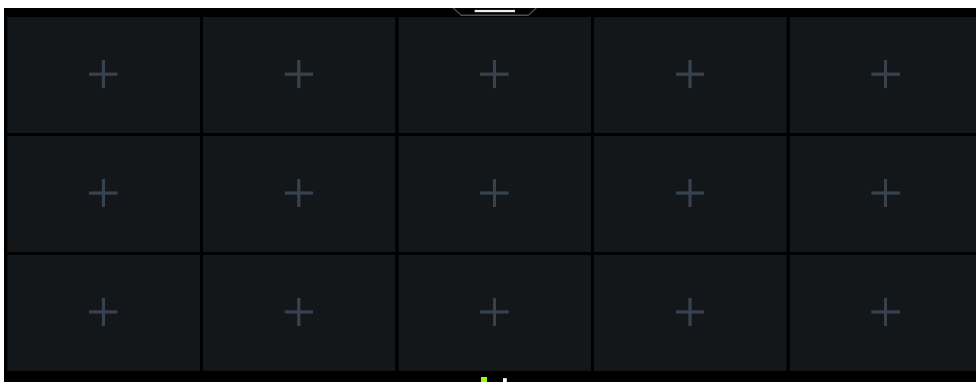
Homescreen

Velmi specifickou částí infotainment systému je domovská obrazovka. Ta se skládá se dvou až čtyř obrazovek s nastavitelnými dlaždicemi. Na místa těchto dlaždic mohou být vybrány různé widgety, například s mapou, hudbou či informacemi o vozidle. Tyto widgety je také třeba testovat, proto je implementována funkce `DeleteHomeScreen`, která smaže všechny widgety domovské obrazovky a poté jsou vybrány konkrétní položky pro testování. Obrázky 3.21, 3.22 a 3.23 ukazují proces smazání widgetů z domovské obrazovky, prázdnou domovskou obrazovku, respektive obrazovku s vybranými specifickými widgety.

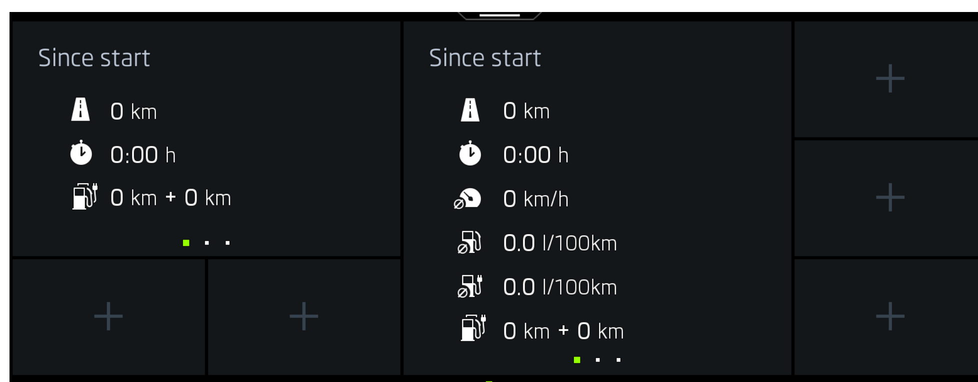
Některé widgety, jako například jízdní data z obrázku 3.23, mají samy o sobě více obrazovek. Testcases jsou samozřejmě napsané tak, aby byly otestovány všechny obrazovky i těchto widgetů.



Obrázek 3.21: Odebírání widgetů z domovské obrazovky



Obrázek 3.22: Prázdná domovská obrazovka



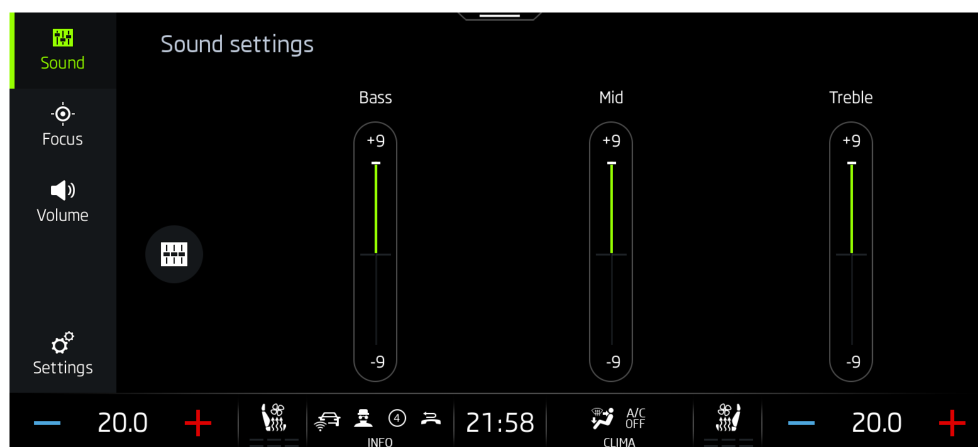
Obrázek 3.23: Domovská obrazovka s widgety o jízdních datech

3.2.4 Focení a vyhodnocení

Ve chvíli, kdy se provedou příkazy parametrů `path` a `prepare`, nachází se jednotka na obrazovce, která je připravena na focení. Každý testcase má určený mód, jak s obrazovkou v tuto chvíli naložit. Má taky určenou oblast, kterou má uložit jako výsledný obrázek. Obrazovka je často rozdělena do logických celků, které je možné a dokonce vhodné testovat nezávisle na sobě. Následující kapitola obsahuje jejich popis.

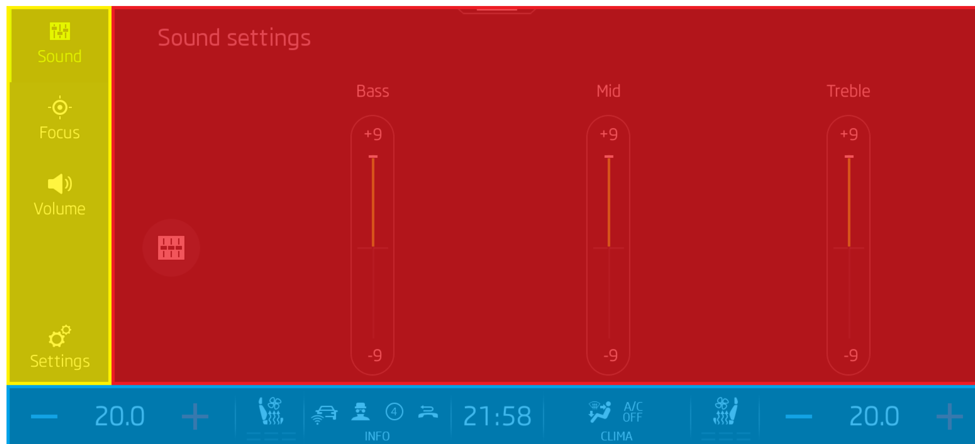
Výřezové oblasti

Speciální výřezové oblasti mají své definice uložené v určitém souboru XML, se kterým pracují funkce, jakou je například ukládání fotografií. Ta dokáže podle zadané oblasti uložit pouze odpovídající výřez obrazovky. Výřezy jsou v XML souboru uloženy jako počáteční souřadnice oblasti a její šířka a výška.



Obrázek 3.24: Celý displej

Je-li to třeba, je samozřejmě možné fotit celý displej. Ten má označení výřezové oblasti intuitivní, tedy **DISPLAY**. Ve většině případů tomu tak není. Displej se totiž skládá z různých částí, které spolu nemusí mít mnoho společného a jejich společné nastavování by bylo příliš zdlouhavé. Obrázek 3.24 ukazuje celý, neořezaný displej, stejně jako obrázek 3.25, který ovšem barvami vyznačuje ořezové oblasti.

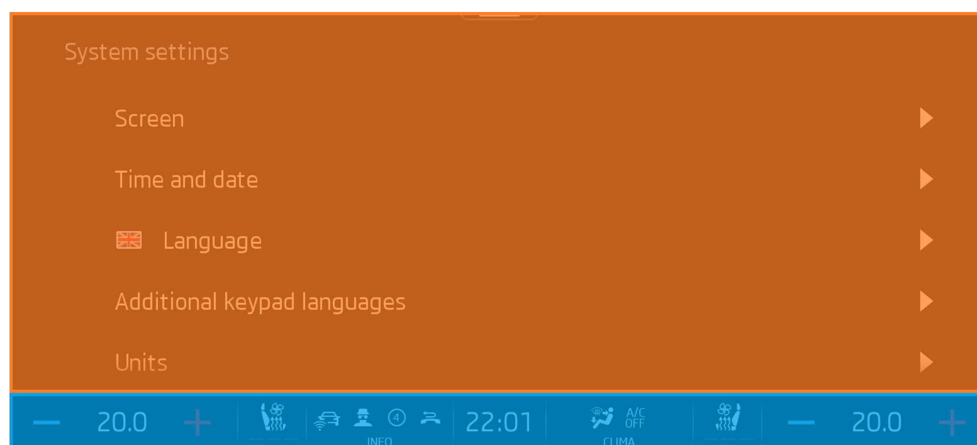


Obrázek 3.25: Celý displej s vyznačenými ořezovými oblastmi

Jednou z nich je spodní lišta, na které je zobrazena informace o klimatizaci, hodiny nebo nastavení vyhřívaných sedadel. Tato oblast displeje je pojmenována **MAINBAR**. Na obrázku 3.25 je mainbar vyznačen modrou barvou. Tato lišta je zobrazen u většiny obrazovek, které se mohou na displeji zobrazit. Z tohoto důvodu je vhodné ji k samotným obrazovkám nezahrnovat a testovat samostatně.

Žlutou barvu má tzv. **TABBAR**, kterým disponuje velké množství obrazovek. Záložky na liště tabbar uživateli zpříjemňují pohyb uvnitř seskupení obrazovek patřících k jednomu kontextu. Pokud má obrazovka ve své levé části tabbar, pak je focena pouze jako červený výřez a tabbar je testován samostatně. Červený výřez má název **SCREEN_INSIDE**.

Poslední pravidelně využívanou oblastí ořezu obrazovky je výřez jménem **WIDE_SCREEN**, neboli široký obraz. Ten se využívá v případech, kdy obrazovka nemá tabbar, jako například v nastavení nebo na domovské obrazovce. WIDE_SCREEN je znázorněn na obrázku č. 3.26 oranžovou barvou.



Obrázek 3.26: Ořezová oblast `WIDE_SCREEN`

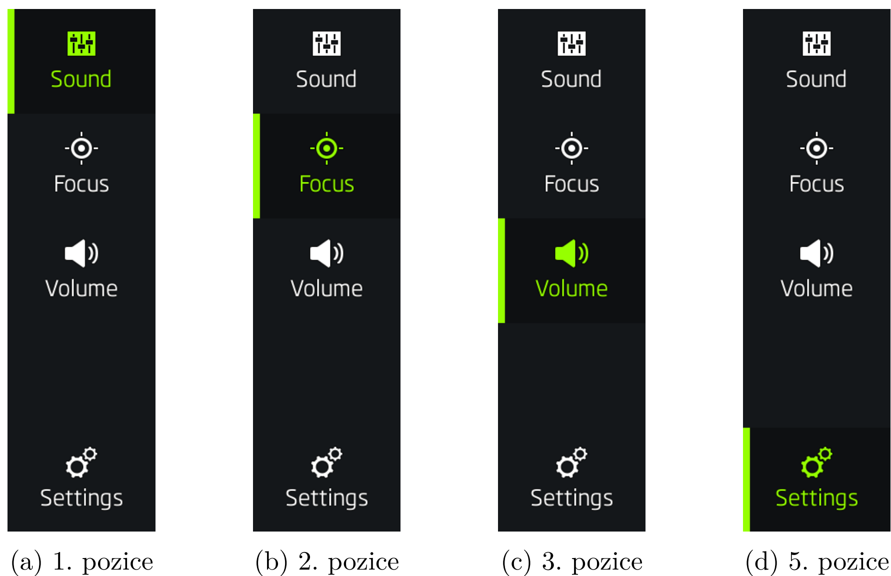
Módy focení

Každý jednotlivý testcase má určený mód focení. Ten představuje informaci pro funkce pořizující snímky obrazovek, jak mají s obrazovkou naložit. Tyto módy vyprodukují jeden nebo více snímků, podle potřeby konkrétní obrazovky. Jednotlivé módy jsou popsány v tabulce 3.3.

Tabulka 3.3: Módy pořizování snímků

Mód	Popis
classic	klasický mód, pořídí jeden snímek obrazovky
classic+drop	stejně jako předchozí mód, pořídí jeden snímek obrazovky, navíc však vyfotí všechny drop-down obrazovky v aktivním stavu
scroll	vyfotí všechny části scrollovatelné obrazovky
scroll+drop	stejně jako scroll, navíc fotí aktivní drop-down
sidescroll	určené pro obrazovky, které mají více částí horizontálně vedle sebe
tabbar	pořídí snímky všech položek tabbaru

Pro ilustraci je na obrázku č. 3.27 ukázán výstup módu `tabbar`, jenž uloží snímky všech pozic tabbaru. Aby nedošlo ke kolizi při ukládání obrázku, jsou jednotlivé snímky číslovány dle jejich pozice, např. SOUND_TABBAR_1.png. Podobně fungují i ostatní módy, které ukládají více než jeden obrázek.



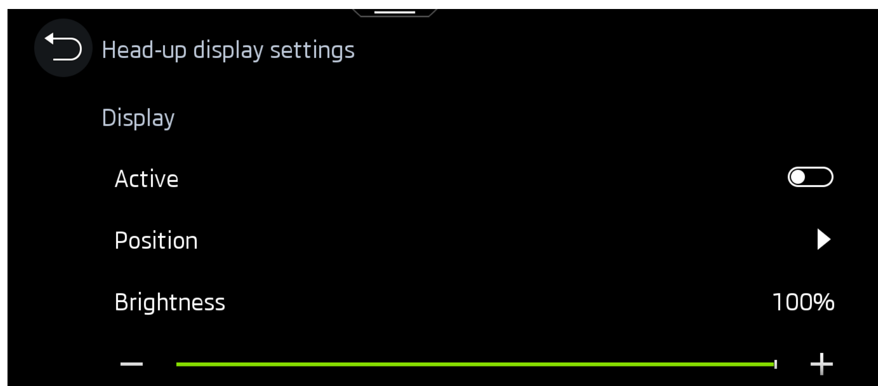
(a) 1. pozice (b) 2. pozice (c) 3. pozice (d) 5. pozice

Obrázek 3.27: Ukázka výstupu módu `tabbar`

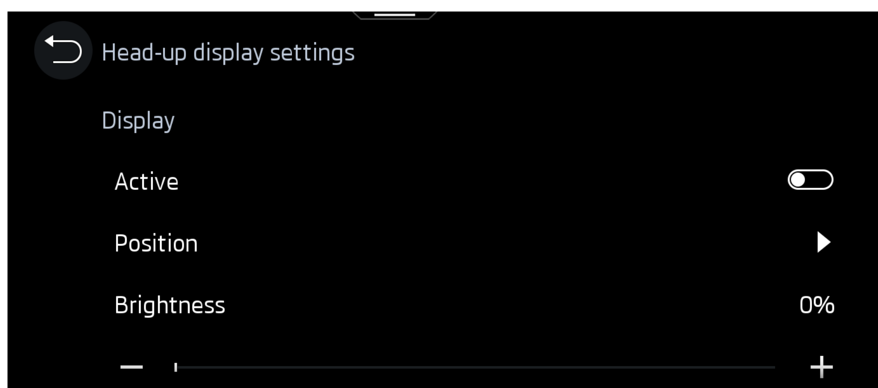
Vyhodnocení výsledku

Ve chvíli, kdy je uložen nový obrázek zbývá pouze vyhodnotit jeho správnost. Tento proces využívá opět program ImageMagick, který porovná nový obrázek s obrázkem referenčním a vrátí počet pixelů, které mají obrázky odlišné. Nejenže rozhoduje, zda jsou obrázky stejné nebo se liší, ale poskytne jako svůj výstup i obrázek s vyznačenými rozdíly. Díky této zpětné vazbě z porovnání obrázků může tester snadno objevit chybu, i přesto, že se jedná o pouhé pixely rozdílu.

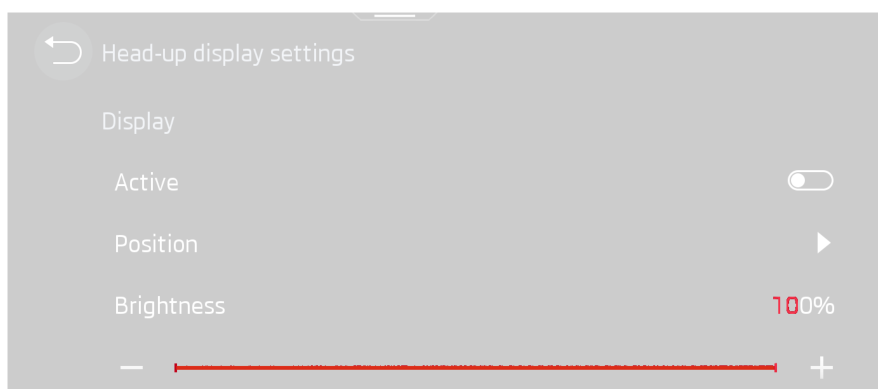
Obrázky 3.28, 3.29 a 3.30 tento proces přibližují. Zároveň tyto obrázky ilustrují důležitost správného nastavení všech elementů obrazovky před jejím focením a vyhodnocením. Špatně nastavený slider proti referenční verzi představuje rozdíl a tedy chybu, avšak chyba HMI to není. Tyto falešné chyby se v klasifikačních metodách označují jako *false negative*.



Obrázek 3.28: Referenční obrázek



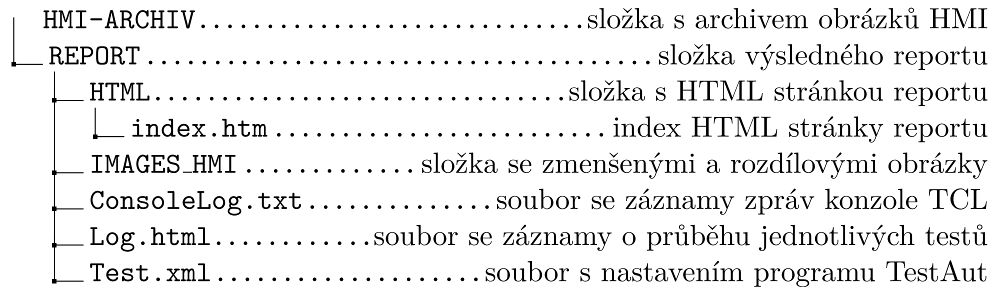
Obrázek 3.29: Nový obrázek



Obrázek 3.30: Rozdílový obrázek

3.2.5 Report

Posledním krokem před ukončením běhu testů je vytvoření přehledného reportu ve formě HTML. V této podobě má tester pohromadě všechny důležité informace, které mu může automatické testování nabídnout. Obrázek č. 3.31 ilustruje adresářovou strukturu reportu.



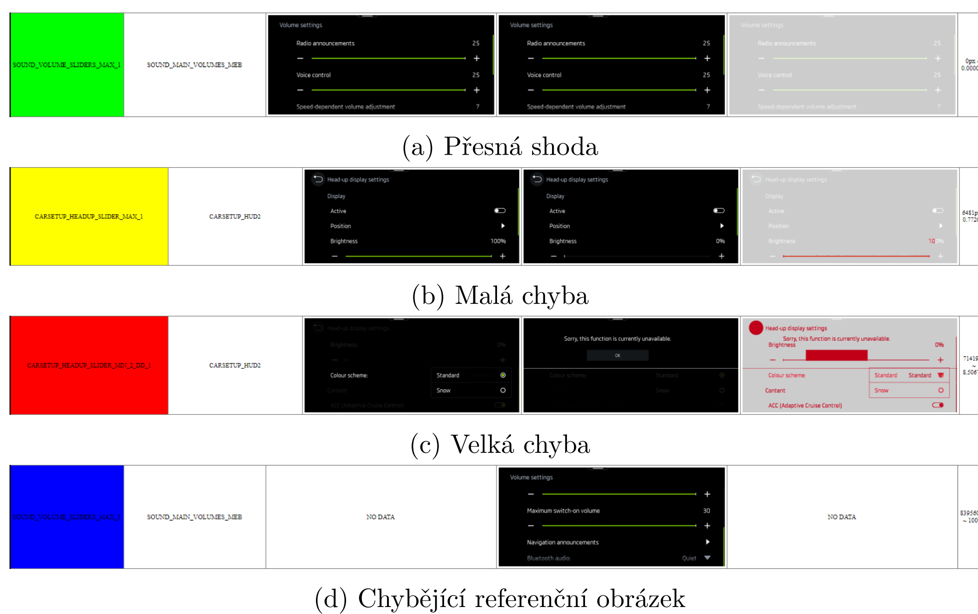
Obrázek 3.31: Adresářová struktura reportu

Prvním, podle čeho se tester orientuje je **výsledek testu** jednotlivých obrazovek. Ten je v reportu vyznačen také procentuálním rozdílem pixelů obrazovek (poslední sloupec), na první pohled jsou však řádky barevně označeny. Každý řádek má specifickou barvu, podle výsledku testu. Výsledek může mít čtyři podoby:

1. přesná shoda - označena zelenou barvou
2. malá chyba (do 1% rozdílu) - označena žlutou barvou
3. velká chyba (nad 1% rozdílu) - označena červenou barvou
4. chybějící referenční obrázek - označen modrou barvou

Dalším důležitým údajem je **název obrazovky**, jakým je označena v softwaru a ve všech systémech pro správu obrazovek. Tento údaj se nachází ve druhém sloupci. Podle tohoto názvu může tester rychle obrazovku ve vývojářském systému najít, změnit její stav na `reopened` a přidat k ní popisek chyby. O jakou chybu se jedná odhalí tester snadno, neboť má v dalších sloupcích přehledně zobrazen obrázek **reference**, **nový snímek** i **rozdílový obrázek**. Přímo ve složce reportu jsou uloženy pouze rozdílové obrázky a zmenšeniny snímků zobrazující se na hlavní stránce reportu. Obrázek v plném rozlišení si tester snadno zobrazí kliknutím na jeho zmenšeninu. Aby se zamezilo zbytečnému kopírování souborů a velké paměťové náročnosti, nachází se snímky v plném rozlišení pouze v archivu HMI, do kterého se report odkazuje.

Výsledný report pak v závislosti na spuštěných testech může být i velice dlouhý, proto je pro testera příhodné se orientovat podle barev jednotlivých řádků tabulek a zabývat se pouze snímky, na kterých automatické testování odhalilo chybu. Část reportu je ukázána na obrázku 3.33.



(a) Přesná shoda

(b) Malá chyba

(c) Velká chyba

(d) Chybějící referenční obrázek

Obrázek 3.32: Ukázka jednotlivých řádků reportu

HELP_SYSTEM_OPERATION_6	QUICK_START_GUIDE_TIP_DETAILS	Quick swipe	Quick swipe	Quick swipe	0px ~ 0.0000%
HELP_SYSTEM_OPERATION_7	QUICK_START_GUIDE_TIP_DETAILS	Advanced list functions	Advanced list functions	Advanced list functions	377px ~ 0.0449%
HELP_SYSTEM_OPERATION_8	QUICK_START_GUIDE_TIP_DETAILS	Touch Slider	Touch Slider	Touch Slider	0px ~ 0.0000%
HELP_SYSTEM_OPERATION_9	QUICK_START_GUIDE_TIP_DETAILS	NO DATA	Button assignment	NO DATA	839560px ~ 100%
HELP_VOICE_CTRL_MEDIA_1	SPEECH_TUTORIAL_TIP_ENTERTAINMENT	Music	Music	Music	0px ~ 0.0000%
HELP_VOICE_CTRL_MEDIA_2	SPEECH_TUTORIAL_TIP_ENTERTAINMENT	Station selection	Station selection	Station selection	0px ~ 0.0000%
HELP_VOICE_CTRL_MEDIA_3	SPEECH_TUTORIAL_TIP_ENTERTAINMENT	Frequency selection	Frequency selection	Frequency selection	0px ~ 0.0000%
HELP_VOICE_CTRL_MEDIA_4	SPEECH_TUTORIAL_TIP_ENTERTAINMENT	Source selection	Source selection	Source selection	0px ~ 0.0000%
HELP_VOICE_CTRL_MEDIA_5	SPEECH_TUTORIAL_TIP_ENTERTAINMENT	Music selection	Music selection	Music selection	0px ~ 0.0000%
HELP_VOICE_CTRL_MEDIA_6	SPEECH_TUTORIAL_TIP_ENTERTAINMENT	Audiobook or podcast selection	Audiobook or podcast selection	Audiobook or podcast selection	0px ~ 0.0000%
HELP_VOICE_CTRL_MEDIA_7	SPEECH_TUTORIAL_TIP_ENTERTAINMENT	Selecting a genre	Selecting a genre	Selecting a genre	0px ~ 0.0000%
HELP_VOICE_CTRL_MEDIA_8	SPEECH_TUTORIAL_TIP_ENTERTAINMENT	Other media functions	Other media functions	Other media functions	0px ~ 0.0000%
HELP_VOICE_CTRL_MEDIA_9	SPEECH_TUTORIAL_TIP_ENTERTAINMENT	Playlists	Playlists	Playlists	0px ~ 0.0000%
HELP_TABBAR_1	HELPDESKTOP_MAIN				0px ~ 0.0000%
HELP_TABBAR_2	HELPDESKTOP_MAIN				0px ~ 0.0000%

Obrázek 3.33: Ukázka konečného reportu

Kapitola 4

Testování

Kapitola nesoucí název testování, představuje proces, jakým jsou testy a funkce testovány v průběhu vývoje. Navíc obsahuje popis a řešení problémů, které v průběhu tvorby funkcí nastaly.

4.1 Princip

Princip testování vytvářených funkcí a testů je velmi přímočarý. V průběhu vývoje se funkce testuje průběžně, aby bylo dosaženo její správné funkčnosti a funkčnosti dílčích částí.

V úvodní části vývoje funkce je výhodné ji testovat přímo z konzole jazyka TCL. Program TestAut tuto konzoli spouští při svém startu a vývojář má možnost ji využít. Tímto způsobem se funkce rychle spustí a je možné okamžitě poznat, zda funguje podle očekávání.

Pokud již funkce splňuje svůj účel, je nutné ji otestovat v jednoduchém testcase. Aby bylo možné testovat jeden testcase, existuje test nesoucí název `HMI-simpleTest.tcl`, jenž umožňuje vývojáři spustit jeden nebo více testů v uměle vytvořeném kontextu `test`. Kontext `test` obsahuje většinou pouze jeden testcase, který vývojář spouští a rozhoduje, zda funguje správně. Tímto způsobem se zabrání zbytečné časové prodlevě. Přidání nového testcase např. do kontextu `car` a spuštění tohoto kontextu by bylo velmi časově náročné, neboť takový kontext obsahuje desítky testcases a jeho doba běhu činí desítky minut. Proto je vhodné nový testcase nejprve zařadit do kontextu `test` a zjistit, zda funguje, než bude zařazen na své místo do správného kontextu.

4.2 Problémy

Během implementace nového řešení se přirozeně vyskytnou překážky, jež je nutno vyřešit. Tato kapitola obsahuje popis problémů, které vyvstaly během vývoje jednotlivých funkcionalit testování.

4.2.1 Úsporný režim

V úvodu vývoje prvních funkcí nastal problém s úsporným režimem jednotky infotainmentu. Jednotka přirozeně přechází po určité době do úsporného režimu, pokud automobil není nastartován. Úsporný režim se projeví přerušením napájení displeje. Toto opatření šetří baterii vozu, která by se po dlouhém užívání infotainmentu mohla vybít. Pro testování je však nutné, aby jednotka byla stále v provozu a aby aktivně vysílala obraz na displej.

Jednotka infotainmentu dokáže poznat, zda je automobil nastartován pomocí komunikace s řídící jednotkou. Ta pravidelně zasílá zprávu o stavu motoru přes sběrnici CAN. Přestože řídící jednotka není součástí testovacího stavu, měl tento problém jednoduché řešení. Pomocí jednoho ze skriptů Black Box PC je periodicky zasílána zpráva o nastartovaném motoru do infotainment jednotky, která tak nikdy nepřejde do úsporného režimu.

4.2.2 Navození obrazovek

Jednotka infotainmentu dokáže fungovat v omezeném provozu bez ostatních jednotek automobilu. V tomto režimu však zobrazuje omezené množství obrazovek nebo je zobrazuje necelé. Simulace jednotek pomocí softwaru CANoe, jež skrze CAN sběrnici posílá zprávy, dokáže nahradit komunikaci téměř všech zařízení. Proto tento software umožňuje zdárně testovat téměř všechny obrazovky infotainmentu.

4.2.3 Uchování testcases ve slovníku

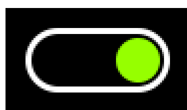
Definice testcases se ukládá do slovníku `views`, který se nachází ve jmenném prostoru `DICT`. Po přidání testcase jménem `TEST_1` do kontextu `test`, může vývojář zkoumat funkčnost tohoto testcase a pouštět ho stále dokola. V případě, kdy vývojář smaže z kódu `TEST_1` a začne se zabývat jiným testcase s názvem `TEST_2`, bude mu spouštění kontextu `test` provádět oba testcases. Děje se tak z důvodu nastavení struktury slovníku TCL, která si pamatuje všechny své položky, dokud nejsou explicitně vymazány. Za tímto účelem jsou implementovány funkce `RemoveView` a `RemoveAllContextViews`, jež odeberou ze slovníku konkrétní testcase, respektive všechny testcases daného kontextu.

4.2.4 Chybějící obrazový podklad

Všechny funkce implementované pro nastavování různých elementů obrazovky do konkrétních podob, využívají obrázků uložených v databázi. Ty slouží jako předloha pro vyhledání konkrétního elementu v obrazu displeje. Při implementování nové funkce je třeba tyto obrázky do databáze dodat.

Jedním způsobem umožňujícím zachycení obrazovky s požadovaným obrázkem je využití funkcí `V4T::Snap` a `V4T::SaveOrigElement`. Tyto funkce slouží k zachycení a uložení aktuálního obrazu a využívá jich mimo jiné i funkce `HMI::TakeScreen`.

Druhým způsobem uložení obrazovky je použití programu Grimmer. Takto uložený obrázek je třeba oříznout a požadovaný element lze následně podle nového vzoru vyhledávat. Obrázky 4.1, ale i 3.10 jsou příklady databázových předloh elementů.



(a) Switch v pozici ON

(b) Switch v pozici OFF

Obrázek 4.1: Databázové předlohy pro switch

4.2.5 Neúspěšné klikání

Dalším problémem, jenž se v průběhu tvorby testů objevil bylo neúspěšné klikání. Projevem tohoto problému bylo kliknutí, které nezpůsobilo chtěný efekt. Obrazovka tak po kliknutí zůstala stejná, jako před ním. Faktu, že kliknutí na obrazovce má nějakou odezvu, využívají funkce řešící tento problém. Jsou jimi `FoolProofClick`, `FoolProofImgClick` a `FoolProofTextClick`. Tyto funkce mají nastavený počet pokusů, po který zkoušejí kliknout v případě, že kliknutí nemělo požadovaný efekt. Že efekt kliknutí nenastal, poznají právě díky skutečnosti, že se obrazovka po kliknutí vůbec nezměnila. Díky těmto funkcím se tak nestane, že by testcase nedošel na správnou obrazovku nebo ji nenastavil do požadované podoby.

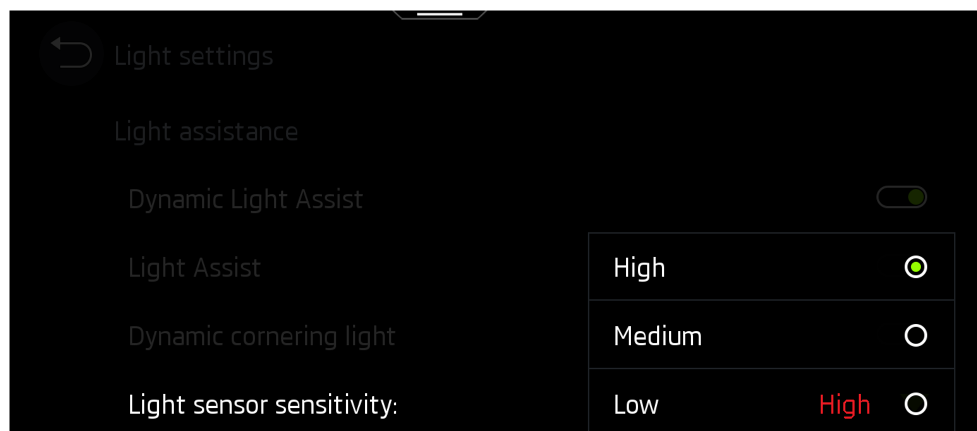
4.2.6 Scrollování

Značnou překážku ve vývoji představoval problém scrollování obrazovky. Posuvné obrazovky obsahují velké množství řádků a je třeba testovat, zda se všechny zobrazují správně. Proto se musí obrazovka postupně posouvat směrem k dalším položkám. Nejprve byla použita funkce `SwipeFromTo`, která simuluje posunutí prstu po obrazovce. Pomocí této funkce se obrazovka posouvala, avšak nebylo zaručeno, že se posune vždy o stejný počet řádků. Tento problém byl pro automatické testy závažný, neboť je nutné porovnávat vždy přesné snímky obrazovky.

Řešením toho problému se stala funkce `ScrollDownOnePage` využívající faktu, že displej platformy MQB_37W rozeznává více dotyků. První dotyk se provede na posledním řádku seznamu, aniž by se poslala zpráva o puštění prstu. Následuje dotyk na první řádek seznamu. Poté se provede uvolnění prvního dotyku. V tomto okamžiku se obrazovka posune posledním řádkem na řádek první. Poslední zprávou, kterou funkce jednotce odešle je uvolnění druhého dotyku. Tímto postupem je zajištěno přesné scrollování o jednu viditelnou část scrollovací obrazovky.

4.2.7 Překryv drop-down

Funkce infotainment systému drop-down, která umožňuje uživateli nastavit hodnotu z předem daného seznamu, se vykresluje jako vrstva přes aktuálně ukazovanou obrazovku. Tato vrstva ale obrazovku nezakryje úplně. Její průhlednost způsobí, že původní text je stále viditelný i na obrazovce s aktivním oknem drop-down. Tento jev ilustruje obrázek 4.2, který je pro lepší viditelnost graficky upraven. Slovo *High* vyznačené červenou barvou by ve skutečnosti mělo barvu šedou, podobně jako text nevybraných řádků, nicméně stále by bylo na obrázku rozeznatelné. Text nabídky drop-down se tak může objevit i na jiném místě obrazovky a funkce pro klikání na text se může špatně rozhodnout kam kliknout.



Obrázek 4.2: Ukázka překryvu obrazovky elementem drop-down

Funkce `DropDownSet` nenastavuje drop-down podle textu, ale orientuje se podle ikony vybrané položky. Prohledává řádky od prvního nebo naopak od posledního a pokud narazí na ikonu nevybrané položky, tak na ní klikne. Tím se nastaví požadovaná položka seznamu.

4.2.8 Generování reportu

V původní definici tvorby reportu se objevil problém při porovnávání s obrázkem, který neexistoval. V tomto bodě selhal program ImageMagick, který porovnávání obrázků provádí a výsledný report se vůbec nevytvořil. Proto byla do tvorby reportu přidána podmínka, kdy se nejprve kontroluje existence referenční obrazovky a teprve poté se obrazovky porovnávají. Za tím účelem byla přidána i nová funkčnost, kdy report zobrazí modrou barvou, že referenční obrazovka pro daný snímek neexistuje.

Závěr

V této diplomové práci bylo zkoumáno téma automatických testů grafického rozhraní infotainment jednotky automobilu. Jedním z dílčích cílů práce bylo teoretické seznámení se skriptovacím jazykem TCL a sběrnicí CAN. Další úkol představovalo seznámení s vývojem grafického rozhraní infotainment jednotky a důvody, jež motivují zapojení automatických testů do tohoto procesu. Následující část práce si kladla za cíl navrhnout způsob zapojení automatizace testů do stávající vývojového procesu grafického rozhraní. Dalším dílčím cílem bylo zdokumentování testovacího stavu, sloužícího pro fyzickou realizaci automatických testů. Za hlavní cíl práce bylo určeno rozšíření stávající knihovny testovacích funkcí a vytvoření testů grafického rozhraní infotainment jednotky.

Cíl práce, spočívající v přípravě teoretických informací o skriptovacím jazyce TCL a sběrnici CAN, dal základ pro jejich praktické využití při tvorbě automatických testů. Po seznámení se s principy skriptovacího jazyka TCL, jich bylo s výhodou využito při psaní konkrétních testovacích funkcí a testů. Teoretická znalost sběrnice a protokolu CAN dala možnost pochopit, jak se jednotka infotainmentu pomocí testů ovládá a jak se vytvářejí zprávy, jež jednotku ovládají.

Při seznámení se s vývojovým procesem grafického rozhraní byly představeny nejdůležitější stavy, ve kterých se jednotlivé obrazovky mohou nacházet. Zároveň byl kladen důraz na pochopení, že na obrazovce, jež zdárně prošla testovacím cyklem, se případná chyba odhalí jen velmi těžko. Na tyto argumenty navázalo představení motivace zavedení automatických testů do vývojového procesu grafického rozhraní. V této části práce byl také vysvětlen případ, kdy se na již otestované obrazovce mohou vyskytnout nové chyby.

Testovací stav pro automatické testování byl zdokumentován jak z hlediska hardware, tak i z hlediska software. Byly popsány všechny důležité součástky stavu a na dvou schématech znázorněno jejich vzájemné propojení. Jedno ze schémat ukázalo propojení jednotlivých součástek se zdrojem napájení elektrického proudu a druhé popsalo propojení součástek z hlediska jejich vzájemné datové komunikace.

Jako hlavní náplň práce byly vytvořeny knihovní funkce a testy v jazyce TCL, které slouží pro automatické testování grafického rozhraní infotainment jednotky. Před popisem procesu spouštění testů byla podrobně popsána struktura zdrojového kódu. Dále byly uvedeny jednotlivé funkce a jejich význam v nastavování konkrétních elementů displeje. Při popisu principu pořizování snímků obrazovky byly uvedeny nejen funkce, jež tyto úkony zastávají, ale také znázorněny ořezové oblasti obrazovky, které se užívají pro přehledné testování obrazovky jako celku. Následně byla představena struktura výsledného reportu testu a jeho význam pro usnadnění práce.

Poslední část diplomové práce se zabývá testováním jednotlivých funkcí a testcases. Následuje seznámení s problémy, které v průběhu vývoje nastaly a s jejich řešením.

Při vytváření této práce jsem se dozvěděl a naučil mnoho nových informací. Rozšířil jsem si povědomí nejen o programovacích principech a automatizaci testování, ale především o fungování elektroniky osobního automobilu. Tyto informace jsem využil při tvorbě testovacích funkcí a testů grafického rozhraní infotainment jednotky, jež byly zapojeny do vývojového procesu nových vozů.

V současné době využívá grafické oddělení firmy Digiteq Automotive s.r.o. automatických testů pro zjištění chyb grafického rozhraní nové Škody Octavie 4. generace. Díky tomuto principu testování jsou kontrolovány stovky obrazovek pomocí násobného počtu snímků. Každý snímek je uložen a jeho správnost vyhodnocena, což značně ulehčuje práci testera. Do budoucna se počítá s adaptací projektu automatického testování na další platformy infotainment jednotek a rozšířením počtu testovaných obrazovek a stavů jednotky infotainmentu.

Literatura

[1] Tišnovský, P.: Programovací jazyk TCL. [online], 2005, [cit. 2020-03-19]. Dostupné z: https://www.root.cz/clanky/programovaci-jazyk-tcl/

[2] Tcl Developer Xchange: Tcl/Tk Software. [online], [cit. 2020-04-10]. Dostupné z: https://www.tcl.tk/software/tcltk/

[3] Dařena, F.: *Programovací jazyk Perl*. Mendelova univerzita v Brně, první vydání, 2018, ISBN 978-80-7509-585-5.

[4] Tišnovský, P.: Programovací jazyk TCL (2). [online], 2005, [cit. 2020-03-07]. Dostupné z: https://www.root.cz/clanky/programovaci-jazyk-tcl-2/

[5] Tutorials Point: Tcl - Basic Syntax. [online], [cit. 2020-03-21]. Dostupné z: https://www.tutorialspoint.com/tcl-tk/tcl_basic_syntax.htm

[6] Tutorials Point: Tcl - Procedures. [online], [cit. 2020-03-21]. Dostupné z: https://www.tutorialspoint.com/tcl-tk/tcl_procedures.htm

[7] Tišnovský, P.: Programovací jazyk TCL (3). [online], 2005, [cit. 2020-03-21]. Dostupné z: https://www.root.cz/clanky/programovaci-jazyk-tcl-3/

[8] Tišnovský, P.: Programovací jazyk TCL (4). [online], 2005, [cit. 2020-06-09]. Dostupné z: https://www.root.cz/clanky/programovaci-jazyk-tcl-4/

[9] Tutorials Point: Tcl - Switch Statement. [online], [cit. 2020-03-23]. Dostupné z: https://www.tutorialspoint.com/tcl-tk/tcl_switch_statement.htm

[10] Tutorials Point: Tcl - Dictionary. [online], [cit. 2020-06-11]. Dostupné z: https://www.tutorialspoint.com/tcl-tk/tcl_dictionary.htm

[11] Tcl Developer Xchange: dict — Manipulate dictionaries. [online], [cit. 2020-06-11]. Dostupné z: https://www.tcl.tk/man/tcl8.6/TclCmd/dict.htm

[12] Tutorials Point: Tcl - Namespaces. [online], [cit. 2020-06-12]. Dostupné z: https://www.tutorialspoint.com/tcl-tk/tcl_namespaces.htm

[13] Tcl Developer Xchange: namespace - create and manipulate contexts for commands and variables. [online], [cit. 2020-06-12]. Dostupné z: https://www.tcl.tk/man/tcl8.4/TclCmd/namespace.htm

[14] Tcl Developer Xchange: file — Manipulate file names and attributes. [online], [cit. 2020-06-09]. Dostupné z: https://www.tcl.tk/man/tcl/TclCmd/file.htm

[15] Ing. Karel Polák: Sběrnice CAN. [online], 2003, [cit. 2020-06-14]. Dostupné z: http://www.elektrorevue.cz/clanky/03021/index.html

[16] AndyMark: CAN Bus Wire, 10ft. [online], [cit. 2020-07-12]. Dostupné z: https://andymark-weblinc.netdna-ssl.com/product_images/can-bus-cable-10ft/5bd35a6661a10d295496434a/zoom.jpg?c=1540577894

[17] Kvaser: CAN Bus Connectors. [online], [cit. 2020-07-12]. Dostupné z: https://www.kvaser.com/wp-content/uploads/2014/01/can-connectors-1.gif

[18] Elektrorevue: Datová zpráva podle specifikace CAN 2.0A. [online], 2003, [cit. 2020-06-27]. Dostupné z: http://www.elektrorevue.cz/clanky/03021/obr3.png

[19] Elektrorevue: Začátek datové zprávy (standardní formát) podle specifikace 2.0B. [online], 2003, [cit. 2020-06-27]. Dostupné z: http://www.elektrorevue.cz/clanky/03021/obr4.png

[20] Elektrorevue: Začátek datové zprávy (rozšířený formát) podle specifikace 2.0B. [online], 2003, [cit. 2020-06-27]. Dostupné z: http://www.elektrorevue.cz/clanky/03021/obr5.png

[21] Alza.cz: Pracovní HP PC. [online], [cit. 2020-07-18]. Dostupné z: https://i.alza.cz/Foto/ImgGalery/Image/pracovni%20HP%20PC.jpg

[22] Mironet: CHIEFTEC MiniT FI-02BC-U3. [online], [cit. 2020-07-18]. Dostupné z: https://www.mironet.cz/Foto/s4/92430296.jpg

[23] PPM Power: ZUP200. [online], [cit. 2020-07-18]. Dostupné z: https://ppmpower.co.uk/wp-content/uploads/ZUP-200.jpg

[24] GM Electronic: Spínaný zdroj MEAN WELL NDR-120-12. [online], [cit. 2020-07-18]. Dostupné z: https://www.gme.cz/data/product/1024_1024/pctdetail.751-998.1.jpg

[25] D-Link: DSG-1100-08. [online], [cit. 2020-07-18]. Dostupné z: https://www.dlink.com/-/media/global-product-images/business/switches/smart-managed-switches/dgs-1100-08/product-preview/dgs-1100-08_side.png

[26] tp-link: TL-SF1005D. [online], [cit. 2020-07-18]. Dostupné z: https://static.tp-link.com/TL-SF1005D-02_1481511184673c.jpg

[27] auto-navi: Škoda SuperB III - Columbus MIB2.5 originální navigace se sim slotem. [online], [cit. 2020-07-18]. Dostupné z: https://img.auto-navi.cz/images/%C5%A0koda/%C5%A0koda%20SuperB%20III%20-%20Columbus%20MIB2.5%20origin%C3%A1ln%C3%AD%20navigace%20se%20SIM%20slotem%209.jpg?vid=1&tid=35&r=B

[28] Bazar.AutoMedik.cz: MODUL-GATEWAY-5Q0907530A. [online], [cit. 2020-07-18]. Dostupné z: https://6.allegroimg.com/s360/01b172/80c5580e4005956fe1ff2a077116/MODUL-GATEWAY-5Q0907530A

[29] Digiteq Automotive: MGB Grabber. [online], [cit. 2020-07-18]. Dostupné z: https://www.digiteqautomotive.com/sites/default/files/styles/490x490_produkt_detail/public/2019-05/_DSC1645_jpg_2500px-min.jpg?itok=DjQ8L-KI

[30] auto-navi: Škoda SuperB III - Columbus MIB2.5 originální navigace se sim slotem. [online], [cit. 2020-07-18]. Dostupné z: https://img.auto-navi.cz/images/%C5%A0koda/%C5%A0koda%20SuperB%20III%20-%20Columbus%20MIB2.5%20origin%C3%A1ln%C3%AD%20navigace%20se%20SIM%20slotem%203.jpg?tid=35

[31] Vector: VH6501. [online], [cit. 2020-07-18]. Dostupné z: https://eenews.cdnartwhere.eu/sites/default/files/styles/inner_article/public/sites/default/files/images/vector_1.jpg?itok=a_utUQPy

[32] Image Magick: Home. [online], 2020, [cit. 2020-07-16]. Dostupné z: https://imagemagick.org/

[33] Škoda Auto: Digitální technologie: Moderní infotainment systémy a rozsáhlé možnosti konektivity. [online], [cit. 2020-07-18]. Dostupné z: https://cdn.skoda-storyboard.com/2019/11/OCTAVIA_078.jpg

Příloha A

Seznam použitých zkratek

ABT Anzeigebedienteil

ASCII American Standard Code for Information Interchange

CAN Controller Area Network

CRC Cyclic Redundancy Check

GUI Graphical User Interface

HMI Human Machine Interface

HTML Hypertext Markup Language

IoT Internet of Things

IP Internet Protocol

LAN Local Area Network

LIN Local Interconnect Network

LVDS Low-Voltage Differential Signaling

OOP Object-Oriented programming

SSH Secure Shell

TCL Tool Command Language

USB Universal Serial Bus

VW Volkswagen

WAN Wide Area Network

XML eXtensible Markup Language

Příloha **B**

Obsah přiloženého CD

```
readme.txt ........................ stručný popis obsahu CD
Example_Report_HMI ........... adresář s ukázkovám reportem
HMI-ARCHIV ..................... adresář s archivem obrázků
src
  code ............. adresář se zdrojovými kódy implementace
  obrazky ........ adresář s obrázky použitými v textu práce
  DP_Kubat_Jan_2020.tex ...... zdrojová forma práce ve formátu LaTeX
text ............................................ text práce
  DP_Kubat_Jan_2020.pdf ............ text práce ve formátu PDF
```